GLOBAL EDITION

Software Engineering

TENTH EDITION

Ian Sommerville

ALWAYS LEARNING

PEARSON

SOFTWARE ENGINEERING

Tenth Edition

Ian Sommerville

PEARSON

Boston Columbus Indianapolis New York San Francisco Hoboken
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editorial Director: Marcia Horton
Editor in Chief: Michael Hirsch
Acquisitions Editor: Matt Goldstein
Editorial Assistant: Chelsea Bell
Assistant Acquisitions Editor, Global Edition: Murchana Borthakur
Associate Project Editor, Global Edition: Binita Roy
Managing Editor: Jeff Holcomb
Senior Production Project Manager: Marilyn Lloyd
Director of Marketing: Margaret Waples
Marketing Coordinator: Kathryn Ferranti
Senior Manufacturing Buyer: Carol Melville
Senior Manufacturing Controller, Production, Global Edition: Trudy Kimber
Text Designer: Susan Raymond
Cover Art Designer: Lumina Datamatics
Cover Image: © Andrey Bayda/Shutterstock
Interior Chapter Opener: © graficart.net/Alamy
Full-Service Project Management: Rashmi Tickyani, Aptara®, Inc.
Composition and Illustrations: Aptara®, Inc.

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsonglobaleditions.com

© Pearson Education Limited 2016

The rights of Ian Sommerville to be identified as the author of this work have been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

Authorized adaptation from the United States edition, entitled Software Engineering, 10th edition, ISBN 978-0-13-394303-0, by Ian Sommerville, published by Pearson Education © 2016.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a license permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS.

All trademarks used herein are the property of their respective owners. The use of any trademark in this text does not vest in the author or publisher any trademark ownership rights in such trademarks, nor does the use of such trademarks imply any affiliation with or endorsement of this book by such owners.

ISBN 10: 1-292-09613-6
ISBN 13: 978-1-292-09613-1

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

10 9 8 7 6 5 4 3 2 1

Typeset in 9 New Aster LT Std by Aptara®, Inc.

Printed and bound by Courier Westford in the United States of America.

PREFACE

Progress in software engineering over the last 50 years has been astonishing. Our societies could not function without large professional software systems. National utilities and infrastructure—energy, communications and transport—all rely on complex and mostly reliable computer systems. Software has allowed us to explore space and to create the World Wide Web—the most significant information system in the history of mankind. Smartphones and tablets are ubiquitous and an entire 'apps industry' developing software for these devices has emerged in the past few years.

Humanity is now facing a demanding set of challenges—climate change and extreme weather, declining natural resources, an increasing world population to be fed and housed, international terrorism, and the need to help elderly people lead satisfying and fulfilled lives. We need new technologies to help us address these challenges and, for sure, software will have a central role in these technologies. Software engineering is, therefore, critically important for our future on this planet. We have to continue to educate software engineers and develop the discipline so that we meet the demand for more software and create the increasingly complex future systems that we need.

Of course, there are still problems with software projects. Systems are still sometimes delivered late and cost more than expected. We are creating increasingly complex software systems of systems and we should not be surprised that we encounter difficulties along the way. However, we should not let these problems conceal the real successes in software engineering and the impressive software engineering methods and technologies that have been developed.

This book, in different editions, has now been around for over 30 years and this edition is based around the essential principles that were established in the first edition:

1. I write about software engineering as it is practiced in industry, without taking an evangelical position on particular approaches such as agile development or formal methods. In reality, industry mixes techniques such as agile and plan-based development and this is reflected in the book.

2. I write about what I know and understand. I have had many suggestions for additional topics that might be covered in more detail such as open source development, the use of the UML and mobile software engineering. But I don't really know enough about these areas. My own work has been in system dependability and in systems engineering and this is reflected in my selection of advanced topics for the book.

I believe that the key issues for modern software engineering are managing complexity, integrating agility with other methods and ensuring that our systems are secure and resilient. These issues have been the driver for the changes and additions in this new edition of my book.

Changes from the 9th edition

In summary, the major updates and additions in this book from the 9th edition are:

- I have extensively updated the chapter on agile software engineering, with new material on Scrum. I have updated other chapters as required to reflect the increasing use of agile methods of software engineering.
- I have added new chapters on resilience engineering, systems engineering, and systems of systems.
- I have completely reorganized three chapters covering reliability, safety, and security.
- I have added new material on RESTful services to the chapter covering service-oriented software engineering.
- I have revised and updated the chapter on configuration management with new material on distributed version control systems.
- I have moved chapters on aspect-oriented software engineering and process improvement from the print version of the book to the web site.
- New supplementary material has been added to the web site, including a set of supporting videos. I have explained key topics on video and recommended related YouTube videos.

The 4-part structure of the book, introduced in earlier editions, has been retained but I have made significant changes in each part of the book.

1. In Part 1, Introduction to software engineering, I have completely rewritten Chapter 3 (agile methods) and updated this to reflect the increasing use of Scrum. A new case study on a digital learning environment has been added to Chapter 1 and is used in a number of chapters. Legacy systems are covered in more detail in Chapter 9. Minor changes and updates have been made to all other chapters.

2. Part 2, which covers dependable systems, has been revised and restructured. Rather than an activity-oriented approach where information on safety, security and reliability is spread over several chapters, I have reorganized this so that each topic has a chapter in its own right. This makes it easier to cover a single topic, such as security, as part of a more general course. I have added a completely new chapter on resilience engineering which covers cybersecurity, organizational resilience, and resilient systems design.

3. In Part 3, I have added new chapters on systems engineering and systems of systems and have extensively revised the material on service-oriented systems engineering to reflect the increasing use of RESTful services. The chapter on aspect-oriented software engineering has been deleted from the print version but remains available as a web chapter.

4. In Part 4, I have updated the material on configuration management to reflect the increasing use of distributed version control tools such as Git. The chapter on process improvement has been deleted from the print version but remains available as a web chapter.

An important change in the supplementary material for the book is the addition of video recommendations in all chapters. I have made over 40 videos on a range of topics that are available on my YouTube channel and linked from the book's web pages. In cases where I have not made videos, I have recommended YouTube videos that may be useful.

I explain the rationale behind the changes that I've made in this short video:

http://software-engineering-book/videos/10th-edition-changes

Readership

The book is primarily aimed at university and college students taking introductory and advanced courses in software and systems engineering. I assume that readers understand the basics of programming and fundamental data structures.

Software engineers in industry may find the book useful as general reading and to update their knowledge on topics such as software reuse, architectural design, dependability and security and systems engineering.

Using the book in software engineering courses

I have designed the book so that it can be used in three different types of software engineering course:

1. *General introductory courses in software engineering*. The first part of the book has been designed to support a 1-semester course in introductory software engineering. There are 9 chapters that cover fundamental topics in software engineering.

If your course has a practical component, management chapters in Part 4 may be substituted for some of these.

2. *Introductory or intermediate courses on specific software engineering topics.* You can create a range of more advanced courses using the chapters in parts 2–4. For example, I have taught a course in critical systems using the chapters in Part 2 plus chapters on systems engineering and quality management. In a course covering software-intensive systems engineering, I used chapters on systems engineering, requirements engineering, systems of systems, distributed software engineering, embedded software, project management and project planning.

3. *More advanced courses in specific software engineering topics.* In this case, the chapters in the book form a foundation for the course. These are then supplemented with further reading that explores the topic in more detail. For example, a course on software reuse could be based around Chapters 15–18.

Instructors may access additional teaching support material from Pearson's website. Some of this is password-protected and instructors using the book for teaching can obtain a password by registering at the Pearson website. The material available includes:

- Model answers to selected end of chapter exercises.
- Quiz questions and answers for each chapter.

You can access this material at:

www.pearsonglobaleditions.com/Sommerville

Book website

This book has been designed as a hybrid print/web text in which core information in the printed edition is linked to supplementary material on the web. Several chapters include specially written 'web sections' that add to the information in that chapter. There are also six 'web chapters' on topics that I have not covered in the print version of the book.

You can download a wide range of supporting material from the book's website (software-engineering-book.com) including:

- A set of videos where I cover a range of software engineering topics. I also recommend other YouTube videos that can support learning.
- An instructor's guide that gives advice on how to use the book in teaching different courses.
- Further information on the book's case studies (insulin pump, mental health care system, wilderness weather system, digital learning system), as well other case studies, such as the failure of the Ariane 5 launcher.

- Six web chapters covering process improvement, formal methods, interaction design, application architectures, documentation and aspect-oriented development.
- Web sections that add to the content presented in each chapter. These web sections are linked from breakout boxes in each chapter.
- PowerPoint presentations for all of the chapters in the book and additional PowerPoint presentations covering a range of systems engineering topics are available at pearsonglobaleditions.com/Sommerville.

In response to requests from users of the book, I have published a complete requirements specification for one of the system case studies on the book's web site. It is difficult for students to get access to such documents and so understand their structure and complexity. To avoid confidentiality issues, I have re-engineered the requirements document from a real system so there are no restrictions on its use.

Contact details

Website: software-engineering-book.com
Email: name: software.engineering.book; domain: gmail.com
Blog: iansommerville.com/systems-software-and-technology
YouTube: youtube.com/user/SoftwareEngBook
Facebook: facebook.com/sommerville.software.engineering
Twitter: @SoftwareEngBook or @iansommerville (for more general tweets)

Follow me on Twitter or Facebook to get updates on new material and comments on software and systems engineering.

Acknowledgements

A large number of people have contributed over the years to the evolution of this book and I'd like to thank everyone (reviewers, students and book users) who have commented on previous editions and made constructive suggestions for change. I'd particularly like to thank my family, Anne, Ali, and Jane, for their love, help and support while I was working on this book (and all of the previous editions).

Ian Sommerville,
September 2014

Contents at a glance

	Preface	3
Part 1	Introduction to Software Engineering	15
Chapter 1	Introduction	17
Chapter 2	Software processes	43
Chapter 3	Agile software development	72
Chapter 4	Requirements engineering	101
Chapter 5	System modeling	138
Chapter 6	Architectural design	167
Chapter 7	Design and implementation	196
Chapter 8	Software testing	226
Chapter 9	Software evolution	255
Part 2	System Dependability and Security	283
Chapter 10	Dependable systems	285
Chapter 11	Reliability engineering	306
Chapter 12	Safety engineering	339
Chapter 13	Security engineering	373
Chapter 14	Resilience engineering	408
Part 3	Advanced Software Engineering	435
Chapter 15	Software reuse	437
Chapter 16	Component-based software engineering	464
Chapter 17	Distributed software engineering	490
Chapter 18	Service-oriented software engineering	520
Chapter 19	Systems engineering	551
Chapter 20	Systems of systems	580
Chapter 21	Real-time software engineering	610
Part 4	Software Management	639
Chapter 22	Project management	641
Chapter 23	Project planning	667
Chapter 24	Quality management	700
Chapter 25	Configuration management	730
	Glossary	757
	Subject index	777
	Author index	803

Pearson wishes to thank and acknowledge the following people for their work on the Global Edition:

Contributor

Sherif G. Aly, The American University in Cairo
Muthuraj M., Android developer

Reviewers

Mohit P. Tahiliani, National Institute of Technology Karnataka, Surathkal
Chitra Dhawale, P. R. Patil Group of Educational Institutes, Amravati
Sanjeevni Shantaiya, Disha Institute of Management & Technology

CONTENTS

Preface 3

Part 1 Introduction to Software Engineering 15

Chapter 1 Introduction 17

1.1 Professional software development 19
1.2 Software engineering ethics 28
1.3 Case studies 31

Chapter 2 Software processes 43

2.1 Software process models 45
2.2 Process activities 54
2.3 Coping with change 61
2.4 Process improvement 65

Chapter 3 Agile software development 72

3.1 Agile methods 75
3.2 Agile development techniques 77
3.3 Agile project management 84
3.4 Scaling agile methods 88

Chapter 4 Requirements engineering 101

4.1 Functional and non-functional requirements 105
4.2 Requirements engineering processes 111
4.3 Requirements elicitation 112
4.4 Requirements specification 120
4.5 Requirements validation 129
4.6 Requirements change 130

Chapter 5 System modeling 138

5.1 Context models 141
5.2 Interaction models 144
5.3 Structural models 149
5.4 Behavioral models 154
5.5 Model-driven architecture 159

Chapter 6 Architectural design 167

6.1 Architectural design decisions 171
6.2 Architectural views 173
6.3 Architectural patterns 175
6.4 Application architectures 184

Chapter 7 Design and implementation 196

7.1 Object-oriented design using the UML 198
7.2 Design patterns 209
7.3 Implementation issues 212
7.4 Open-source development 219

Chapter 8 Software testing 226

8.1 Development testing 231
8.2 Test-driven development 242

8.3 Release testing 245
8.4 User testing 249

Chapter 9 Software evolution 255

9.1 Evolution processes 258
9.2 Legacy systems 261
9.3 Software maintenance 270

Part 2 System Dependability and Security 283

Chapter 10 Dependable systems 285

10.1 Dependability properties 288
10.2 Sociotechnical systems 291
10.3 Redundancy and diversity 295
10.4 Dependable processes 297
10.5 Formal methods and dependability 299

Chapter 11 Reliability engineering 306

11.1 Availability and reliability 309
11.2 Reliability requirements 312
11.3 Fault-tolerant architectures 318
11.4 Programming for reliability 325
11.5 Reliability measurement 331

Chapter 12 Safety engineering 339

12.1 Safety-critical systems 341
12.2 Safety requirements 344
12.3 Safety engineering processes 352
12.4 Safety cases 361

Chapter 13 Security engineering 373

13.1 Security and dependability 376
13.2 Security and organizations 380
13.3 Security requirements 382
13.4 Secure systems design 388
13.5 Security testing and assurance 402

Chapter 14 Resilience engineering 408

14.1 Cybersecurity 412
14.2 Sociotechnical resilience 416
14.3 Resilient systems design 424

Part 3 Advanced Software Engineering 435

Chapter 15 Software reuse 437

15.1 The reuse landscape 440
15.2 Application frameworks 443
15.3 Software product lines 446
15.4 Application system reuse 453

Chapter 16 Component-based software engineering 464

16.1 Components and component models 467
16.2 CBSE processes 473
16.3 Component composition 480

Chapter 17 Distributed software engineering 490

17.1 Distributed systems 492
17.2 Client–server computing 499

17.3 Architectural patterns for distributed systems 501
17.4 Software as a service 512

Chapter 18 Service-oriented software engineering 520

18.1 Service-oriented architecture 524
18.2 RESTful services 529
18.3 Service engineering 533
18.4 Service composition 541

Chapter 19 Systems engineering 551

19.1 Sociotechnical systems 556
19.2 Conceptual design 563
19.3 System procurement 566
19.4 System development 570
19.5 System operation and evolution 574

Chapter 20 Systems of systems 580

20.1 System complexity 584
20.2 Systems of systems classification 587
20.3 Reductionism and complex systems 590
20.4 Systems of systems engineering 593
20.5 Systems of systems architecture 599

Chapter 21 Real-time software engineering 610

21.1 Embedded system design 613
21.2 Architectural patterns for real-time software 620
21.3 Timing analysis 626
21.4 Real-time operating systems 631

Part 4 Software Management 639

Chapter 22 Project management 641

22.1 Risk management 644
22.2 Managing people 652
22.3 Teamwork 656

Chapter 23 Project planning 667

23.1 Software pricing 670
23.2 Plan-driven development 672
23.3 Project scheduling 675
23.4 Agile planning 680
23.5 Estimation techniques 682
23.6 COCOMO cost modeling 686

Chapter 24 Quality management 700

24.1 Software quality 703
24.2 Software standards 706
24.3 Reviews and inspections 710
24.4 Quality management and agile development 714
24.5 Software measurement 716

Chapter 25 Configuration management 730

25.1 Version management 735
25.2 System building 740
25.3 Change management 745
25.4 Release management 750

Glossary 757
Subject index 777
Author index 803

PART 1 Introduction to Software Engineering

My aim in this part of the book is to provide a general introduction to software engineering. The chapters in this part have been designed to support a one-semester first course in software engineering. I introduce important concepts such as software processes and agile methods, and describe essential software development activities, from requirements specification through to system evolution.

Chapter 1 is a general introduction that introduces professional software engineering and defines some software engineering concepts. I have also included a brief discussion of ethical issues in software engineering. It is important for software engineers to think about the wider implications of their work. This chapter also introduces four case studies that I use in the book. These are an information system for managing records of patients undergoing treatment for mental health problems (Mentcare), a control system for a portable insulin pump, an embedded system for a wilderness weather station and a digital learning environment (iLearn).

Chapters 2 and 3 cover software engineering processes and agile development. In Chapter 2, I introduce software process models, such as the waterfall model, and I discuss the basic activities that are part of these processes. Chapter 3 supplements this with a discussion of agile development methods for software engineering. This chapter had been

extensively changed from previous editions with a focus on agile development using Scrum and a discussion of agile practices such as stories for requirements definition and test-driven development.

The remaining chapters in this part are extended descriptions of the software process activities that are introduced in Chapter 2. Chapter 4 covers the critically important topic of requirements engineering, where the requirements for what a system should do are defined. Chapter 5 explains system modeling using the UML, where I focus on the use of use case diagrams, class diagrams, sequence diagrams and state diagrams for modeling a software system. In Chapter 6, I discuss the importance of software architecture and the use of architectural patterns in software design.

Chapter 7 introduces object oriented design and the use of design patterns. I also introduce important implementation issues here—reuse, configuration management and host-target development and discuss open source development. Chapter 8 focuses on software testing from unit testing during system development to the testing of software releases. I also discuss the use of test-driven development—an approach pioneered in agile methods but which has wide applicability. Finally, Chapter 9 presents an overview of software evolution issues. I cover evolution processes, software maintenance and legacy system management.

1 Introduction

Objectives

The objectives of this chapter are to introduce software engineering and to provide a framework for understanding the rest of the book. When you have read this chapter, you will:

- understand what software engineering is and why it is important;
- understand that the development of different types of software system may require different software engineering techniques;
- understand ethical and professional issues that are important for software engineers;
- have been introduced to four systems, of different types, which are used as examples throughout the book.

Contents

1.1 Professional software development
1.2 Software engineering ethics
1.3 Case studies

Software engineering is essential for the functioning of government, society, and national and international businesses and institutions. We can't run the modern world without software. National infrastructures and utilities are controlled by computer-based systems, and most electrical products include a computer and controlling software. Industrial manufacturing and distribution is completely computerized, as is the financial system. Entertainment, including the music industry, computer games, and film and television, is software-intensive. More than 75% of the world's population have a software-controlled mobile phone, and, by 2016, almost all of these will be Internet-enabled.

Software systems are abstract and intangible. They are not constrained by the properties of materials, nor are they governed by physical laws or by manufacturing processes. This simplifies software engineering, as there are no natural limits to the potential of software. However, because of the lack of physical constraints, software systems can quickly become extremely complex, difficult to understand, and expensive to change.

There are many different types of software system, ranging from simple embedded systems to complex, worldwide information systems. There are no universal notations, methods, or techniques for software engineering because different types of software require different approaches. Developing an organizational information system is completely different from developing a controller for a scientific instrument. Neither of these systems has much in common with a graphics-intensive computer game. All of these applications need software engineering; they do not all need the same software engineering methods and techniques.

There are still many reports of software projects going wrong and of "software failures." Software engineering is criticized as inadequate for modern software development. However, in my opinion, many of these so-called software failures are a consequence of two factors:

1. *Increasing system complexity* As new software engineering techniques help us to build larger, more complex systems, the demands change. Systems have to be built and delivered more quickly; larger, even more complex systems are required; and systems have to have new capabilities that were previously thought to be impossible. New software engineering techniques have to be developed to meet new the challenges of delivering more complex software.

2. *Failure to use software engineering methods* It is fairly easy to write computer programs without using software engineering methods and techniques. Many companies have drifted into software development as their products and services have evolved. They do not use software engineering methods in their everyday work. Consequently, their software is often more expensive and less reliable than it should be. We need better software engineering education and training to address this problem.

Software engineers can be rightly proud of their achievements. Of course, we still have problems developing complex software, but without software engineering we would not have explored space and we would not have the Internet or modern telecommunications. All forms of travel would be more dangerous and expensive. Challenges for humanity in the 21st century are climate change, fewer natural

History of software engineering

The notion of software engineering was first proposed in 1968 at a conference held to discuss what was then called the software crisis (Naur and Randell 1969). It became clear that individual approaches to program development did not scale up to large and complex software systems. These were unreliable, cost more than expected, and were delivered late.

Throughout the 1970s and 1980s, a variety of new software engineering techniques and methods were developed, such as structured programming, information hiding, and object-oriented development. Tools and standard notations were developed which are the basis of today's software engineering.

http://software-engineering-book.com/web/history/

resources, changing demographics, and an expanding world population. We will rely on software engineering to develop the systems that we need to cope with these issues.

1.1 Professional software development

Lots of people write programs. People in business write spreadsheet programs to simplify their jobs; scientists and engineers write programs to process their experimental data; hobbyists write programs for their own interest and enjoyment. However, most software development is a professional activity in which software is developed for business purposes, for inclusion in other devices, or as software products such as information systems and computer-aided design systems. The key distinctions are that professional software is intended for use by someone apart from its developer and that teams rather than individuals usually develop the software. It is maintained and changed throughout its life.

Software engineering is intended to support professional software development rather than individual programming. It includes techniques that support program specification, design, and evolution, none of which are normally relevant for personal software development. To help you to get a broad view of software engineering, I have summarized frequently asked questions about the subject in Figure 1.1.

Many people think that software is simply another word for computer programs. However, when we are talking about software engineering, software is not just the programs themselves but also all associated documentation, libraries, support websites, and configuration data that are needed to make these programs useful. A professionally developed software system is often more than a single program. A system may consist of several separate programs and configuration files that are used to set up these programs. It may include system documentation, which describes the structure of the system, user documentation, which explains how to use the system, and websites for users to download recent product information.

This is one of the important differences between professional and amateur software development. If you are writing a program for yourself, no one else will use it

Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production from initial conception to operation and maintenance.
What are the fundamental software engineering activities?	Software specification, software development, software validation and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. There are no methods and techniques that are good for everything.
What differences has the Internet made to software engineering?	Not only has the Internet led to the development of massive, highly distributed, service-based systems, it has also supported the creation of an "app" industry for mobile devices which has changed the economics of software.

Figure 1.1 Frequently asked questions about software engineering

and you don't have to worry about writing program guides, documenting the program design, and so on. However, if you are writing software that other people will use and other engineers will change, then you usually have to provide additional information as well as the code of the program.

Software engineers are concerned with developing software products, that is, software that can be sold to a customer. There are two kinds of software product:

1. *Generic products* These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them. Examples of this type of product include apps for mobile devices, software for PCs such as databases, word processors, drawing packages, and project management tools. This kind of software also includes "vertical"

applications designed for a specific market such as library information systems, accounting systems, or systems for maintaining dental records.

2. *Customized (or bespoke) software* These are systems that are commissioned by and developed for a particular customer. A software contractor designs and implements the software especially for that customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process, and air traffic control systems.

The critical distinction between these types of software is that, in generic products, the organization that develops the software controls the software specification. This means that if they run into development problems, they can rethink what is to be developed. For custom products, the specification is developed and controlled by the organization that is buying the software. The software developers must work to that specification.

However, the distinction between these system product types is becoming increasingly blurred. More and more systems are now being built with a generic product as a base, which is then adapted to suit the requirements of a customer. Enterprise Resource Planning (ERP) systems, such as systems from SAP and Oracle, are the best examples of this approach. Here, a large and complex system is adapted for a company by incorporating information about business rules and processes, reports required, and so on.

When we talk about the quality of professional software, we have to consider that the software is used and changed by people apart from its developers. Quality is therefore not just concerned with what the software does. Rather, it has to include the software's behavior while it is executing and the structure and organization of the system programs and associated documentation. This is reflected in the software's quality or non-functional attributes. Examples of these attributes are the software's response time to a user query and the understandability of the program code.

The specific set of attributes that you might expect from a software system obviously depends on its application. Therefore, an aircraft control system must be safe, an interactive game must be responsive, a telephone switching system must be reliable, and so on. These can be generalized into the set of attributes shown in Figure 1.2, which I think are the essential characteristics of a professional software system.

1.1.1 Software engineering

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use. In this definition, there are two key phrases:

1. *Engineering discipline* Engineers make things work. They apply theories, methods, and tools where these are appropriate. However, they use them selectively

Product characteristic	Description
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use.
Dependability and security	Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Software has to be secure so that malicious users cannot access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, resource utilization, etc.
Maintainability	Software should be written in such a way that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.

Figure 1.2 Essential attributes of good software

and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work within organizational and financial constraints, and they must look for solutions within these constraints.

2. *All aspects of software production* Software engineering is not just concerned with the technical processes of software development. It also includes activities such as software project management and the development of tools, methods, and theories to support software development.

Engineering is about getting results of the required quality within schedule and budget. This often involves making compromises—engineers cannot be perfectionists. People writing programs for themselves, however, can spend as much time as they wish on the program development.

In general, software engineers adopt a systematic and organized approach to their work, as this is often the most effective way to produce high-quality software. However, engineering is all about selecting the most appropriate method for a set of circumstances, so a more creative, less formal approach to development may be the right one for some kinds of software. A more flexible software process that accommodates rapid change is particularly appropriate for the development of interactive web-based systems and mobile apps, which require a blend of software and graphical design skills.

Software engineering is important for two reasons:

1. More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.

2. It is usually cheaper, in the long run, to use software engineering methods and techniques for professional software systems rather than just write programs as

a personal programming project. Failure to use software engineering method leads to higher costs for testing, quality assurance, and long-term maintenance.

The systematic approach that is used in software engineering is sometimes called a software process. A software process is a sequence of activities that leads to the production of a software product. Four fundamental activities are common to all software processes.

1. Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.
2. Software development, where the software is designed and programmed.
3. Software validation, where the software is checked to ensure that it is what the customer requires.
4. Software evolution, where the software is modified to reflect changing customer and market requirements.

Different types of systems need different development processes, as I explain in Chapter 2. For example, real-time software in an aircraft has to be completely specified before development begins. In e-commerce systems, the specification and the program are usually developed together. Consequently, these generic activities may be organized in different ways and described at different levels of detail, depending on the type of software being developed.

Software engineering is related to both computer science and systems engineering.

1. Computer science is concerned with the theories and methods that underlie computers and software systems, whereas software engineering is concerned with the practical problems of producing software. Some knowledge of computer science is essential for software engineers in the same way that some knowledge of physics is essential for electrical engineers. Computer science theory, however, is often most applicable to relatively small programs. Elegant theories of computer science are rarely relevant to large, complex problems that require a software solution.
2. System engineering is concerned with all aspects of the development and evolution of complex systems where software plays a major role. System engineering is therefore concerned with hardware development, policy and process design, and system deployment, as well as software engineering. System engineers are involved in specifying the system, defining its overall architecture, and then integrating the different parts to create the finished system.

As I discuss in the next section, there are many different types of software. There are no universal software engineering methods or techniques that may be used. However, there are four related issues that affect many different types of software:

1. *Heterogeneity* Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices. As well as running on general-purpose computers, software may also have to execute on mobile phones and tablets. You often have to integrate new software with older legacy systems written in different programming languages. The challenge here is to develop techniques for building dependable software that is flexible enough to cope with this heterogeneity.

2. *Business and social change* Businesses and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to rapidly develop new software. Many traditional software engineering techniques are time consuming, and delivery of new systems often takes longer than planned. They need to evolve so that the time required for software to deliver value to its customers is reduced.

3. *Security and trust* As software is intertwined with all aspects of our lives, it is essential that we can trust that software. This is especially true for remote software systems accessed through a web page or web service interface. We have to make sure that malicious users cannot successfully attack our software and that information security is maintained.

4. *Scale* Software has to be developed across a very wide range of scales, from very small embedded systems in portable or wearable devices through to Internet-scale, cloud-based systems that serve a global community.

To address these challenges, we will need new tools and techniques as well as innovative ways of combining and using existing software engineering methods.

1.1.2 Software engineering diversity

Software engineering is a systematic approach to the production of software that takes into account practical cost, schedule, and dependability issues, as well as the needs of software customers and producers. The specific methods, tools, and techniques used depend on the organization developing the software, the type of software, and the people involved in the development process. There are no universal software engineering methods that are suitable for all systems and all companies. Rather, a diverse set of software engineering methods and tools has evolved over the past 50 years. However, the SEMAT initiative (Jacobson et al. 2013) proposes that there can be a fundamental meta-process that can be instantiated to create different kinds of process. This is at an early stage of development and may be a basis for improving our current software engineering methods.

Perhaps the most significant factor in determining which software engineering methods and techniques are most important is the type of application being developed. There are many different types of application, including:

1. *Stand-alone applications* These are application systems that run on a personal computer or apps that run on a mobile device. They include all necessary functionality and may not need to be connected to a network. Examples of such applications are office applications on a PC, CAD programs, photo manipulation software, travel apps, productivity apps, and so on.

2. *Interactive transaction-based applications* These are applications that execute on a remote computer and that are accessed by users from their own computers, phones, or tablets. Obviously, these include web applications such as e-commerce applications where you interact with a remote system to buy goods and services. This class of application also includes business systems, where a business provides access to its systems through a web browser or special-purpose client program and cloud-based services, such as mail and photo sharing. Interactive applications often incorporate a large data store that is accessed and updated in each transaction.

3. *Embedded control systems* These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system. Examples of embedded systems include the software in a mobile (cell) phone, software that controls antilock braking in a car, and software in a microwave oven to control the cooking process.

4. *Batch processing systems* These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs. Examples of batch systems are periodic billing systems, such as phone billing systems, and salary payment systems.

5. *Entertainment systems* These are systems for personal use that are intended to entertain the user. Most of these systems are games of one kind or another, which may run on special-purpose console hardware. The quality of the user interaction offered is the most important distinguishing characteristic of entertainment systems.

6. *Systems for modeling and simulation* These are systems that are developed by scientists and engineers to model physical processes or situations, which include many separate, interacting objects. These are often computationally intensive and require high-performance parallel systems for execution.

7. *Data collection and analysis systems* Data collection systems are systems that collect data from their environment and send that data to other systems for processing. The software may have to interact with sensors and often is installed in a hostile environment such as inside an engine or in a remote location. “Big data” analysis may involve cloud-based systems carrying out statistical analysis and looking for relationships in the collected data.

8. *Systems of systems* These are systems, used in enterprises and other large organizations, that are composed of a number of other software systems. Some of these may be generic software products, such as an ERP system. Other systems in the assembly may be specially written for that environment.

Of course, the boundaries between these system types are blurred. If you develop a game for a phone, you have to take into account the same constraints (power, hardware interaction) as the developers of the phone software. Batch processing systems are often used in conjunction with web-based transaction systems. For example, in a company, travel expense claims may be submitted through a web application but processed in a batch application for monthly payment.

Each type of system requires specialized software engineering techniques because the software has different characteristics. For example, an embedded control system in an automobile is safety-critical and is burned into ROM (read-only memory) when installed in the vehicle. It is therefore very expensive to change. Such a system needs extensive verification and validation so that the chances of having to recall cars after sale to fix software problems are minimized. User interaction is minimal (or perhaps nonexistent), so there is no need to use a development process that relies on user interface prototyping.

For an interactive web-based system or app, iterative development and delivery is the best approach, with the system being composed of reusable components. However, such an approach may be impractical for a system of systems, where detailed specifications of the system interactions have to be specified in advance so that each system can be separately developed.

Nevertheless, there are software engineering fundamentals that apply to all types of software systems:

1. They should be developed using a managed and understood development process. The organization developing the software should plan the development process and have clear ideas of what will be produced and when it will be completed. Of course, the specific process that you should use depends on the type of software that you are developing.

2. Dependability and performance are important for all types of system. Software should behave as expected, without failures, and should be available for use when it is required. It should be safe in its operation and, as far as possible, should be secure against external attack. The system should perform efficiently and should not waste resources.

3. Understanding and managing the software specification and requirements (what the software should do) are important. You have to know what different customers and users of the system expect from it, and you have to manage their expectations so that a useful system can be delivered within budget and to schedule.

4. You should make effective use of existing resources. This means that, where appropriate, you should reuse software that has already been developed rather than write new software.

These fundamental notions of process, dependability, requirements, management, and reuse are important themes of this book. Different methods reflect them in different ways, but they underlie all professional software development.

These fundamentals are independent of the program language used for software development. I don't cover specific programming techniques in this book because these vary dramatically from one type of system to another. For example, a dynamic language, such as Ruby, is the right type of language for interactive system development but is inappropriate for embedded systems engineering.

1.1.3 Internet software engineering

The development of the Internet and the World Wide Web has had a profound effect on all of our lives. Initially, the web was primarily a universally accessible information store, and it had little effect on software systems. These systems ran on local computers and were only accessible from within an organization. Around 2000, the web started to evolve, and more and more functionality was added to browsers. This meant that web-based systems could be developed where, instead of a special-purpose user interface, these systems could be accessed using a web browser. This led to the development of a vast range of new system products that delivered innovative services, accessed over the web. These are often funded by adverts that are displayed on the user's screen and do not involve direct payment from users.

As well as these system products, the development of web browsers that could run small programs and do some local processing led to an evolution in business and organizational software. Instead of writing software and deploying it on users' PCs, the software was deployed on a web server. This made it much cheaper to change and upgrade the software, as there was no need to install the software on every PC. It also reduced costs, as user interface development is particularly expensive. Wherever it has been possible to do so, businesses have moved to web-based interaction with company software systems.

The notion of software as a service (Chapter 17) was proposed early in the 21st century This has now become the standard approach to the delivery of web-based system products such as Google Apps, Microsoft Office 365, and Adobe Creative Suite. More and more software runs on remote "clouds" instead of local servers and is accessed over the Internet. A computing cloud is a huge number of linked computer systems that is shared by many users. Users do not buy software but pay according to how much the software is used or are given free access in return for watching adverts that are displayed on their screen. If you use services such as web-based mail, storage, or video, you are using a cloud-based system.

The advent of the web has led to a dramatic change in the way that business software is organized. Before the web, business applications were mostly monolithic, single programs running on single computers or computer clusters. Communications were local, within an organization. Now, software is highly distributed, sometimes across the world. Business applications are not programmed from scratch but involve extensive reuse of components and programs.

This change in software organization has had a major effect on software engineering for web-based systems. For example:

1. Software reuse has become the dominant approach for constructing web-based systems. When building these systems, you think about how you can assemble them from preexisting software components and systems, often bundled together in a framework.
2. It is now generally recognized that it is impractical to specify all the requirements for such systems in advance. Web-based systems are always developed and delivered incrementally.
3. Software may be implemented using service-oriented software engineering, where the software components are stand-alone web services. I discuss this approach to software engineering in Chapter 18.
4. Interface development technology such as AJAX (Holdener 2008) and HTML5 (Freeman 2011) have emerged that support the creation of rich interfaces within a web browser.

The fundamental ideas of software engineering, discussed in the previous section, apply to web-based software, as they do to other types of software. Web-based systems are getting larger and larger, so software engineering techniques that deal with scale and complexity are relevant for these systems.

1.2 Software engineering ethics

Like other engineering disciplines, software engineering is carried out within a social and legal framework that limits the freedom of people working in that area. As a software engineer, you must accept that your job involves wider responsibilities than simply the application of technical skills. You must also behave in an ethical and morally responsible way if you are to be respected as a professional engineer.

It goes without saying that you should uphold normal standards of honesty and integrity. You should not use your skills and abilities to behave in a dishonest way or in a way that will bring disrepute to the software engineering profession. However, there are areas where standards of acceptable behavior are not bound by laws but by the more tenuous notion of professional responsibility. Some of these are:

1. *Confidentiality* You should normally respect the confidentiality of your employers or clients regardless of whether or not a formal confidentiality agreement has been signed.
2. *Competence* You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.
3. *Intellectual property rights* You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.

Software Engineering Code of Ethics and Professional Practice

ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing, and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety, and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. PUBLIC – Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER – Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT – Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT – Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT – Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION – Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES – Software engineers shall be fair to and supportive of their colleagues.
8. SELF – Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Figure 1.3 The ACM/IEEE Code of Ethics (ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, short version. http://www.acm.org/about/se-code)

(© 1999 by the ACM, Inc. and the IEEE, Inc.)

4. *Computer misuse* You should not use your technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine) to extremely serious (dissemination of viruses or other malware).

Professional societies and institutions have an important role to play in setting ethical standards. Organizations such as the ACM, the IEEE (Institute of Electrical and Electronic Engineers), and the British Computer Society publish a code of professional conduct or code of ethics. Members of these organizations undertake to follow that code when they sign up for membership. These codes of conduct are generally concerned with fundamental ethical behavior.

Professional associations, notably the ACM and the IEEE, have cooperated to produce a joint code of ethics and professional practice. This code exists in both a short form, shown in Figure 1.3, and a longer form (Gotterbarn, Miller, and Rogerson 1999) that adds detail and substance to the shorter version. The rationale behind this code is summarized in the first two paragraphs of the longer form:

Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance and testing of software systems. Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession. In accordance with that commitment, software engineers shall adhere to the following Code of Ethics and Professional Practice[†].

The Code contains eight Principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession. The Principles identify the ethically responsible relationships in which individuals, groups, and organizations participate and the primary obligations within these relationships. The Clauses of each Principle are illustrations of some of the obligations included in these relationships. These obligations are founded in the software engineer's humanity, in special care owed to people affected by the work of software engineers, and the unique elements of the practice of software engineering. The Code prescribes these as obligations of anyone claiming to be or aspiring to be a software engineer[†].

In any situation where different people have different views and objectives, you are likely to be faced with ethical dilemmas. For example, if you disagree, in principle, with the policies of more senior management in the company, how should you react? Clearly, this depends on the people involved and the nature of the disagreement. Is it best to argue a case for your position from within the organization or to resign in principle? If you feel that there are problems with a software project, when do you reveal these problems to management? If you discuss these while they are just a suspicion, you may be overreacting to a situation; if you leave it too long, it may be impossible to resolve the difficulties.

We all face such ethical dilemmas in our professional lives, and, fortunately, in most cases they are either relatively minor or can be resolved without too much difficulty. Where they cannot be resolved, the engineer is faced with, perhaps, another problem. The principled action may be to resign from their job, but this may well affect others such as their partner or their children.

A difficult situation for professional engineers arises when their employer acts in an unethical way. Say a company is responsible for developing a safety-critical system and, because of time pressure, falsifies the safety validation records. Is the engineer's responsibility to maintain confidentiality or to alert the customer or publicize, in some way, that the delivered system may be unsafe?

[†]ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, short version Preamble. http://www.acm.org/about/se-code Copyright © 1999 by the Association for Computing Machinery, Inc. and the Institute for Electrical and Electronics Engineers, Inc.

The problem here is that there are no absolutes when it comes to safety. Although the system may not have been validated according to predefined criteria, these criteria may be too strict. The system may actually operate safely throughout its lifetime. It is also the case that, even when properly validated, the system may fail and cause an accident. Early disclosure of problems may result in damage to the employer and other employees; failure to disclose problems may result in damage to others.

You must make up your own mind in these matters. The appropriate ethical position here depends on the views of the people involved. The potential for damage, the extent of the damage, and the people affected by the damage should influence the decision. If the situation is very dangerous, it may be justified to publicize it using the national press or social media. However, you should always try to resolve the situation while respecting the rights of your employer.

Another ethical issue is participation in the development of military and nuclear systems. Some people feel strongly about these issues and do not wish to participate in any systems development associated with defense systems. Others will work on military systems but not on weapons systems. Yet others feel that national security is an overriding principle and have no ethical objections to working on weapons systems.

In this situation, it is important that both employers and employees should make their views known to each other in advance. Where an organization is involved in military or nuclear work, it should be able to specify that employees must be willing to accept any work assignment. Equally, if an employee is taken on and makes clear that he or she does not wish to work on such systems, employers should not exert pressure to do so at some later date.

The general area of ethics and professional responsibility is increasingly important as software-intensive systems pervade every aspect of work and everyday life. It can be considered from a philosophical standpoint where the basic principles of ethics are considered and software engineering ethics are discussed with reference to these basic principles. This is the approach taken by Laudon (Laudon 1995) and Johnson (Johnson 2001). More recent texts such as that by Tavani (Tavani 2013) introduce the notion of cyberethics and cover both the philosophical background and practical and legal issues. They include ethical issues for technology users as well as developers.

I find that a philosophical approach is too abstract and difficult to relate to everyday experience so I prefer the more concrete approach embodied in professional codes of conduct (Bott 2005; Duquenoy 2007). I think that ethics are best discussed in a software engineering context and not as a subject in its own right. Therefore, I do not discuss software engineering ethics in an abstract way but include examples in the exercises that can be the starting point for a group discussion.

1.3 Case studies

To illustrate software engineering concepts, I use examples from four different types of system. I have deliberately not used a single case study, as one of the key messages in this book is that software engineering practice depends on the type of systems

being produced. I therefore choose an appropriate example when discussing concepts such as safety and dependability, system modeling, reuse, etc.

The system types that I use as case studies are:

1. *An embedded system* This is a system where the software controls some hardware device and is embedded in that device. Issues in embedded systems typically include physical size, responsiveness, and power management, etc. The example of an embedded system that I use is a software system to control an insulin pump for people who have diabetes.
2. *An information system* The primary purpose of this type of system is to manage and provide access to a database of information. Issues in information systems include security, usability, privacy, and maintaining data integrity. The example of an information system used is a medical records system.
3. *A sensor-based data collection system* This is a system whose primary purposes are to collect data from a set of sensors and to process that data in some way. The key requirements of such systems are reliability, even in hostile environmental conditions, and maintainability. The example of a data collection system that I use is a wilderness weather station.
4. *A support environment.* This is an integrated collection of software tools that are used to support some kind of activity. Programming environments, such as Eclipse (Vogel 2012) will be the most familiar type of environment for readers of this book. I describe an example here of a digital learning environment that is used to support students' learning in schools.

I introduce each of these systems in this chapter; more information about each of them is available on the website (software-engineering-book.com).

1.3.1 An insulin pump control system

An insulin pump is a medical system that simulates the operation of the pancreas (an internal organ). The software controlling this system is an embedded system that collects information from a sensor and controls a pump that delivers a controlled dose of insulin to a user.

People who suffer from diabetes use the system. Diabetes is a relatively common condition in which the human pancreas is unable to produce sufficient quantities of a hormone called insulin. Insulin metabolizes glucose (sugar) in the blood. The conventional treatment of diabetes involves regular injections of genetically engineered insulin. Diabetics measure their blood sugar levels periodically using an external meter and then estimate the dose of insulin they should inject.

The problem is that the level of insulin required does not just depend on the blood glucose level but also on the time of the last insulin injection. Irregular checking can lead to very low levels of blood glucose (if there is too much insulin) or very high levels of blood sugar (if there is too little insulin). Low blood glucose is, in the short term, a more serious condition as it can result in temporary brain malfunctioning and,

Figure 1.4 Insulin pump hardware architecture

Figure 1.5 Activity model of the insulin pump

ultimately, unconsciousness and death. In the long term, however, continual high levels of blood glucose can lead to eye damage, kidney damage, and heart problems.

Advances in developing miniaturized sensors have meant that it is now possible to develop automated insulin delivery systems. These systems monitor blood sugar levels and deliver an appropriate dose of insulin when required. Insulin delivery systems like this one are now available and are used by patients who find it difficult to control their insulin levels. In future, it may be possible for diabetics to have such systems permanently attached to their bodies.

A software-controlled insulin delivery system uses a microsensor embedded in the patient to measure some blood parameter that is proportional to the sugar level. This is then sent to the pump controller. This controller computes the sugar level and the amount of insulin that is needed. It then sends signals to a miniaturized pump to deliver the insulin via a permanently attached needle.

Figure 1.4 shows the hardware components and organization of the insulin pump. To understand the examples in this book, all you need to know is that the blood sensor measures the electrical conductivity of the blood under different conditions and that these values can be related to the blood sugar level. The insulin pump delivers one unit of insulin in response to a single pulse from a controller. Therefore, to deliver 10 units of insulin, the controller sends 10 pulses to the pump. Figure 1.5 is a Unified Modeling

Figure 1.6 The organization of the Mentcare system

Language (UML) activity model that illustrates how the software transforms an input blood sugar level to a sequence of commands that drive the insulin pump.

Clearly, this is a safety-critical system. If the pump fails to operate or does not operate correctly, then the user's health may be damaged or they may fall into a coma because their blood sugar levels are too high or too low. This system must therefore meet two essential high-level requirements:

1. The system shall be available to deliver insulin when required.
2. The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.

The system must therefore be designed and implemented to ensure that it always meets these requirements. More detailed requirements and discussions of how to ensure that the system is safe are discussed in later chapters.

1.3.2 A patient information system for mental health care

A patient information system to support mental health care (the Mentcare system) is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received. Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems. To make it easier for patients to attend, these clinics are not just run in hospitals. They may also be held in local medical practices or community centers.

The Mentcare system (Figure 1.6) is a patient information system that is intended for use in clinics. It makes use of a centralized database of patient information but

has also been designed to run on a laptop, so that it may be accessed and used from sites that do not have secure network connectivity. When the local systems have secure network access, they use patient information in the database, but they can download and use local copies of patient records when they are disconnected. The system is not a complete medical records system and so does not maintain information about other medical conditions. However, it may interact and exchange data with other clinical information systems.

This system has two purposes:

1. To generate management information that allows health service managers to assess performance against local and government targets.
2. To provide medical staff with timely information to support the treatment of patients.

Patients who suffer from mental health problems are sometimes irrational and disorganized so may miss appointments, deliberately or accidentally lose prescriptions and medication, forget instructions and make unreasonable demands on medical staff. They may drop in on clinics unexpectedly. In a minority of cases, they may be a danger to themselves or to other people. They may regularly change address or may be homeless on a long-term or short-term basis. Where patients are dangerous, they may need to be "sectioned"—that is, confined to a secure hospital for treatment and observation.

Users of the system include clinical staff such as doctors, nurses, and health visitors (nurses who visit people at home to check on their treatment). Nonmedical users include receptionists who make appointments, medical records staff who maintain the records system, and administrative staff who generate reports.

The system is used to record information about patients (name, address, age, next of kin, etc.), consultations (date, doctor seen, subjective impressions of the patient, etc.), conditions, and treatments. Reports are generated at regular intervals for medical staff and health authority managers. Typically, reports for medical staff focus on information about individual patients, whereas management reports are anonymized and are concerned with conditions, costs of treatment, etc.

The key features of the system are:

1. *Individual care management* Clinicians can create records for patients, edit the information in the system, view patient history, and so on. The system supports data summaries so that doctors who have not previously met a patient can quickly learn about the key problems and treatments that have been prescribed.
2. *Patient monitoring* The system regularly monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected. Therefore, if a patient has not seen a doctor for some time, a warning may be issued. One of the most important elements of the monitoring system is to keep track of patients who have been sectioned and to ensure that the legally required checks are carried out at the right time.

3. *Administrative reporting* The system generates monthly management reports showing the number of patients treated at each clinic, the number of patients who have entered and left the care system, the number of patients sectioned, the drugs prescribed and their costs, etc.

Two different laws affect the system: laws on data protection that govern the confidentiality of personal information and mental health laws that govern the compulsory detention of patients deemed to be a danger to themselves or others. Mental health is unique in this respect as it is the only medical speciality that can recommend the detention of patients against their will. This is subject to strict legislative safeguards. One aim of the Mentcare system is to ensure that staff always act in accordance with the law and that their decisions are recorded for judicial review if necessary.

As in all medical systems, privacy is a critical system requirement. It is essential that patient information is confidential and is never disclosed to anyone apart from authorized medical staff and the patient themselves. The Mentcare system is also a safety-critical system. Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients.

The overall design of the system has to take into account privacy and safety requirements. The system must be available when needed; otherwise safety may be compromised, and it may be impossible to prescribe the correct medication to patients. There is a potential conflict here. Privacy is easiest to maintain when there is only a single copy of the system data. However, to ensure availability in the event of server failure or when disconnected from a network, multiple copies of the data should be maintained. I discuss the trade-offs between these requirements in later chapters.

1.3.3 A wilderness weather station

To help monitor climate change and to improve the accuracy of weather forecasts in remote areas, the government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas. These weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed and wind direction.

Wilderness weather stations are part of a larger system (Figure 1.7), which is a weather information system that collects data from weather stations and makes it available to other systems for processing. The systems in Figure 1.7 are:

1. *The weather station system* This system is responsible for collecting weather data, carrying out some initial data processing, and transmitting it to the data management system.

2. *The data management and archiving system* This system collects the data from all of the wilderness weather stations, carries out data processing and analysis, and archives the data in a form that can be retrieved by other systems, such as weather forecasting systems.

Figure 1.7 The weather station's environment

3. *The station maintenance system* This system can communicate by satellite with all wilderness weather stations to monitor the health of these systems and provide reports of problems. It can update the embedded software in these systems. In the event of system problems, this system can also be used to remotely control the weather station.

In Figure 1.7, I have used the UML package symbol to indicate that each system is a collection of components and the separate systems are identified using the UML stereotype «system». The associations between the packages indicate there is an exchange of information but, at this stage, there is no need to define them in any more detail.

The weather stations include instruments that measure weather parameters such as wind speed and direction, ground and air temperatures, barometric pressure, and rainfall over a 24-hour period. Each of these instruments is controlled by a software system that takes parameter readings periodically and manages the data collected from the instruments.

The weather station system operates by collecting weather observations at frequent intervals; for example, temperatures are measured every minute. However, because the bandwidth to the satellite is relatively narrow, the weather station carries out some local processing and aggregation of the data. It then transmits this aggregated data when requested by the data collection system. If it is impossible to make a connection, then the weather station maintains the data locally until communication can be resumed.

Each weather station is battery-powered and must be entirely self-contained; there are no external power or network cables. All communications are through a relatively slow satellite link, and the weather station must include some mechanism (solar or wind power) to charge its batteries. As they are deployed in wilderness areas, they are exposed to severe environmental conditions and may be damaged by animals. The station software is therefore not just concerned with data collection. It must also:

1. Monitor the instruments, power. and communication hardware and report faults to the management system.

2. Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such as high wind.

3. Allow for dynamic reconfiguration where parts of the software are replaced with new versions and where backup instruments are switched into the system in the event of system failure.

Because weather stations have to be self-contained and unattended, this means that the software installed is complex, even though the data collection functionality is fairly simple.

1.3.4 A digital learning environment for schools

Many teachers argue that using interactive software systems to support education can lead to both improved learner motivation and a deeper level of knowledge and understanding in students. However, there is no general agreement on the 'best' strategy for computer-supported learning, and teachers in practice use a range of different interactive, web-based tools to support learning. The tools used depend on the ages of the learners, their cultural background, their experience with computers, equipment available, and the preferences of the teachers involved.

A digital learning environment is a framework in which a set of general-purpose and specially designed tools for learning may be embedded, plus a set of applications that are geared to the needs of the learners using the system. The framework provides general services such as an authentication service, synchronous and asynchronous communication services, and a storage service.

The tools included in each version of the environment are chosen by teachers and learners to suit their specific needs. These can be general applications such as spreadsheets, learning management applications such as a Virtual Learning Environment (VLE) to manage homework submission and assessment, games, and simulations. They may also include specific content, such as content about the American Civil War and applications to view and annotate that content.

Figure 1.8 is a high-level architectural model of a digital learning environment (iLearn) that was designed for use in schools for students from 3 to 18 years of age. The approach adopted is that this is a distributed system in which all components of the environment are services that can be accessed from anywhere on the Internet. There is no requirement that all of the learning tools are gathered together in one place.

The system is a service-oriented system with all system components considered to be a replaceable service. There are three types of service in the system:

1. *Utility services* that provide basic application-independent functionality and that may be used by other services in the system. Utility services are usually developed or adapted specifically for this system.
2. *Application services* that provide specific applications such as email, conferencing, photo sharing, etc., and access to specific educational content such as scientific films or historical resources. Application services are external services that are either specifically purchased for the system or are available freely over the Internet.

Figure 1.8 The architecture of a digital learning environment (iLearn)

3. *Configuration services* that are used to adapt the environment with a specific set of application services and to define how services are shared between students, teachers, and their parents.

The environment has been designed so that services can be replaced as new services become available and to provide different versions of the system that are suited for the age of the users. This means that the system has to support two levels of service integration:

1. *Integrated services* are services that offer an API (application programming interface) and that can be accessed by other services through that API. Direct service-to-service communication is therefore possible. An authentication service is an example of an integrated service. Rather than use their own authentication mechanisms, an authentication service may be called on by other services to authenticate users. If users are already authenticated, then the authentication service may pass authentication information directly to another service, via an API, with no need for users to reauthenticate themselves.
2. *Independent services* are services that are simply accessed through a browser interface and that operate independently of other services. Information can only be shared with other services through explicit user actions such as copy and paste; reauthentication may be required for each independent service.

If an independent service becomes widely used, the development team may then integrate that service so that it becomes an integrated and supported service.

KEY POINTS

- Software engineering is an engineering discipline that is concerned with all aspects of software production.
- Software is not just a program or programs but also includes all electronic documentation that is needed by system users, quality assurance staff, and developers. Essential software product attributes are maintainability, dependability and security, efficiency, and acceptability.
- The software process includes all of the activities involved in software development. The high-level activities of specification, development, validation, and evolution are part of all software processes.
- There are many different types of system, and each requires appropriate software engineering tools and techniques for their development. Few, if any, specific design and implementation techniques are applicable to all kinds of system.
- The fundamental ideas of software engineering are applicable to all types of software system. These fundamentals include managed software processes, software dependability and security, requirements engineering, and software reuse.
- Software engineers have responsibilities to the engineering profession and society. They should not simply be concerned with technical issues but should be aware of the ethical issues that affect their work.
- Professional societies publish codes of conduct that embed ethical and professional standards. These set out the standards of behavior expected of their members.

FURTHER READING

"Software Engineering Code of Ethics Is Approved." An article that discusses the background to the development of the ACM/IEEE Code of Ethics and that includes both the short and long form of the code. (*Comm. ACM*, D. Gotterbarn, K. Miller, and S. Rogerson, October 1999). http://dx.doi.org/10.1109/MC.1999.796142

"A View of 20th and 21st Century Software Engineering." A backward and forward look at software engineering from one of the first and most distinguished software engineers. Barry Boehm identifies timeless software engineering principles but also suggests that some commonly used practices are obsolete. (B. Boehm, *Proc. 28th Software Engineering Conf.*, Shanghai. 2006). http://dx.doi.org/10.1145/1134285.1134288

"Software Engineering Ethics." Special issue of *IEEE Computer*, with several papers on the topic (*IEEE Computer*, 42 (6), June 2009).

Ethics for the Information Age. This is a wide-ranging book that covers all aspects of information technology (IT) ethics, not simply ethics for software engineers. I think this is the right approach as you really need to understand software engineering ethics within a wider ethical framework (M. J. Quinn, 2013, Addison-Wesley).

The Essence of Software Engineering: Applying the SEMAT kernel. This book discusses the idea of a universal framework that can underlie all software engineering methods. It can be adapted and used for all types of systems and organizations. I am personally skeptical about whether or not a universal approach is realistic in practice, but the book has some interesting ideas that are worth exploring. (I. Jacobsen, P-W Ng, P. E. McMahon, I. Spence, and S. Lidman, 2013, Addison-Wesley)

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-engineering/

Links to case study descriptions:

http://software-engineering-book.com/case-studies/

EXERCISES

1.1. Explain why professional software that is developed for a customer is not simply the programs that have been developed and delivered.

1.2. What is the most important difference between generic software product development and custom software development? What might this mean in practice for users of generic software products?

1.3. Briefly discuss why it is usually cheaper in the long run to use software engineering methods and techniques for software systems.

1.4. Software engineering is not only concerned with issues like system heterogeneity, business and social change, trust, and security, but also with ethical issues affecting the domain. Give some examples of ethical issues that have an impact on the software engineering domain.

1.5. Based on your own knowledge of some of the application types discussed in Section 1.1.2, explain, with examples, why different application types require specialized software engineering techniques to support their design and development.

1.6. Explain why the fundamental software engineering principles of process, dependability, requirements management, and reuse are relevant to all types of software system.

1.7. Explain how electronic connectivity between various development teams can support software engineering activities.

1.8. Noncertified individuals are still allowed to practice software engineering. Discuss some of the possible drawbacks of this.

1.9. For each of the clauses in the ACM/IEEE Code of Ethics shown in Figure 1.4, propose an appropriate example that illustrates that clause.

1.10. The "Drone Revolution" is currently being debated and discussed all over the world. Drones are unmanned flying machines that are built and equipped with various kinds of software systems that allow them to see, hear, and act. Discuss some of the societal challenges of building such kinds of systems.

REFERENCES

Bott, F. 2005. *Professional Issues in Information Technology*. Swindon, UK: British Computer Society.

Duquenoy, P. 2007. *Ethical, Legal and Professional Issues in Computing*. London: Thomson Learning.

Freeman, A. 2011. *The Definitive Guide to HTML5*. New York: Apress.

Gotterbarn, D., K. Miller, and S. Rogerson. 1999. "Software Engineering Code of Ethics Is Approved." *Comm. ACM* 42 (10): 102–107. doi:10.1109/MC.1999.796142.

Holdener, A. T. 2008. *Ajax: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates.

Jacobson, I., P-W. Ng, P. E. McMahon, I. Spence, and S. Lidman. 2013. *The Essence of Software Engineering*. Boston: Addison-Wesley.

Johnson, D. G. 2001. *Computer Ethics*. Englewood Cliffs, NJ: Prentice-Hall.

Laudon, K. 1995. "Ethical Concepts and Information Technology." *Comm. ACM* 38 (12): 33–39. doi:10.1145/219663.219677.

Naur, P., and Randell, B. 1969. Software Engineering: Report on a conference sponsored by the NATO Science Committee. Brussels. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.pdf

Tavani, H. T. 2013. *Ethics and Technology: Controversies, Questions, and Strategies for Ethical Computing, 4th ed.* New York: John Wiley & Sons.

Vogel, L. 2012. *Eclipse 4 Application Development: The Complete Guide to Eclipse 4 RCP Development*. Sebastopol, CA: O'Reilly & Associates.

2

Software processes

Objectives

The objective of this chapter is to introduce you to the idea of a software process—a coherent set of activities for software production. When you have read this chapter, you will:

- understand the concepts of software processes and software process models;
- have been introduced to three general software process models and when they might be used;
- know about the fundamental process activities of software requirements engineering, software development, testing, and evolution;
- understand why processes should be organized to cope with changes in the software requirements and design;
- understand the notion of software process improvement and the factors that affect software process quality.

Contents

2.1 Software process models
2.2 Process activities
2.3 Coping with change
2.4 Process improvement

A software process is a set of related activities that leads to the production of a software system. As I discussed in Chapter 1, there are many different types of software systems, and there is no universal software engineering method that is applicable to all of them. Consequently, there is no universally applicable software process. The process used in different companies depends on the type of software being developed, the requirements of the software customer, and the skills of the people writing the software.

However, although there are many different software processes, they all must include, in some form, the four fundamental software engineering activities that I introduced in Chapter 1:

1. *Software specification* The functionality of the software and constraints on its operation must be defined.
2. *Software development* The software to meet the specification must be produced.
3. *Software validation* The software must be validated to ensure that it does what the customer wants.
4. *Software evolution* The software must evolve to meet changing customer needs.

These activities are complex activities in themselves, and they include subactivities such as requirements validation, architectural design, and unit testing. Processes also include other activities, such as software configuration management and project planning that support production activities.

When we describe and discuss processes, we usually talk about the activities in these processes, such as specifying a data model and designing a user interface, and the ordering of these activities. We can all relate to what people do to develop software. However, when describing processes, it is also important to describe who is involved, what is produced, and conditions that influence the sequence of activities:

1. Products or deliverables are the outcomes of a process activity. For example, the outcome of the activity of architectural design may be a model of the software architecture.
2. Roles reflect the responsibilities of the people involved in the process. Examples of roles are project manager, configuration manager, and programmer.
3. Pre- and postconditions are conditions that must hold before and after a process activity has been enacted or a product produced. For example, before architectural design begins, a precondition may be that the consumer has approved all requirements; after this activity is finished, a postcondition might be that the UML models describing the architecture have been reviewed.

Software processes are complex and, like all intellectual and creative processes, rely on people making decisions and judgments. As there is no universal process that is right for all kinds of software, most software companies have developed their own

development processes. Processes have evolved to take advantage of the capabilities of the software developers in an organization and the characteristics of the systems that are being developed. For safety-critical systems, a very structured development process is required where detailed records are maintained. For business systems, with rapidly changing requirements, a more flexible, agile process is likely to be better.

As I discussed in Chapter 1, professional Professional software development is a managed activity, so planning is an inherent part of all processes. Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan. In agile processes, which I discuss in Chapter 3, planning is incremental and continual as the software is developed. It is therefore easier to change the process to reflect changing customer or product requirements. As Boehm and Turner (Boehm and Turner 2004) explain, each approach is suitable for different types of software. Generally, for large systems, you need to find a balance between plan-driven and agile processes.

Although there is no universal software process, there is scope for process improvement in many organizations. Processes may include outdated techniques or may not take advantage of the best practice in industrial software engineering. Indeed, many organizations still do not take advantage of software engineering methods in their software development. They can improve their process by introducing techniques such as UML modeling and test-driven development. I discuss software process improvement briefly later in thischapter text and in more detail in web Chapter 26.

2.1 Software process models

As I explained in Chapter 1, a software process model (sometimes called a Software Development Life Cycle or SDLC model) is a simplified representation of a software process. Each process model represents a process from a particular perspective and thus only provides partial information about that process. For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities. In this section, I introduce a number of very general process models (sometimes called *process paradigms*) and present these from an architectural perspective. That is, we see the framework of the process but not the details of process activities.

These generic models are high-level, abstract descriptions of software processes that can be used to explain different approaches to software development. You can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes.

The general process models that I cover here are:

1. *The waterfall model* This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, and testing.

The Rational Unified Process

The Rational Unified Process (RUP) brings together elements of all of the general process models discussed here and supports prototyping and incremental delivery of software (Krutchen 2003). The RUP is normally described from three perspectives: a dynamic perspective that shows the phases of the model in time, a static perspective that shows process activities, and a practice perspective that suggests good practices to be used in the process. Phases of the RUP are inception, where a business case for the system is established; elaboration, where requirements and architecture are developed; construction where the software is implemented; and transition, where the system is deployed.

http://software-engineering-book.com/web/rup/

2. *Incremental development* This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.
3. *Integration and configuration* This approach relies on the availability of reusable components or systems. The system development process focuses on configuring these components for use in a new setting and integrating them into a system.

As I have said, there is no universal process model that is right for all kinds of software development. The right process depends on the customer and regulatory requirements, the environment where the software will be used, and the type of software being developed. For example, safety-critical software is usually developed using a waterfall process as lots of analysis and documentation is required before implementation begins. Software products are now always developed using an incremental process model. Business systems are increasingly being developed by configuring existing systems and integrating these to create a new system with the functionality that is required.

The majority of practical software processes are based on a general model but often incorporate features of other models. This is particularly true for large systems engineering. For large systems, it makes sense to combine some of the best features of all of the general processes. You need to have information about the essential system requirements to design a software architecture to support these requirements. You cannot develop this incrementally. Subsystems within a larger system may be developed using different approaches. Parts of the system that are well understood can be specified and developed using a waterfall-based process or may be bought in as off-the-shelf systems for configuration. Other parts of the system, which are difficult to specify in advance, should always be developed using an incremental approach. In both cases, software components are likely to be reused.

Various attempts have been made to develop "universal" process models that draw on all of these general models. One of the best known of these universal models is the Rational Unified Process (RUP) (Krutchen 2003), which was developed by Rational, a U.S. software engineering company. The RUP is a flexible model that

Figure 2.1 The waterfall model

can be instantiated in different ways to create processes that resemble any of the general process models discussed here. The RUP has been adopted by some large software companies (notably IBM), but it has not gained widespread acceptance.

2.1.1 The waterfall model

The first published model of the software development process was derived from engineering process models used in large military systems engineering (Royce 1970). It presents the software development process as a number of stages, as shown in Figure 2.1. Because of the cascade from one phase to another, this model is known as the waterfall model or software life cycle. The waterfall model is an example of a plan-driven process. In principle at least, you plan and schedule all of the process activities before starting software development.

The stages of the waterfall model directly reflect the fundamental software development activities:

1. *Requirements analysis and definition* The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.

2. *System and software design* The systems design process allocates the requirements to either hardware or software systems. It establishes an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.

3. *Implementation and unit testing* During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

Boehm's spiral process model

Barry Boehm, one of the pioneers in software engineering, proposed an incremental process model that was risk-driven. The process is represented as a spiral rather than a sequence of activities (Boehm 1988).

Each loop in the spiral represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design, and so on. The spiral model combines change avoidance with change tolerance. It assumes that changes are a result of project risks and includes explicit risk management activities to reduce these risks.

http://software-engineering-book.com/web/spiral-model/

4. *Integration and system testing* The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

5. *Operation and maintenance* Normally, this is the longest life-cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors that were not discovered in earlier stages of the life cycle, improving the implementation of system units, and enhancing the system's services as new requirements are discovered.

In principle, the result of each phase in the waterfall model is one or more documents that are approved ("signed off"). The following phase should not start until the previous phase has finished. For hardware development, where high manufacturing costs are involved, this makes sense. However, for software development, these stages overlap and feed information to each other. During design, problems with requirements are identified; during coding design problems are found, and so on. The software process, in practice, is never a simple linear model but involves feedback from one phase to another.

As new information emerges in a process stage, the documents produced at previous stages should be modified to reflect the required system changes. For example, if it is discovered that a requirement is too expensive to implement, the requirements document should be changed to remove that requirement. However, this requires customer approval and delays the overall development process.

As a result, both customers and developers may prematurely freeze the software specification so that no further changes are made to it. Unfortunately, this means that problems are left for later resolution, ignored, or programmed around. Premature freezing of requirements may mean that the system won't do what the user wants. It may also lead to badly structured systems as design problems are circumvented by implementation tricks.

During the final life-cycle phase (operation and maintenance) the software is put into use. Errors and omissions in the original software requirements are discovered.

Program and design errors emerge, and the need for new functionality is identified. The system must therefore evolve to remain useful. Making these changes (software maintenance) may involve repeating previous process stages.

In reality, software has to be flexible and accommodate change as it is being developed. The need for early commitment and system rework when changes are made means that the waterfall model is only appropriate for some types of system:

1. Embedded systems where the software has to interface with hardware systems. Because of the inflexibility of hardware, it is not usually possible to delay decisions on the software's functionality until it is being implemented.
2. Critical systems where there is a need for extensive safety and security analysis of the software specification and design. In these systems, the specification and design documents must be complete so that this analysis is possible. Safety-related problems in the specification and design are usually very expensive to correct at the implementation stage.
3. Large software systems that are part of broader engineering systems developed by several partner companies. The hardware in the systems may be developed using a similar model, and companies find it easier to use a common model for hardware and software. Furthermore, where several companies are involved, complete specifications may be needed to allow for the independent development of different subsystems.

The waterfall model is not the right process model in situations where informal team communication is possible and software requirements change quickly. Iterative development and agile methods are better for these systems.

An important variant of the waterfall model is formal system development, where a mathematical model of a system specification is created. This model is then refined, using mathematical transformations that preserve its consistency, into executable code. Formal development processes, such as that based on the B method (Abrial 2005, 2010), are mostly used in the development of software systems that have stringent safety, reliability, or security requirements. The formal approach simplifies the production of a safety or security case. This demonstrates to customers or regulators that the system actually meets its safety or security requirements. However, because of the high costs of developing a formal specification, this development model is rarely used except for critical systems engineering.

2.1.2 Incremental development

Incremental development is based on the idea of developing an initial implementation, getting feedback from users and others, and evolving the software through several versions until the required system has been developed (Figure 2.2). Specification, development, and validation activities are interleaved rather than separate, with rapid feedback across activities.

Figure 2.2 Incremental development

Incremental development in some form is now the most common approach for the development of application systems and software products. This approach can be either plan-driven, agile or, more usually, a mixture of these approaches. In a plan-driven approach, the system increments are identified in advance; if an agile approach is adopted, the early increments are identified, but the development of later increments depends on progress and customer priorities.

Incremental software development, which is a fundamental part of agile development methods, is better than a waterfall approach for systems whose requirements are likely to change during the development process. This is the case for most business systems and software products. Incremental development reflects the way that we solve problems. We rarely work out a complete problem solution in advance but move toward a solution in a series of steps, backtracking when we realize that we have made a mistake. By developing the software incrementally, it is cheaper and easier to make changes in the software as it is being developed.

Each increment or version of the system incorporates some of the functionality that is needed by the customer. Generally, the early increments of the system include the most important or most urgently required functionality. This means that the customer or user can evaluate the system at a relatively early stage in the development to see if it delivers what is required. If not, then only the current increment has to be changed and, possibly, new functionality defined for later increments.

Incremental development has three major advantages over the waterfall model:

1. The cost of implementing requirements changes is reduced. The amount of analysis and documentation that has to be redone is significantly less than is required with the waterfall model.

2. It is easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how

Problems with incremental development

Although incremental development has many advantages, it is not problem free. The primary cause of the difficulty is the fact that large organizations have bureaucratic procedures that have evolved over time and there may be a mismatch between these procedures and a more informal iterative or agile process.

Sometimes these procedures are there for good reasons. For example, there may be procedures to ensure that the software meets properly implements external regulations (e.g., in the United States, the Sarbanes Oxley accounting regulations). Changing these procedures may not be possible, so process conflicts may be unavoidable.

http://software-engineering-book.com/web/incremental-development/

much has been implemented. Customers find it difficult to judge progress from software design documents.

3. Early delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

From a management perspective, the incremental approach has two problems:

1. The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost effective to produce documents that reflect every version of the system.
2. System structure tends to degrade as new increments are added. Regular change leads to messy code as new functionality is added in whatever way is possible. It becomes increasingly difficult and costly to add new features to a system. To reduce structural degradation and general code messiness, agile methods suggest that you should regularly refactor (improve and restructure) the software.

The problems of incremental development become particularly acute for large, complex, long-lifetime systems, where different teams develop different parts of the system. Large systems need a stable framework or architecture, and the responsibilities of the different teams working on parts of the system need to be clearly defined with respect to that architecture. This has to be planned in advance rather than developed incrementally.

Incremental development does not mean that you have to deliver each increment to the system customer. You can develop a system incrementally and expose it to customers and other stakeholders for comment, without necessarily delivering it and deploying it in the customer's environment. Incremental delivery (covered in Section 2.3.2) means that the software is used in real, operational processes, so user feedback is likely to be realistic. However, providing feedback is not always possible as experimenting with new software can disrupt normal business processes.

Figure 2.3 Reuse-oriented software engineering

2.1.3 Integration and configuration

In the majority of software projects, there is some software reuse. This often happens informally when people working on the project know of or search for code that is similar to what is required. They look for these, modify them as needed, and integrate them with the new code that they have developed.

This informal reuse takes place regardless of the development process that is used. However, since 2000, software development processes that focus on the reuse of existing software have become widely used. Reuse-oriented approaches rely on a base of reusable software components and an integrating framework for the composition of these components.

Three types of software components are frequently reused:

1. Stand-alone application systems that are configured for use in a particular environment. These systems are general-purpose systems that have many features, but they have to be adapted for use in a specific application.
2. Collections of objects that are developed as a component or as a package to be integrated with a component framework such as the Java Spring framework (Wheeler and White 2013).
3. Web services that are developed according to service standards and that are available for remote invocation over the Internet.

Figure 2.3 shows a general process model for reuse-based development, based on integration and configuration. The stages in this process are:

1. *Requirements specification* The initial requirements for the system are proposed. These do not have to be elaborated in detail but should include brief descriptions of essential requirements and desirable system features.
2. *Software discovery and evaluation* Given an outline of the software requirements, a search is made for components and systems that provide the functionality required. Candidate components and systems are evaluated to see if

Software development tools

Software development tools are programs that are used to support software engineering process activities. These tools include requirements management tools, design editors, refactoring support tools, compilers, debuggers, bug trackers, and system building tools.

Software tools provide process support by automating some process activities and by providing information about the software that is being developed. For example:

- The development of graphical system models as part of the requirements specification or the software design
- The generation of code from these graphical models
- The generation of user interfaces from a graphical interface description that is created interactively by the user
- Program debugging through the provision of information about an executing program
- The automated translation of programs written using an old version of a programming language to a more recent version

Tools may be combined within a framework called an Interactive Development Environment or IDE. This provides a common set of facilities that tools can use so that it is easier for tools to communicate and operate in an integrated way.

http://software-engineering-book.com/web/software-tools/

they meet the essential requirements and if they are generally suitable for use in the system.

3. *Requirements refinement* During this stage, the requirements are refined using information about the reusable components and applications that have been discovered. The requirements are modified to reflect the available components, and the system specification is re-defined. Where modifications are impossible, the component analysis activity may be reentered to search for alternative solutions.

4. *Application system configuration* If an off-the-shelf application system that meets the requirements is available, it may then be configured for use to create the new system.

5. *Component adaptation and integration* If there is no off-the-shelf system, individual reusable components may be modified and new components developed. These are then integrated to create the system.

Reuse-oriented software engineering, based around configuration and integration, has the obvious advantage of reducing the amount of software to be developed and so reducing cost and risks. It usually also leads to faster delivery of the software. However, requirements compromises are inevitable, and this may lead to a system

that does not meet the real needs of users. Furthermore, some control over the system evolution is lost as new versions of the reusable components are not under the control of the organization using them.

Software reuse is very important, and so several chapters in the third I have dedicated several chapters in the 3rd part of the book to this topic. General issues of software reuse are covered in Chapter 15, component-based software engineering in Chapters 16 and 17, and service-oriented systems in Chapter 18.

2.2 Process activities

Real software processes are interleaved sequences of technical, collaborative, and managerial activities with the overall goal of specifying, designing, implementing, and testing a software system. Generally, processes are now tool-supported. This means that software developers may use a range of software tools to help them, such as requirements management systems, design model editors, program editors, automated testing tools, and debuggers.

The four basic process activities of specification, development, validation, and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved. How these activities are carried out depends on the type of software being developed, the experience and competence of the developers, and the type of organization developing the software.

2.2.1 Software specification

Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development. Requirements engineering is a particularly critical stage of the software process, as mistakes made at this stage inevitably lead to later problems in the system design and implementation.

Before the requirements engineering process starts, a company may carry out a feasibility or marketing study to assess whether or not there is a need or a market for the software and whether or not it is technically and financially realistic to develop the software required. Feasibility studies are short-term, relatively cheap studies that inform the decision of whether or not to go ahead with a more detailed analysis.

The requirements engineering process (Figure 2.4) aims to produce an agreed requirements document that specifies a system satisfying stakeholder requirements. Requirements are usually presented at two levels of detail. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification.

Figure 2.4 The requirements engineering process

There are three main activities in the requirements engineering process:

1. *Requirements elicitation and analysis* This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on. This may involve the development of one or more system models and prototypes. These help you understand the system to be specified.

2. *Requirements specification* Requirements specification is the activity of translating the information gathered during requirements analysis into a document that defines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are a more detailed description of the functionality to be provided.

3. *Requirements validation* This activity checks the requirements for realism, consistency, and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

Requirements analysis continues during definition and specification, and new requirements come to light throughout the process. Therefore, the activities of analysis, definition, and specification are interleaved.

In agile methods, requirements specification is not a separate activity but is seen as part of system development. Requirements are informally specified for each increment of the system just before that increment is developed. Requirements are specified according to user priorities. The elicitation of requirements comes from users who are part of or work closely with the development team.

Figure 2.5 A general model of the design process

2.2.2 Software design and implementation

The implementation stage of software development is the process of developing an executable system for delivery to the customer. Sometimes this involves separate activities of software design and programming. However, if an agile approach to development is used, design and implementation are interleaved, with no formal design documents produced during the process. Of course, the software is still designed, but the design is recorded informally on whiteboards and programmer's notebooks.

A software design is a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and, sometimes, the algorithms used. Designers do not arrive at a finished design immediately but develop the design in stages. They add detail as they develop their design, with constant backtracking to modify earlier designs.

Figure 2.5 is an abstract model of the design process showing the inputs to the design process, process activities, and the process outputs. The design process activities are both interleaved and interdependent. New information about the design is constantly being generated, and this affects previous design decisions. Design rework is therefore inevitable.

Most software interfaces with other software systems. These other systems include the operating system, database, middleware, and other application systems. These make up the "software platform,' the environment in which the software will execute. Information about this platform is an essential input to the design process, as designers must decide how best to integrate it with its environment. If the system is to process existing data, then the description of that data may be included in the platform specification. Otherwise, the data description must be an input to the design process so that the system data organization can be defined.

The activities in the design process vary, depending on the type of system being developed. For example, real-time systems require an additional stage of timing design but may not include a database, so there is no database design involved. Figure 2.5 shows four activities that may be part of the design process for information systems:

1. *Architectural design,* where you identify the overall structure of the system, the principal components (sometimes called subsystems or modules), their relationships, and how they are distributed.
2. *Database design,* where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.
3. *Interface design,* where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component may be used by other components without them having to know how it is implemented. Once interface specifications are agreed, the components can be separately designed and developed.
4. *Component selection and design,* where you search for reusable components and, if no suitable components are available, design new software components. The design at this stage may be a simple component description with the implementation details left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model expressed in the UML. The design model may then be used to automatically generate an implementation.

These activities lead to the design outputs, which are also shown in Figure 2.5. For critical systems, the outputs of the design process are detailed design documents setting out precise and accurate descriptions of the system. If a model-driven approach is used (Chapter 5), the design outputs are design diagrams. Where agile methods of development are used, the outputs of the design process may not be separate specification documents but may be represented in the code of the program.

The development of a program to implement a system follows naturally from system design. Although some classes of program, such as safety-critical systems, are usually designed in detail before any implementation begins, it is more common for design and program development to be interleaved. Software development tools may be used to generate a skeleton program from a design. This includes code to

Figure 2.6 Stages of testing

define and implement interfaces, and, in many cases, the developer need only add details of the operation of each program component.

Programming is an individual activity, and there is no general process that is usually followed. Some programmers start with components that they understand, develop these, and then move on to less understood components. Others take the opposite approach, leaving familiar components till last because they know how to develop them. Some developers like to define data early in the process and then use this to drive the program development; others leave data unspecified for as long as possible.

Normally, programmers carry out some testing of the code they have developed. This often reveals program defects (bugs) that must be removed from the program. Finding and fixing program defects is called *debugging*. Defect testing and debugging are different processes. Testing establishes the existence of defects. Debugging is concerned with locating and correcting these defects.

When you are debugging, you have to generate hypotheses about the observable behavior of the program and then test these hypotheses in the hope of finding the fault that caused the output anomaly. Testing the hypotheses may involve tracing the program code manually. It may require new test cases to localize the problem. Interactive debugging tools, which show the intermediate values of program variables and a trace of the statements executed, are usually used to support the debugging process.

2.2.3 Software validation

Software validation or, more generally, verification and validation (V & V) is intended to show that a system both conforms to its specification and meets the expectations of the system customer. Program testing, where the system is executed using simulated test data, is the principal validation technique. Validation may also involve checking processes, such as inspections and reviews, at each stage of the software process from user requirements definition to program development. However, most V & V time and effort is spent on program testing.

Except for small programs, systems should not be tested as a single, monolithic unit. Figure 2.6 shows a three-stage testing process in which system components are individually tested, then the integrated system is tested. For custom software, customer testing involves testing the system with real customer data. For products that are sold as applications, customer testing is sometimes called beta testing where selected users try out and comment on the software.

The stages in the testing process are:

1. *Component testing* The components making up the system are tested by the people developing the system. Each component is tested independently, without other system components. Components may be simple entities such as functions or object classes or may be coherent groupings of these entities. Test automation tools, such as JUnit for Java, that can rerun tests when new versions of the component are created, are commonly used (Koskela 2013).
2. *System testing* System components are integrated to create a complete system. This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems. It is also concerned with showing that the system meets its functional and non-functional requirements, and testing the emergent system properties. For large systems, this may be a multistage process where components are integrated to form subsystems that are individually tested before these subsystems are integrated to form the final system.
3. *Customer testing* This is the final stage in the testing process before the system is accepted for operational use. The system is tested by the system customer (or potential customer) rather than with simulated test data. For custom-built software, customer testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data. Customer testing may also reveal requirements problems where the system's facilities do not really meet the users' needs or the system performance is unacceptable. For products, customer testing shows how well the software product meets the customer's needs.

Ideally, component defects are discovered early in the testing process, and interface problems are found when the system is integrated. However, as defects are discovered, the program must be debugged, and this may require other stages in the testing process to be repeated. Errors in program components, say, may come to light during system testing. The process is therefore an iterative one with information being fed back from later stages to earlier parts of the process.

Normally, component testing is simply part of the normal development process. Programmers make up their own test data and incrementally test the code as it is developed. The programmer knows the component and is therefore the best person to generate test cases.

If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that increment. In test-driven development, which is a normal part of agile processes, tests are developed along with the requirements before development starts. This helps the testers and developers to understand the requirements and ensures that there are no delays as test cases are created.

When a plan-driven software process is used (e.g., for critical systems development), testing is driven by a set of test plans. An independent team of testers works

Figure 2.7 Testing phases in a plan-driven software process

from these test plans, which have been developed from the system specification and design. Figure 2.7 illustrates how test plans are the link between testing and development activities. This is sometimes called the V-model of development (turn it on its side to see the V). The V-model shows the software validation activities that correspond to each stage of the waterfall process model.

When a system is to be marketed as a software product, a testing process called beta testing is often used. Beta testing involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors that may not have been anticipated by the product developers. After this feedback, the software product may be modified and released for further beta testing or general sale.

2.2.4 Software evolution

The flexibility of software is one of the main reasons why more and more software is being incorporated into large, complex systems. Once a decision has been made to manufacture hardware, it is very expensive to make changes to the hardware design. However, changes can be made to software at any time during or after the system development. Even extensive changes are still much cheaper than corresponding changes to system hardware.

Historically, there has always been a split between the process of software development and the process of software evolution (software maintenance). People think of software development as a creative activity in which a software system is developed from an initial concept through to a working system. However, they sometimes think of software maintenance as dull and uninteresting. They think that software maintenance is less interesting and challenging than original software development.

This distinction between development and maintenance is increasingly irrelevant. Very few software systems are completely new systems, and it makes much more

Figure 2.8 Software system evolution

sense to see development and maintenance as a continuum. Rather than two separate processes, it is more realistic to think of software engineering as an evolutionary process (Figure 2.8) where software is continually changed over its lifetime in response to changing requirements and customer needs.

2.3 Coping with change

Change is inevitable in all large software projects. The system requirements change as businesses respond to external pressures, competition, and changed management priorities. As new technologies become available, new approaches to design and implementation become possible. Therefore whatever software process model is used, it is essential that it can accommodate changes to the software being developed.

Change adds to the costs of software development because it usually means that work that has been completed has to be redone. This is called rework. For example, if the relationships between the requirements in a system have been analyzed and new requirements are then identified, some or all of the requirements analysis has to be repeated. It may then be necessary to redesign the system to deliver the new requirements, change any programs that have been developed, and retest the system.

Two related approaches may be used to reduce the costs of rework:

1. *Change anticipation,* where the software process includes activities that can anticipate or predict possible changes before significant rework is required. For example, a prototype system may be developed to show some key features of the system to customers. They can experiment with the prototype and refine their requirements before committing to high software production costs.

2. *Change tolerance,* where the process and software are designed so that changes can be easily made to the system. This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have to be altered to incorporate the change.

In this section, I discuss two ways of coping with change and changing system requirements:

1. *System prototyping,* where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of design decisions. This is a method of change anticipation as it allows users to experiment with the system before delivery and so refine their requirements. The number of requirements change proposals made after delivery is therefore likely to be reduced.
2. *Incremental delivery,* where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance. It avoids the premature commitment to requirements for the whole system and allows changes to be incorporated into later increments at relatively low cost.

The notion of refactoring, namely, improving the structure and organization of a program, is also an important mechanism that supports change tolerance. I discuss this in Chapter 3 (Agile methods).

2.3.1 Prototyping

A prototype is an early version of a software system that is used to demonstrate concepts, try out design options, and find out more about the problem and its possible solutions. Rapid, iterative development of the prototype is essential so that costs are controlled and system stakeholders can experiment with the prototype early in the software process.

A software prototype can be used in a software development process to help anticipate changes that may be required:

1. In the requirements engineering process, a prototype can help with the elicitation and validation of system requirements.
2. In the system design process, a prototype can be used to explore software solutions and in the development of a user interface for the system.

System prototypes allow potential users to see how well the system supports their work. They may get new ideas for requirements and find areas of strength and weakness in the software. They may then propose new system requirements. Furthermore, as the prototype is developed, it may reveal errors and omissions in the system requirements. A feature described in a specification may seem to be clear and useful. However, when that function is combined with other functions, users often find that their initial view was incorrect or incomplete. The system specification can then be modified to reflect the changed understanding of the requirements.

Figure 2.9 Prototype development

A system prototype may be used while the system is being designed to carry out design experiments to check the feasibility of a proposed design. For example, a database design may be prototyped and tested to check that it supports efficient data access for the most common user queries. Rapid prototyping with end-user involvement is the only sensible way to develop user interfaces. Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good enough for expressing the user interface requirements and design.

A process model for prototype development is shown in Figure 2.9. The objectives of prototyping should be made explicit from the start of the process. These may be to develop the user interface, to develop a system to validate functional system requirements, or to develop a system to demonstrate the application to managers. The same prototype usually cannot meet all objectives. If the objectives are left unstated, management or end-users may misunderstand the function of the prototype. Consequently, they may not get the benefits that they expected from the prototype development.

The next stage in the process is to decide what to put into and, perhaps more importantly, what to leave out of the prototype system. To reduce prototyping costs and accelerate the delivery schedule, you may leave some functionality out of the prototype. You may decide to relax non-functional requirements such as response time and memory utilization. Error handling and management may be ignored unless the objective of the prototype is to establish a user interface. Standards of reliability and program quality may be reduced.

The final stage of the process is prototype evaluation. Provision must be made during this stage for user training, and the prototype objectives should be used to derive a plan for evaluation. Potential users need time to become comfortable with a new system and to settle into a normal pattern of usage. Once they are using the system normally, they then discover requirements errors and omissions. A general problem with prototyping is that users may not use the prototype in the same way as they use the final system. Prototype testers may not be typical of system users. There may not be enough time to train users during prototype evaluation. If the prototype is slow, the evaluators may adjust their way of working and avoid those system features that have slow response times. When provided with better response in the final system, they may use it in a different way.

Figure 2.10
Incremental delivery

2.3.2 Incremental delivery

Incremental delivery (Figure 2.10) is an approach to software development where some of the developed increments are delivered to the customer and deployed for use in their working environment. In an incremental delivery process, customers define which of the services are most important and which are least important to them. A number of delivery increments are then defined, with each increment providing a subset of the system functionality. The allocation of services to increments depends on the service priority, with the highest priority services implemented and delivered first.

Once the system increments have been identified, the requirements for the services to be delivered in the first increment are defined in detail and that increment is developed. During development, further requirements analysis for later increments can take place, but requirements changes for the current increment are not accepted.

Once an increment is completed and delivered, it is installed in the customer's normal working environment. They can experiment with the system, and this helps them clarify their requirements for later system increments. As new increments are completed, they are integrated with existing increments so that system functionality improves with each delivered increment.

Incremental delivery has a number of advantages:

1. Customers can use the early increments as prototypes and gain experience that informs their requirements for later system increments. Unlike prototypes, these are part of the real system, so there is no relearning when the complete system is available.

2. Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical requirements, so they can use the software immediately.

3. The process maintains the benefits of incremental development in that it should be relatively easy to incorporate changes into the system.

4. As the highest priority services are delivered first and later increments then integrated, the most important system services receive the most testing. This means that customers are less likely to encounter software failures in the most important parts of the system.

However, there are problems with incremental delivery. In practice, it only works in situations where a brand-new system is being introduced and the system evaluators are given time to experiment with the new system. Key problems with this approach are:

1. Iterative delivery is problematic when the new system is intended to replace an existing system. Users need all of the functionality of the old system and are usually unwilling to experiment with an incomplete new system. It is often impractical to use the old and the new systems alongside each other as they are likely to have different databases and user interfaces.
2. Most systems require a set of basic facilities that are used by different parts of the system. As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
3. The essence of iterative processes is that the specification is developed in conjunction with the software. However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract. In the incremental approach, there is no complete system specification until the final increment is specified. This requires a new form of contract, which large customers such as government agencies may find difficult to accommodate.

For some types of systems, incremental development and delivery is not the best approach. These are very large systems where development may involve teams working in different locations, some embedded systems where the software depends on hardware development, and some critical systems where all the requirements must be analyzed to check for interactions that may compromise the safety or security of the system.

These large systems, of course, suffer from the same problems of uncertain and changing requirements. Therefore, to address these problems and get some of the benefits of incremental development, a system prototype may be developed and used as a platform for experiments with the system requirements and design. With the experience gained from the prototype, definitive requirements can then be agreed.

2.4 Process improvement

Nowadays, there is a constant demand from industry for cheaper, better software, which has to be delivered to ever-tighter deadlines. Consequently, many software companies have turned to software process improvement as a way of enhancing the

Figure 2.11 The process improvement cycle

quality of their software, reducing costs, or accelerating their development processes. Process improvement means understanding existing processes and changing these processes to increase product quality and/or reduce costs and development time. I cover general issues of process measurement and process improvement in detail in web Chapter 26.

Two quite different approaches to process improvement and change are used:

1. The process maturity approach, which has focused on improving process and project management and introducing good software engineering practice into an organization. The level of process maturity reflects the extent to which good technical and management practice has been adopted in organizational software development processes. The primary goals of this approach are improved product quality and process predictability.

2. The agile approach, which has focused on iterative development and the reduction of overheads in the software process. The primary characteristics of agile methods are rapid delivery of functionality and responsiveness to changing customer requirements. The improvement philosophy here is that the best processes are those with the lowest overheads and agile approaches can achieve this. I describe agile approaches in Chapter 3.

People who are enthusiastic about and committed to each of these approaches are generally skeptical of the benefits of the other. The process maturity approach is rooted in plan-driven development and usually requires increased "overhead," in the sense that activities are introduced that are not directly relevant to program development. Agile approaches focus on the code being developed and deliberately minimize formality and documentation.

The general process improvement process underlying the process maturity approach is a cyclical process, as shown in Figure 2.11. The stages in this process are:

1. *Process measurement* You measure one or more attributes of the software process or product. These measurements form a baseline that helps you decide if

process improvements have been effective. As you introduce improvements, you re-measure the same attributes, which will hopefully have improved in some way.

2. *Process analysis* The current process is assessed, and process weaknesses and bottlenecks are identified. Process models (sometimes called process maps) that describe the process may be developed during this stage. The analysis may be focused by considering process characteristics such as rapidity and robustness.

3. *Process change* Process changes are proposed to address some of the identified process weaknesses. These are introduced, and the cycle resumes to collect data about the effectiveness of the changes.

Without concrete data on a process or the software developed using that process, it is impossible to assess the value of process improvement. However, companies starting the process improvement process are unlikely to have process data available as an improvement baseline. Therefore, as part of the first cycle of changes, you may have to collect data about the software process and to measure software product characteristics.

Process improvement is a long-term activity, so each of the stages in the improvement process may last several months. It is also a continuous activity as, whatever new processes are introduced, the business environment will change and the new processes will themselves have to evolve to take these changes into account.

The notion of process maturity was introduced in the late 1980s when the Software Engineering Institute (SEI) proposed their model of process capability maturity (Humphrey 1988). The maturity of a software company's processes reflects the process management, measurement, and use of good software engineering practices in the company. This idea was introduced so that the U.S. Department of Defense could assess the software engineering capability of defense contractors, with a view to limiting contracts to those contractors who had reached a required level of process maturity. Five levels of process maturity were proposed. as shown in Figure 2.12. These have evolved and developed over the last 25 years (Chrissis, Konrad, and Shrum 2011), but the fundamental ideas in Humphrey's model are still the basis of software process maturity assessment.

The levels in the process maturity model are:

1. *Initial* The goals associated with the process area are satisfied, and for all processes the scope of the work to be performed is explicitly set out and communicated to the team members.

2. *Managed* At this level, the goals associated with the process area are met, and organizational policies are in place that define when each process should be used. There must be documented project plans that define the project goals. Resource management and process monitoring procedures must be in place across the institution.

3. *Defined* This level focuses on organizational standardization and deployment of processes. Each project has a managed process that is adapted to the project requirements from a defined set of organizational processes. Process assets and process measurements must be collected and used for future process improvements.

Figure 2.12 Capability maturity levels

4. *Quantitatively managed* At this level, there is an organizational responsibility to use statistical and other quantitative methods to control subprocesses. That is, collected process and product measurements must be used in process management.

5. *Optimizing* At this highest level, the organization must use the process and product measurements to drive process improvement. Trends must be analyzed and the processes adapted to changing business needs.

The work on process maturity levels has had a major impact on the software industry. It focused attention on the software engineering processes and practices that were used and led to significant improvements in software engineering capability. However, there is too much overhead in formal process improvement for small companies, and maturity estimation with agile processes is difficult. Consequently, only large software companies now use this maturity-focused approach to software process improvement.

KEY POINTS

- Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- General process models describe the organization of software processes. Examples of these general models include the waterfall model, incremental development, and reusable component configuration and integration.

- Requirements engineering is the process of developing a software specification. Specifications are intended to communicate the system needs of the customer to the system developers.
- Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
- Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- Software evolution takes place when you change existing software systems to meet new requirements. Changes are continuous, and the software must evolve to remain useful.
- Processes should include activities to cope with change. This may involve a prototyping phase that helps avoid poor decisions on requirements and design. Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- Process improvement is the process of improving existing software processes to improve software quality, lower development costs, or reduce development time. It is a cyclic process involving process measurement, analysis, and change.

FURTHER READING

"Process Models in Software Engineering." This is an excellent overview of a wide range of software engineering process models that have been proposed. (W. Scacchi, *Encyclopaedia of Software Engineering,* ed. J. J. Marciniak, John Wiley & Sons, 2001) http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf

Software Process Improvement: Results and Experience from the Field. This book is a collection of papers focusing on process improvement case studies in several small and medium-sized Norwegian companies. It also includes a good introduction to the general issues of process improvement. (Conradi, R., Dybå, T., Sjøberg, D., and Ulsund, T. (eds.), Springer, 2006).

"Software Development Life Cycle Models and Methodologies." This blog post is a succinct summary of several software process models that have been proposed and used. It discusses the advantages and disadvantages of each of these models (M. Sami, 2012). http://melsatar.wordpress.com/2012/03/15/software-development-life-cycle-models-and-methodologies/

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-engineering/

EXERCISES

2.1. Suggest the most appropriate generic software process model that might be used as a basis for managing the development of the following systems. Explain your answer according to the type of system being developed:

A system to control antilock braking in a car

A virtual reality system to support software maintenance

A university accounting system that replaces an existing system

An interactive travel planning system that helps users plan journeys with the lowest environmental impact

2.2. Incremental software development could be very effectively used for customers who do not have a clear idea about the systems needed for their operations. Discuss.

2.3. Consider the integration and configuration process model shown in Figure 2.3. Explain why it is essential to repeat the requirements engineering activity in the process.

2.4. Suggest why it is important to make a distinction between developing the user requirements and developing system requirements in the requirements engineering process.

2.5. Using an example, explain why the design activities of architectural design, database design, interface design, and component design are interdependent.

2.6. Explain why software testing should always be an incremental, staged activity. Are programmers the best people to test the programs that they have developed?

2.7. Imagine that a government wants a software program that helps to keep track of the utilization of the country's vast mineral resources. Although the requirements put forward by the government were not very clear, a software company was tasked with the development of a prototype. The government found the prototype impressive, and asked it be extended to be the actual system that would be used. Discuss the pros and cons of taking this approach.

2.8. You have developed a prototype of a software system and your manager is very impressed by it. She proposes that it should be put into use as a production system, with new features added as required. This avoids the expense of system development and makes the system immediately useful. Write a short report for your manager explaining why prototype systems should not normally be used as production systems.

2.9. Suggest two advantages and two disadvantages of the approach to process assessment and improvement that is embodied in the SEI's Capability Maturity framework.

2.10. Historically, the introduction of technology has caused profound changes in the labor market and, temporarily at least, displaced people from jobs. Discuss whether the introduction of extensive process automation is likely to have the same consequences for software engineers. If you don't think it will, explain why not. If you think that it will reduce job opportunities, is it ethical for the engineers affected to passively or actively resist the introduction of this technology?

REFERENCES

Abrial, J. R. 2005. *The B Book: Assigning Programs to Meanings*. Cambridge, UK: Cambridge University Press.

———. 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.

Boehm, B. W. (1988). "A Spiral Model of Software Development and Enhancement." *IEEE Computer*, 21 (5), 61–72. doi:10.1145/12944.12948

Boehm, B. W., and R. Turner. 2004. "Balancing Agility and Discipline: Evaluating and Integrating Agile and Plan-Driven Methods." In *26th Int. Conf on Software Engineering*, Edinburgh, Scotland. doi:10.1109/ICSE.2004.1317503.

Chrissis, M. B., M. Konrad, and S. Shrum. 2011. *CMMI for Development: Guidelines for Process Integration and Product Improvement, 3rd ed.* Boston: Addison-Wesley.

Humphrey, W. S. 1988. "Characterizing the Software Process: A Maturity Framework." *IEEE Software* 5 (2): 73–79. doi:10.1109/2.59.

Koskela, L. 2013. *Effective Unit Testing: A Guide for Java Developers*. Greenwich, CT: Manning Publications.

Krutchen, P. 2003. *The Rational Unified Process—An Introduction, 3rd ed.* Reading, MA: Addison-Wesley.

Royce, W. W. 1970. "Managing the Development of Large Software Systems: Concepts and Techniques." In *IEEE WESTCON*, 1–9. Los Angeles, CA.

Wheeler, W., and J. White. 2013. *Spring in Practice*. Greenwich, CT: Manning Publications.

3

Agile software development

Objectives

The objective of this chapter is to introduce you to agile software development methods. When you have read the chapter, you will:

- understand the rationale for agile software development methods, the agile manifesto, and the differences between agile and plan-driven development;
- know about important agile development practices such as user stories, refactoring, pair programming and test-first development;
- understand the Scrum approach to agile project management;
- understand the issues of scaling agile development methods and combining agile approaches with plan-driven approaches in the development of large software systems.

Contents

3.1 Agile methods
3.2 Agile development techniques
3.3 Agile project management
3.4 Scaling agile methods

Businesses now operate in a global, rapidly changing environment. They have to respond to new opportunities and markets, changing economic conditions and the emergence of competing products and services. Software is part of almost all business operations, so new software has to be developed quickly to take advantage of new opportunities and to respond to competitive pressure. Rapid software development and delivery is therefore the most critical requirement for most business systems. In fact, businesses may be willing to trade off software quality and compromise on requirements if they can deploy essential new software quickly.

Because these businesses are operating in a changing environment, it is practically impossible to derive a complete set of stable software requirements. Requirements change because customers find it impossible to predict how a system will affect working practices, how it will interact with other systems, and what user operations should be automated. It may only be after a system has been delivered and users gain experience with it that the real requirements become clear. Even then, external factors drive requirements change.

Plan-driven software development processes that completely specify the requirements and then design, build, and test a system are not geared to rapid software development. As the requirements change or as requirements problems are discovered, the system design or implementation has to be reworked and retested. As a consequence, a conventional waterfall or specification-based process is usually a lengthy one, and the final software is delivered to the customer long after it was originally specified.

For some types of software, such as safety-critical control systems, where a complete analysis of the system is essential, this plan-driven approach is the right one. However, in a fast-moving business environment, it can cause real problems. By the time the software is available for use, the original reason for its procurement may have changed so radically that the software is effectively useless. Therefore, for business systems in particular, development processes that focus on rapid software development and delivery are essential.

The need for rapid software development and processes that can handle changing requirements has been recognized for many years (Larman and Basili 2003). However, faster software development really took off in the late 1990s with the development of the idea of "agile methods" such as Extreme Programming (Beck 1999), Scrum (Schwaber and Beedle 2001), and DSDM (Stapleton 2003).

Rapid software development became known as agile development or agile methods. These agile methods are designed to produce useful software quickly. All of the agile methods that have been proposed share a number of common characteristics:

1. The processes of specification, design and implementation are interleaved. There is no detailed system specification, and design documentation is minimized or generated automatically by the programming environment used to implement the system. The user requirements document is an outline definition of the most important characteristics of the system.

2. The system is developed in a series of increments. End-users and other system stakeholders are involved in specifying and evaluating each increment.

Figure 3.1 Plan-driven and agile development

They may propose changes to the software and new requirements that should be implemented in a later version of the system.

3. Extensive tool support is used to support the development process. Tools that may be used include automated testing tools, tools to support configuration management, and system integration and tools to automate user interface production.

Agile methods are incremental development methods in which the increments are small, and, typically, new releases of the system are created and made available to customers every two or three weeks. They involve customers in the development process to get rapid feedback on changing requirements. They minimize documentation by using informal communications rather than formal meetings with written documents.

Agile approaches to software development consider design and implementation to be the central activities in the software process. They incorporate other activities, such as requirements elicitation and testing, into design and implementation. By contrast, a plan-driven approach to software engineering identifies separate stages in the software process with outputs associated with each stage. The outputs from one stage are used as a basis for planning the following process activity.

Figure 3.1 shows the essential distinctions between plan-driven and agile approaches to system specification. In a plan-driven software development process, iteration occurs within activities, with formal documents used to communicate between stages of the process. For example, the requirements will evolve, and, ultimately, a requirements specification will be produced. This is then an input to the design and implementation process. In an agile approach, iteration occurs across activities. Therefore, the requirements and the design are developed together rather than separately.

In practice, as I explain in Section 3.4.1, plan-driven processes are often used along with agile programming practices, and agile methods may incorporate some planned

activities apart from programming and testing. It is perfectly feasible, in a plan-driven process, to allocate requirements and plan the design and development phase as a series of increments. An agile process is not inevitably code-focused, and it may produce some design documentation. Agile developers may decide that an iteration should not produce new code but rather should produce system models and documentation.

3.1 Agile methods

In the 1980s and early 1990s, there was a widespread view that the best way to achieve better software was through careful project planning, formalized quality assurance, use of analysis and design methods supported by software tools, and controlled and rigorous software development processes. This view came from the software engineering community that was responsible for developing large, long-lived software systems such as aerospace and government systems.

This plan-driven approach was developed for software developed by large teams, working for different companies. Teams were often geographically dispersed and worked on the software for long periods of time. An example of this type of software is the control systems for a modern aircraft, which might take up to 10 years from initial specification to deployment. Plan-driven approaches involve a significant overhead in planning, designing, and documenting the system. This overhead is justified when the work of multiple development teams has to be coordinated, when the system is a critical system, and when many different people will be involved in maintaining the software over its lifetime.

However, when this heavyweight, plan-driven development approach is applied to small and medium-sized business systems, the overhead involved is so large that it dominates the software development process. More time is spent on how the system should be developed than on program development and testing. As the system requirements change, rework is essential and, in principle at least, the specification and design have to change with the program.

Dissatisfaction with these heavyweight approaches to software engineering led to the development of agile methods in the late 1990s. These methods allowed the development team to focus on the software itself rather than on its design and documentation. They are best suited to application development where the system requirements usually change rapidly during the development process. They are intended to deliver working software quickly to customers, who can then propose new and changed requirements to be included in later iterations of the system. They aim to cut down on process bureaucracy by avoiding work that has dubious long-term value and eliminating documentation that will probably never be used.

The philosophy behind agile methods is reflected in the agile manifesto (http://agilemanifesto.org) issued by the leading developers of these methods. This manifesto states:

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Embrace change	Expect the system requirements to change, and so design the system to accommodate these changes.
Incremental delivery	The software is developed in increments, with the customer specifying the requirements to be included in each increment.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.
People, not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.

Figure 3.2 The principles of agile methods

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more[†].

All agile methods suggest that software should be developed and delivered incrementally. These methods are based on different agile processes but they share a set of principles, based on the agile manifesto, and so they have much in common. I have listed these principles in Figure 3.2.

Agile methods have been particularly successful for two kinds of system development.

1. Product development where a software company is developing a small or medium-sized product for sale. Virtually all software products and apps are now developed using an agile approach.
2. Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are few external stakeholders and regulations that affect the software.

Agile methods work well in these situations because it is possible to have continuous communications between the product manager or system customer and the development team. The software itself is a stand-alone system rather than tightly integrated with other systems being developed at the same time. Consequently, there is no need to coordinate parallel development streams. Small and medium-sized

[†]http://agilemanifesto.org/

Figure 3.3 The XP release cycle

systems can be developed by co-located teams, so informal communications among team members work well.

3.2 Agile development techniques

The ideas underlying agile methods were developed around the same time by a number of different people in the 1990s. However, perhaps the most significant approach to changing software development culture was the development of Extreme Programming (XP). The name was coined by Kent Beck (Beck 1998) because the approach was developed by pushing recognized good practice, such as iterative development, to "extreme" levels. For example, in XP, several new versions of a system may be developed by different programmers, integrated, and tested in a day. Figure 3.3 illustrates the XP process to produce an increment of the system that is being developed.

In XP, requirements are expressed as scenarios (called user stories), which are implemented directly as a series of tasks. Programmers work in pairs and develop tests for each task before writing the code. All tests must be successfully executed when new code is integrated into the system. There is a short time gap between releases of the system.

Extreme programming was controversial as it introduced a number of agile practices that were quite different from the development practice of that time. These practices are summarized in Figure 3.4 and reflect the principles of the agile manifesto:

1. Incremental development is supported through small, frequent releases of the system. Requirements are based on simple customer stories or scenarios that are used as a basis for deciding what functionality should be included in a system increment.
2. Customer involvement is supported through the continuous engagement of the customer in the development team. The customer representative takes part in the development and is responsible for defining acceptance tests for the system.
3. People, not process, are supported through pair programming, collective ownership of the system code, and a sustainable development process that does not involve excessively long working hours.

Principle or practice	Description
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Incremental planning	Requirements are recorded on "story cards," and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development "tasks." See Figures 3.5 and 3.6.
On-site customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Refactoring	All developers are expected to refactor the code continuously as soon as potential code improvements are found. This keeps the code simple and maintainable.
Simple design	Enough design is carried out to meet the current requirements and no more.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Sustainable pace	Large amounts of overtime are not considered acceptable, as the net effect is often to reduce code quality and medium-term productivity.
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.

Figure 3.4 Extreme programming practices

4. Change is embraced through regular system releases to customers, test-first development, refactoring to avoid code degeneration, and continuous integration of new functionality.

5. Maintaining simplicity is supported by constant refactoring that improves code quality and by using simple designs that do not unnecessarily anticipate future changes to the system.

In practice, the application of Extreme Programming as originally proposed has proved to be more difficult than anticipated. It cannot be readily integrated with the management practices and culture of most businesses. Therefore, companies adopting agile methods pick and choose those XP practices that are most appropriate for their way of working. Sometimes these are incorporated into their own development processes but, more commonly, they are used in conjunction with a management-focused agile method such as Scrum (Rubin 2013).

Prescribing medication

Kate is a doctor who wishes to prescribe medication for a patient attending a clinic. The patient record is already displayed on her computer so she clicks on the medication field and can select 'current medication', 'new medication' or 'formulary'.

If she selects 'current medication', the system asks her to check the dose; If she wants to change the dose, she enters the new dose then confirms the prescription.

If she chooses 'new medication', the system assumes that she knows which medication to prescribe. She types the first few letters of the drug name. The system displays a list of possible drugs starting with these letters. She chooses the required medication and the system responds by asking her to check that the medication selected is correct. She enters the dose then confirms the prescription.

If she chooses 'formulary', the system displays a search box for the approved formulary. She can then search for the drug required. She selects a drug and is asked to check that the medication is correct. She enters the dose then confirms the prescription.

The system always checks that the dose is within the approved range. If it isn't, Kate is asked to change the dose.

After Kate has confirmed the prescription, it will be displayed for checking. She either clicks 'OK' or 'Change'. If she clicks 'OK', the prescription is recorded on the audit database. If she clicks on 'Change', she reenters the 'Prescribing medication' process.

Figure 3.5 A "prescribing medication" story

I am not convinced that XP on its own is a practical agile method for most companies, but its most significant contribution is probably the set of agile development practices that it introduced to the community. I discuss the most important of these practices in this section.

3.2.1 User stories

Software requirements always change. To handle these changes, agile methods do not have a separate requirements engineering activity. Rather, they integrate requirements elicitation with development. To make this easier, the idea of "user stories" was developed where a user story is a scenario of use that might be experienced by a system user.

As far as possible, the system customer works closely with the development team and discusses these scenarios with other team members. Together, they develop a "story card" that briefly describes a story that encapsulates the customer needs. The development team then aims to implement that scenario in a future release of the software. An example of a story card for the Mentcare system is shown in Figure 3.5. This is a short description of a scenario for prescribing medication for a patient.

User stories may be used in planning system iterations. Once the story cards have been developed, the development team breaks these down into tasks (Figure 3.6) and estimates the effort and resources required for implementing each task. This usually involves discussions with the customer to refine the requirements. The customer then prioritizes the stories for implementation, choosing those stories that can be

Figure 3.6 Examples of task cards for prescribing medication

used immediately to deliver useful business support. The intention is to identify useful functionality that can be implemented in about two weeks, when the next release of the system is made available to the customer.

Of course, as requirements change, the unimplemented stories change or may be discarded. If changes are required for a system that has already been delivered, new story cards are developed and again, the customer decides whether these changes should have priority over new functionality.

The idea of user stories is a powerful one—people find it much easier to relate to these stories than to a conventional requirements document or use cases. User stories can be helpful in getting users involved in suggesting requirements during an initial predevelopment requirements elicitation activity. I discuss this in more detail in Chapter 4.

The principal problem with user stories is completeness. It is difficult to judge if enough user stories have been developed to cover all of the essential requirements of a system. It is also difficult to judge if a single story gives a true picture of an activity. Experienced users are often so familiar with their work that they leave things out when describing it.

3.2.2 Refactoring

A fundamental precept of traditional software engineering is that you should design for change. That is, you should anticipate future changes to the software and design it so that these changes can be easily implemented. Extreme programming, however, has discarded this principle on the basis that designing for change is often wasted effort. It isn't worth taking time to add generality to a program to cope with change. Often the changes anticipated never materialize, or completely different change requests may actually be made.

Of course, in practice, changes will always have to be made to the code being developed. To make these changes easier, the developers of XP suggested that the code being developed should be constantly refactored. Refactoring (Fowler et al. 1999) means that the programming team look for possible improvements to the software and implements

them immediately. When team members see code that can be improved, they make these improvements even in situations where there is no immediate need for them.

A fundamental problem of incremental development is that local changes tend to degrade the software structure. Consequently, further changes to the software become harder and harder to implement. Essentially, the development proceeds by finding workarounds to problems, with the result that code is often duplicated, parts of the software are reused in inappropriate ways, and the overall structure degrades as code is added to the system. Refactoring improves the software structure and readability and so avoids the structural deterioration that naturally occurs when software is changed.

Examples of refactoring include the reorganization of a class hierarchy to remove duplicate code, the tidying up and renaming of attributes and methods, and the replacement of similar code sections, with calls to methods defined in a program library. Program development environments usually include tools for refactoring. These simplify the process of finding dependencies between code sections and making global code modifications.

In principle, when refactoring is part of the development process, the software should always be easy to understand and change as new requirements are proposed. In practice, this is not always the case. Sometimes development pressure means that refactoring is delayed because the time is devoted to the implementation of new functionality. Some new features and changes cannot readily be accommodated by code-level refactoring and require that the architecture of the system be modified.

3.2.3 Test-first development

As I discussed in the introduction to this chapter, one of the important differences between incremental development and plan-driven development is in the way that the system is tested. With incremental development, there is no system specification that can be used by an external testing team to develop system tests. As a consequence, some approaches to incremental development have a very informal testing process, in comparison with plan-driven testing.

Extreme Programming developed a new approach to program testing to address the difficulties of testing without a specification. Testing is automated and is central to the development process, and development cannot proceed until all tests have been successfully executed. The key features of testing in XP are:

1. test-first development,
2. incremental test development from scenarios,
3. user involvement in the test development and validation, and
4. the use of automated testing frameworks.

XP's test-first philosophy has now evolved into more general test-driven development techniques (Jeffries and Melnik 2007). I believe that test-driven development is one of the most important innovations in software engineering. Instead of writing code and then writing tests for that code, you write the tests before you write the code. This

Test 4: Dose checking

Input:
1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:
1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

Output:
OK or error message indicating that the dose is outside the safe range.

Figure 3.7 Test case description for dose checking

means that you can run the test as the code is being written and discover problems during development. I discuss test-driven development in more depth in Chapter 8.

Writing tests implicitly defines both an interface and a specification of behavior for the functionality being developed. Problems of requirements and interface misunderstandings are reduced. Test-first development requires there to be a clear relationship between system requirements and the code implementing the corresponding requirements. In XP, this relationship is clear because the story cards representing the requirements are broken down into tasks and the tasks are the principal unit of implementation.

In test-first development, the task implementers have to thoroughly understand the specification so that they can write tests for the system. This means that ambiguities and omissions in the specification have to be clarified before implementation begins. Furthermore, it also avoids the problem of "test-lag." This may happen when the developer of the system works at a faster pace than the tester. The implementation gets further and further ahead of the testing and there is a tendency to skip tests, so that the development schedule can be maintained.

XP's test-first approach assumes that user stories have been developed, and these have been broken down into a set of task cards, as shown in Figure 3.6. Each task generates one or more unit tests that check the implementation described in that task. Figure 3.7 is a shortened description of a test case that has been developed to check that the prescribed dose of a drug does not fall outside known safe limits.

The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system. As I explain in Chapter 8, acceptance testing is the process whereby the system is tested using customer data to check that it meets the customer's real needs.

Test automation is essential for test-first development. Tests are written as executable components before the task is implemented. These testing components should be stand-alone, should simulate the submission of input to be tested, and should check that the result meets the output specification. An automated test framework is a system that makes it easy to write executable tests and submit a set of tests for execution. Junit (Tahchiev et al. 2010) is a widely used example of an automated testing framework for Java programs.

As testing is automated, there is always a set of tests that can be quickly and easily executed. Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

Test-first development and automated testing usually result in a large number of tests being written and executed. However, there are problems in ensuring that test coverage is complete:

1. Programmers prefer programming to testing, and sometimes they take shortcuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
2. Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the "display logic" and workflow between screens.

It is difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage. Crucial parts of the system may not be executed and so will remain untested. Therefore, although a large set of frequently executed tests may give the impression that the system is complete and correct, this may not be the case. If the tests are not reviewed and further tests are written after development, then undetected bugs may be delivered in the system release.

3.2.4 Pair programming

Another innovative practice that was introduced in XP is that programmers work in pairs to develop the software. The programming pair sits at the same computer to develop the software. However, the same pair do not always program together. Rather, pairs are created dynamically so that all team members work with each other during the development process.

Pair programming has a number of advantages.

1. It supports the idea of collective ownership and responsibility for the system. This reflects Weinberg's idea of egoless programming (Weinberg 1971) where the software is owned by the team as a whole and individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
2. It acts as an informal review process because each line of code is looked at by at least two people. Code inspections and reviews (Chapter 24) are effective in discovering a high percentage of software errors. However, they are time consuming to organize and, typically, introduce delays into the development process. Pair programming is a less formal process that probably doesn't find as many errors as code inspections. However, it is cheaper and easier to organize than formal program inspections.
3. It encourages refactoring to improve the software structure. The problem with asking programmers to refactor in a normal development environment is that effort

involved is expended for long-term benefit. An developer who spends time refactoring may be judged to be less efficient than one who simply carries on developing code. Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

You might think that pair programming would be less efficient than individual programming. In a given time, a pair of developers would produce half as much code as two individuals working alone. Many companies that have adopted agile methods are suspicious of pair programming and do not use it. Other companies mix pair and individual programming with an experienced programmer working with a less experienced colleague when they have problems.

Formal studies of the value of pair programming have had mixed results. Using student volunteers, Williams and her collaborators (Williams et al. 2000) found that productivity with pair programming seems to be comparable to that of two people working independently. The reasons suggested are that pairs discuss the software before development and so probably have fewer false starts and less rework. Furthermore, the number of errors avoided by the informal inspection is such that less time is spent repairing bugs discovered during the testing process.

However, studies with more experienced programmers did not replicate these results (Arisholm et al. 2007). They found that there was a significant loss of productivity compared with two programmers working alone. There were some quality benefits, but these did not fully compensate for the pair-programming overhead. Nevertheless, the sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave. In itself, this may make pair programming worthwhile.

3.3 Agile project management

In any software business, managers need to know what is going on and whether or not a project is likely to meet its objectives and deliver the software on time with the proposed budget. Plan-driven approaches to software development evolved to meet this need. As I discussed in Chapter 23, managers draw up a plan for the project showing what should be delivered, when it should be delivered, and who will work on the development of the project deliverables. A plan-based approach requires a manager to have a stable view of everything that has to be developed and the development processes.

The informal planning and project control that was proposed by the early adherents of agile methods clashed with this business requirement for visibility. Teams were self-organizing, did not produce documentation, and planned development in very short cycles. While this can and does work for small companies developing software products, it is inappropriate for larger companies who need to know what is going on in their organization.

Like every other professional software development process, agile development has to be managed so that the best use is made of the time and resources available to

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than seven people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be "potentially shippable," which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of "to do" items that the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories, or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development, and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2 to 4 weeks long.
Velocity	An estimate of how much product backlog effort a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

Figure 3.8 Scrum terminology

the team. To address this issue, the Scrum agile method was developed (Schwaber and Beedle 2001; Rubin 2013) to provide a framework for organizing agile projects and, to some extent at least, provide external visibility of what is going on. The developers of Scrum wished to make clear that Scrum was not a method for project management in the conventional sense, so they deliberately invented new terminology, such as ScrumMaster, which replaced names such as project manager. Figure 3.8 summarizes Scrum terminology and what it means.

Scrum is an agile method insofar as it follows the principles from the agile manifesto, which I showed in Figure 3.2. However, it focuses on providing a framework for agile project organization, and it does not mandate the use of specific development

Figure 3.9 The Scrum sprint cycle

practices such as pair programming and test-first development. This means that it can be more easily integrated with existing practice in a company. Consequently, as agile methods have become a mainstream approach to software development, Scrum has emerged as the most widely used method.

The Scrum process or sprint cycle is shown in Figure 3.9. The input to the process is the product backlog. Each process iteration produces a product increment that could be delivered to customers.

The starting point for the Scrum sprint cycle is the product backlog—the list of items such as product features, requirements, and engineering improvement that have to be worked on by the Scrum team. The initial version of the product backlog may be derived from a requirements document, a list of user stories, or other description of the software to be developed.

While the majority of entries in the product backlog are concerned with the implementation of system features, other activities may also be included. Sometimes, when planning an iteration, questions that cannot be easily answered come to light and additional work is required to explore possible solutions. The team may carry out some prototyping or trial development to understand the problem and solution. There may also be backlog items to design the system architecture or to develop system documentation.

The product backlog may be specified at varying levels of detail, and it is the responsibility of the Product Owner to ensure that the level of detail in the specification is appropriate for the work to be done. For example, a backlog item could be a complete user story such as that shown in Figure 3.5, or it could simply be an instruction such as "Refactor user interface code" that leaves it up to the team to decide on the refactoring to be done.

Each sprint cycle lasts a fixed length of time, which is usually between 2 and 4 weeks. At the beginning of each cycle, the Product Owner prioritizes the items on the product backlog to define which are the most important items to be developed in that cycle. Sprints are never extended to take account of unfinished work. Items are returned to the product backlog if these cannot be completed within the allocated time for the sprint.

The whole team is then involved in selecting which of the highest priority items they believe can be completed. They then estimate the time required to complete these items. To make these estimates, they use the velocity attained in previous

sprints, that is, how much of the backlog could be covered in a single sprint. This leads to the creation of a sprint backlog—the work to be done during that sprint. The team self-organizes to decide who will work on what, and the sprint begins.

During the sprint, the team holds short daily meetings (Scrums) to review progress and, where necessary, to re-prioritize work. During the Scrum, all team members share information, describe their progress since the last meeting, bring up problems that have arisen, and state what is planned for the following day. Thus, everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them. Everyone participates in this short-term planning; there is no top-down direction from the ScrumMaster.

The daily interactions among Scrum teams may be coordinated using a Scrum board. This is an office whiteboard that includes information and post-it notes about the Sprint backlog, work done, unavailability of staff, and so on. This is a shared resource for the whole team, and anyone can change or move items on the board. It means that any team member can, at a glance, see what others are doing and what work remains to be done.

At the end of each sprint, there is a review meeting, which involves the whole team. This meeting has two purposes. First, it is a means of process improvement. The team reviews the way they have worked and reflects on how things could have been done better. Second, it provides input on the product and the product state for the product backlog review that precedes the next sprint.

While the ScrumMaster is not formally a project manager, in practice ScrumMasters take this role in many organizations that have a conventional management structure. They report on progress to senior management and are involved in longer-term planning and project budgeting. They may be involved in project administration (agreeing on holidays for staff, liaising with HR, etc.) and hardware and software purchases.

In various Scrum success stories (Schatz and Abdelshafi 2005; Mulder and van Vliet 2008; Bellouiti 2009), the things that users like about the Scrum method are:

1. The product is broken down into a set of manageable and understandable chunks that stakeholders can relate to.
2. Unstable requirements do not hold up progress.
3. The whole team has visibility of everything, and consequently team communication and morale are improved.
4. Customers see on-time delivery of increments and gain feedback on how the product works. They are not faced with last-minute surprises when a team announces that software will not be delivered as expected.
5. Trust between customers and developers is established, and a positive culture is created in which everyone expects the project to succeed.

Scrum, as originally designed, was intended for use with co-located teams where all team members could get together every day in stand-up meetings. However, much software development now involves distributed teams, with team members located in different places around the world. This allows companies to take advantage

Figure 3.10 Distributed Scrum

of lower cost staff in other countries, makes access to specialist skills possible, and allows for 24-hour development, with work going on in different time zones.

Consequently, there have been developments of Scrum for distributed development environments and multi-team working. Typically, for offshore development, the product owner is in a different country from the development team, which may also be distributed. Figure 3.10 shows the requirements for Distributed Scrum (Deemer 2011).

3.4 Scaling agile methods

Agile methods were developed for use by small programming teams that could work together in the same room and communicate informally. They were originally used by for the development of small and medium-sized systems and software products. Small companies, without formal processes or bureaucracy, were enthusiastic initial adopters of these methods.

Of course, the need for faster delivery of software, which is more suited to customer needs, also applies to both larger systems and larger companies. Consequently, over the past few years, a lot of work has been put into evolving agile methods for both large software systems and for use in large companies.

Scaling agile methods has closely related facets:

1. Scaling up these methods to handle the development of large systems that are too big to be developed by a single small team.
2. Scaling out these methods from specialized development teams to more widespread use in a large company that has many years of software development experience.

Of course, scaling up and scaling out are closely related. Contracts to develop large software systems are usually awarded to large organizations, with multiple teams working on the development project. These large companies have often experimented with agile methods in smaller projects, so they face the problems of scaling up and scaling out at the same time.

There are many anecdotes about the effectiveness of agile methods, and it has been suggested that these can lead to orders of magnitude improvements in productivity and comparable reductions in defects. Ambler (Ambler 2010), an influential agile method developer, suggests that these productivity improvements are exaggerated for large systems and organizations. He suggests that an organization moving to agile methods can expect to see productivity improvement across the organization of about 15% over 3 years, with similar reductions in the number of product defects.

3.4.1 Practical problems with agile methods

In some areas, particularly in the development of software products and apps, agile development has been incredibly successful. It is by far the best approach to use for this type of system. However, agile methods may not be suitable for other types of software development, such as embedded systems engineering or the development of large and complex systems.

For large, long-lifetime systems that are developed by a software company for an external client, using an agile approach presents a number of problems.

1. The informality of agile development is incompatible with the legal approach to contract definition that is commonly used in large companies.
2. Agile methods are most appropriate for new software development rather than for software maintenance. Yet the majority of software costs in large companies come from maintaining their existing software systems.
3. Agile methods are designed for small co-located teams, yet much software development now involves worldwide distributed teams.

Contractual issues can be a major problem when agile methods are used. When the system customer uses an outside organization for system development, a contract for the software development is drawn up between them. The software requirements document is usually part of that contract between the customer and the supplier. Because the interleaved development of requirements and code is fundamental to agile methods, there is no definitive statement of requirements that can be included in the contract.

Consequently, agile methods have to rely on contracts in which the customer pays for the time required for system development rather than the development of a specific set of requirements. As long as all goes well, this benefits both the customer and the developer. However, if problems arise, then there may be difficult disputes over who is to blame and who should pay for the extra time and resources required to resolve the problems.

As I explain in Chapter 9, a huge amount of software engineering effort goes into the maintenance and evolution of existing software systems. Agile practices, such as incremental delivery, design for change, and maintaining simplicity all make sense when software is being changed. In fact, you can think of an agile development process as a process that supports continual change. If agile methods are used for software product development, new releases of the product or app simply involve continuing the agile approach.

However, where maintenance involves a custom system that must be changed in response to new business requirements, there is no clear consensus on the suitability of agile methods for software maintenance (Bird 2011; Kilner 2012). Three types of problems can arise:

- lack of product documentation
- keeping customers involved
- development team continuity

Formal documentation is supposed to describe the system and so make it easier for people changing the system to understand. In practice, however, formal documentation is rarely updated and so does not accurately reflect the program code. For this reason, agile methods enthusiasts argue that it is a waste of time to write this documentation and that the key to implementing maintainable software is to produce high-quality, readable code. The lack of documentation should not be a problem in maintaining systems developed using an agile approach.

However, my experience of system maintenance is that the most important document is the system requirements document, which tells the software engineer what the system is supposed to do. Without such knowledge, it is difficult to assess the impact of proposed system changes. Many agile methods collect requirements informally and incrementally and do not create a coherent requirements document. The use of agile methods may therefore make subsequent system maintenance more difficult and expensive. This is a particular problem if development team continuity cannot be maintained.

A key challenge in using an agile approach to maintenance is keeping customers involved in the process. While a customer may be able to justify the full-time involvement of a representative during system development, this is less likely during maintenance where changes are not continuous. Customer representatives are likely to lose interest in the system. Therefore, it is likely that alternative mechanisms, such as change proposals, discussed in Chapter 25, will have to be adapted to fit in with an agile approach.

Another potential problem that may arise is maintaining continuity of the development team. Agile methods rely on team members understanding aspects of the system without having to consult documentation. If an agile development team is broken up, then this implicit knowledge is lost and it is difficult for new team members to build up the same understanding of the system and its components. Many programmers prefer to work on new development to software maintenance, and so they are unwilling to continue to work on a software system after the first release has been delivered. Therefore, even when the intention is to keep the development team together, people leave if they are assigned maintenance tasks.

Principle	Practice
Customer involvement	This depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Often, customer representatives have other demands on their time and cannot play a full part in the software development. Where there are external stakeholders, such as regulators, it is difficult to represent their views to the agile team.
Embrace change	Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.
Incremental delivery	Rapid iterations and short-term planning for development does not always fit in with the longer-term planning cycles of business planning and marketing. Marketing managers may need to know product features several months in advance to prepare an effective marketing campaign.
Maintain simplicity	Under pressure from delivery schedules, team members may not have time to carry out desirable system simplifications.
People, not process	Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods and therefore may not interact well with other team members.

Figure 3.11 Agile principles and organizational practice

3.4.2 Agile and plan-driven methods

A fundamental requirement of scaling agile methods is to integrate them with plan-driven approaches. Small startup companies can work with informal and short-term planning, but larger companies have to have longer-term plans and budgets for investment, staffing, and business development. Their software development must support these plans, so longer-term software planning is essential.

Early adopters of agile methods in the first decade of the 21st century were enthusiasts and deeply committed to the agile manifesto. They deliberately rejected the plan-driven approach to software engineering and were reluctant to change the initial vision of agile methods in any way. However, as organizations saw the value and benefits of an agile approach, they adapted these methods to suit their own culture and ways of working. They had to do this because the principles underlying agile methods are sometimes difficult to realize in practice (Figure 3.11).

To address these problems, most large "agile" software development projects combine practices from plan-driven and agile approaches. Some are mostly agile, and others are mostly plan-driven but with some agile practices. To decide on the balance between a plan-based and an agile approach, you have to answer a range of technical, human and organizational questions. These relate to the system being developed, the development team, and the organizations that are developing and procuring the system (Figure 3.12).

Agile methods were developed and refined in projects to develop small to medium-sized business systems and software products, where the software developer controls the specification of the system. Other types of system have attributes such as size, complexity, real-time response, and external regulation that mean a "pure" agile approach is

Figure 3.12 Factors influencing the choice of plan-based or agile development

unlikely to work. There needs to be some up-front planning, design, and documentation in the systems engineering process. Some of the key issues are as follows:

1. How large is the system that is being developed? Agile methods are most effective when the system can be developed with a relatively small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams, so a plan-driven approach may have to be used.
2. What type of system is being developed? Systems that require a lot of analysis before implementation (e.g., real-time system with complex timing requirements) usually need a fairly detailed design to carry out this analysis. A plan-driven approach may be best in those circumstances.
3. What is the expected system lifetime? Long-lifetime systems may require more design documentation to communicate the original intentions of the system developers to the support team. However, supporters of agile methods rightly argue that documentation is frequently not kept up to date and is not of much use for long-term system maintenance.
4. Is the system subject to external regulation? If a system has to be approved by an external regulator (e.g., the Federal Aviation Administration approves software that is critical to the operation of an aircraft), then you will probably be required to produce detailed documentation as part of the system safety case.

Agile methods place a great deal of responsibility on the development team to cooperate and communicate during the development of the system. They rely on individual engineering skills and software support for the development process. However, in reality, not everyone is a highly skilled engineer, people do not communicate effectively, and it is not always possible for teams to work together. Some planning may be required to make the most effective use of the people available. Key issues are:

1. How good are the designers and programmers in the development team? It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code. If you have a team with relatively low skill levels, you may need to use the best people to develop the design, with others responsible for programming.

2. How is the development team organized? If the development team is distributed or if part of the development is being outsourced, then you may need to develop design documents to communicate across the development teams.
3. What technologies are available to support system development? Agile methods often rely on good tools to keep track of an evolving design. If you are developing a system using an IDE that does not have good tools for program visualization and analysis, then more design documentation may be required.

Television and films have created a popular vision of software companies as informal organizations run by young men (mostly) who provide a fashionable working environment, with a minimum of bureaucracy and organizational procedures. This is far from the truth. Most software is developed in large companies that have established their own working practices and procedures. Management in these companies may be uncomfortable with the lack of documentation and the informal decision making in agile methods. Key issues are:

1. Is it important to have a very detailed specification and design before moving to implementation, perhaps for contractual reasons? If so, you probably need to use a plan-driven approach for requirements engineering but may use agile development practices during system implementation.
2. Is an incremental delivery strategy, where you deliver the software to customers or other system stakeholders and get rapid feedback from them, realistic? Will customer representatives be available, and are they willing to participate in the development team?
3. Are there cultural issues that may affect system development? Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering. This usually requires extensive design documentation rather than the informal knowledge used in agile processes.

In reality, the issue of whether a project can be labeled as plan-driven or agile is not very important. Ultimately, the primary concern of buyers of a software system is whether or not they have an executable software system that meets their needs and does useful things for the individual user or the organization. Software developers should be pragmatic and should choose those methods that are most effective for the type of system being developed, whether or not these are labeled agile or plan-driven.

3.4.3 Agile methods for large systems

Agile methods have to evolve to be used for large-scale software development. The fundamental reason for this is that large-scale software systems are much more complex and difficult to understand and manage than small-scale systems or software products. Six principal factors (Figure 3.13) contribute to this complexity:

Figure 3.13 Large project characteristics

1. Large systems are usually systems of systems—collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones. It is practically impossible for each team to have a view of the whole system. Consequently, their priorities are usually to complete their part of the system without regard for wider systems issues.

2. Large systems are brownfield systems (Hopkins and Jenkins 2008); that is, they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development. Political issues can also be significant here—often the easiest solution to a problem is to change an existing system. However, this requires negotiation with the managers of that system to convince them that the changes can be implemented without risk to the system's operation.

3. Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development. This is not necessarily compatible with incremental development and frequent system integration.

4. Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed, that require certain types of system documentation to be produced, and so on. Customers may have specific compliance requirements that may have to be followed, and these may require process documentation to be completed.

5. Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.

6. Large systems usually have a diverse set of stakeholders with different perspectives and objectives. For example, nurses and administrators may be the end-users of a medical system, but senior medical staff, hospital managers, and others, are also stakeholders in the system. It is practically impossible to involve all of these different stakeholders in the development process.

Figure 3.14 IBM's Agility at Scale model (© IBM 2010)

Dean Leffingwell, who has a great deal of experience in scaling agile methods, has developed the Scaled Agile Framework (Leffingwell 2007, 2011) to support large-scale, multi-team software development. He reports how this method has been used successfully in a number of large companies. IBM has also developed a framework for the large-scale use of agile methods called the Agile Scaling Model (ASM). Figure 3.14, taken from Ambler's white paper that discusses ASM (Ambler 2010), shows an overview of this model.

The ASM recognizes that scaling is a staged process where development teams move from the core agile practices discussed here to what is called Disciplined Agile Delivery. Essentially, this stage involves adapting these practices to a disciplined organizational setting and recognizing that teams cannot simply focus on development but must also take into account other stages of the software engineering process, such as requirements and architectural design.

The final scaling stage in ASM is to move to Agility at Scale where the complexity that is inherent in large projects is recognized. This involves taking account of factors such as distributed development, complex legacy environments, and regulatory compliance requirements. The practices used for disciplined agile delivery may have to be modified on a project-by-project basis to take these into account and, sometimes, additional plan-based practices added to the process.

No single model is appropriate for all large-scale agile products as the type of product, the customer requirements, and the people available are all different. However, approaches to scaling agile methods have a number of things in common:

1. A completely incremental approach to requirements engineering is impossible. Some early work on initial software requirements is essential. You need this work to identify the different parts of the system that may be developed by different teams and, often, to be part of the contract for the system development. However, these requirements should not normally be specified in detail; details are best developed incrementally.
2. There cannot be a single product owner or customer representative. Different people have to be involved for different parts of the system, and they have to continuously communicate and negotiate throughout the development process.
3. It is not possible to focus only on the code of the system. You need to do more up-front design and system documentation. The software architecture has to be designed, and there has to be documentation produced to describe critical aspects of the system, such as database schemas and the work breakdown across teams.
4. Cross-team communication mechanisms have to be designed and used. This should involve regular phone and videoconferences between team members and frequent, short electronic meetings where teams update each other on progress. A range of communication channels such as email, instant messaging, wikis, and social networking systems should be provided to facilitate communications.
5. Continuous integration, where the whole system is built every time any developer checks in a change, is practically impossible when several separate programs have to be integrated to create the system. However, it is essential to maintain frequent system builds and regular releases of the system. Configuration management tools that support multi-team software development are essential.

Scrum has been adapted for large-scale development. In essence, the Scrum team model described in Section 3.3 is maintained, but multiple Scrum teams are set up. The key characteristics of multi-team Scrum are:

1. *Role replication* Each team has a Product Owner for its work component and ScrumMaster. There may be a chief Product Owner and ScrumMaster for the entire project.
2. *Product architects* Each team chooses a product architect, and these architects collaborate to design and evolve the overall system architecture.
3. *Release alignment* The dates of product releases from each team are aligned so that a demonstrable and complete system is produced.
4. *Scrum of Scrums* There is a daily Scrum of Scrums where representatives from each team meet to discuss progress, identify problems, and plan the work to be done that day. Individual team Scrums may be staggered in time so that representatives from other teams can attend if necessary.

3.4.4 Agile methods across organizations

Small software companies that develop software products have been among the most enthusiastic adopters of agile methods. These companies are not constrained by organizational bureaucracies or process standards, and they can change quickly to adopt new ideas. Of course, larger companies have also experimented with agile methods in specific projects, but it is much more difficult for them to "scale out" these methods across the organization.

It can be difficult to introduce agile methods into large companies for a number of reasons:

1. Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach, as they do not know how this will affect their particular projects.
2. Large organizations often have quality procedures and standards that all projects are expected to follow, and, because of their bureaucratic nature, these are likely to be incompatible with agile methods. Sometimes, these are supported by software tools (e.g., requirements management tools), and the use of these tools is mandated for all projects.
3. Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities, and people with lower skill levels may not be effective team members in agile processes.
4. There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

Change management and testing procedures are examples of company procedures that may not be compatible with agile methods. Change management is the process of controlling changes to a system, so that the impact of changes is predictable and costs are controlled. All changes have to be approved in advance before they are made, and this conflicts with the notion of refactoring. When refactoring is part of an agile process, any developer can improve any code without getting external approval. For large systems, there are also testing standards where a system build is handed over to an external testing team. This may conflict with test-first approaches used in agile development methods.

Introducing and sustaining the use of agile methods across a large organization is a process of cultural change. Cultural change takes a long time to implement and often requires a change of management before it can be accomplished. Companies wishing to use agile methods need evangelists to promote change. Rather than trying to force agile methods onto unwilling developers, companies have found that the best way to introduce agile is bit by bit, starting with an enthusiastic group of developers. A successful agile project can act as a starting point, with the project team spreading agile practice across the organization. Once the notion of agile is widely known, explicit actions can then be taken to spread it across the organization.

KEY POINTS

- Agile methods are iterative development methods that focus on reducing process overheads and documentation and on incremental software delivery. They involve customer representatives directly in the development process.
- The decision on whether to use an agile or a plan-driven approach to development should depend on the type of software being developed, the capabilities of the development team, and the culture of the company developing the system. In practice, a mix of agile and plan-based techniques may be used.
- Agile development practices include requirements expressed as user stories, pair programming, refactoring, continuous integration, and test-first development.
- Scrum is an agile method that provides a framework for organizing agile projects. It is centered around a set of sprints, which are fixed time periods when a system increment is developed. Planning is based on prioritizing a backlog of work and selecting the highest priority tasks for a sprint.
- To scale agile methods, some plan-based practices have to be integrated with agile practice. These include up-front requirements, multiple customer representatives, more documentation, common tooling across project teams, and the alignment of releases across teams.

FURTHER READING

"Get Ready for Agile Methods, With Care." A thoughtful critique of agile methods that discusses their strengths and weaknesses, written by a vastly experienced software engineer. Still very relevant, although almost 15 years old. (B. Boehm, *IEEE Computer*, January 2002) http://dx.doi.org/10.1109/2.976920

Extreme Programming Explained. This was the first book on XP and is still, perhaps, the most readable. It explains the approach from the perspective of one of its inventors, and his enthusiasm comes through very clearly in the book. (K. Beck and C. Andres, Addison-Wesley, 2004) *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. This is a comprehensive and readable description of the 2011 development of the Scrum method (K.S. Rubin, Addison-Wesley, 2013).

"Agility at Scale: Economic Governance, Measured Improvement and Disciplined Delivery." This paper discusses IBM's approach to scale agile methods, where they have a systematic approach to integrating plan-based and agile development. It is an excellent and thoughtful discussion of the key issues in scaling agile (A.W. Brown, S.W. Ambler, and W. Royce, *Proc. 35th Int. Conf. on Software Engineering*, 2013) http://dx.doi.org/10.1145/12944.12948

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/agile-methods/

EXERCISES

3.1. At the end of their study program, students in a software engineering course are typically expected to complete a major project. Explain how the agile methodology may be very useful for the students to use in this case.

3.2. Explain how the principles underlying agile methods lead to the accelerated development and deployment of software.

3.3. Extreme programming expresses user requirements as stories, with each story written on a card. Discuss the advantages and disadvantages of this approach to requirements description.

3.4. In test-first development, tests are written before the code. Explain how the test suite may compromise the quality of the software system being developed.

3.5. Suggest four reasons why the productivity rate of programmers working as a pair might be more than half that of two programmers working individually.

3.6. Compare and contrast the Scrum approach to project management with conventional plan-based approaches as discussed in Chapter 23. Your comparison should be based on the effectiveness of each approach for planning the allocation of people to projects, estimating the cost of projects, maintaining team cohesion, and managing changes in project team membership.

3.7. To reduce costs and the environmental impact of commuting, your company decides to close a number of offices and to provide support for staff to work from home. However, the senior management who introduce the policy are unaware that software is developed using Scrum. Explain how you could use technology to support Scrum in a distributed environment to make this possible. What problems are you likely to encounter using this approach?

3.8. Why is it necessary to introduce some methods and documentation from plan-based approaches when scaling agile methods to larger projects that are developed by distributed development teams?

3.9. Explain why agile methods may not work well in organizations that have teams with a wide range of skills and abilities and well-established processes.

3.10. One of the problems of having a user closely involved with a software development team is that they "go native." That is, they adopt the outlook of the development team and lose sight of the needs of their user colleagues. Suggest three ways how you might avoid this problem, and discuss the advantages and disadvantages of each approach.

REFERENCES

Ambler, S. W. 2010. "Scaling Agile: A Executive Guide." http://www.ibm.com/developerworks/community/blogs/ambler/entry/scaling_agile_an_executive_guide10/

Arisholm, E., H. Gallis, T. Dyba, and D. I. K. Sjoberg. 2007. "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise." *IEEE Trans. on Software Eng.* 33 (2): 65–86. doi:10.1109/TSE.2007.17.

Beck, K. 1998. "Chrysler Goes to 'Extremes.'" *Distributed Computing* (10): 24–28.

———. 1999. "Embracing Change with Extreme Programming." *IEEE Computer* 32 (10): 70–78. doi:10.1109/2.796139.

Bellouiti, S. 2009. "How Scrum Helped Our A-Team." http://www.scrumalliance.org/community/articles/2009/2009-june/how-scrum-helped-our team

Bird, J. 2011. "You Can't Be Agile in Maintenance." http://swreflections.blogspot.co.uk/2011/10/you-cant-be-agile-in-maintenance.html

Deemer, P. 2011. "The Distributed Scrum Primer." http://www.goodagile.com/distributedscrumprimer/.

Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley.

Hopkins, R., and K. Jenkins. 2008. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston: IBM Press.

Jeffries, R., and G. Melnik. 2007. "TDD: The Art of Fearless Programming." *IEEE Software* 24: 24–30. doi:10.1109/MS.2007.75.

Kilner, S. 2012. "Can Agile Methods Work for Software Maintenance." http://www.vlegaci.com/can-agile-methods-work-for-software-maintenance-part-1/

Larman, C., and V. R. Basili. 2003. "Iterative and Incremental Development: A Brief History." *IEEE Computer* 36 (6): 47–56. doi:10.1109/MC.2003.1204375.

Leffingwell, D. 2007. *Scaling Software Agility: Best Practices for Large Enterprises*. Boston: Addison-Wesley.

Leffingwell, D. 2011. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs and the Enterprise*. Boston: Addison-Wesley.

Mulder, M., and M. van Vliet. 2008. "Case Study: Distributed Scrum Project for Dutch Railways." *InfoQ*. http://www.infoq.com/articles/dutch-railway-scrum

Rubin, K. S. 2013. *Essential Scrum*. Boston: Addison-Wesley.

Schatz, B., and I. Abdelshafi. 2005. "Primavera Gets Agile: A Successful Transition to Agile Development." *IEEE Software* 22 (3): 36–42. doi:10.1109/MS.2005.74.

Schwaber, K., and M. Beedle. 2001. *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice-Hall.

Stapleton, J. 2003. *DSDM: Business Focused Development, 2nd ed*. Harlow, UK: Pearson Education.

Tahchiev, P., F. Leme, V. Massol, and G. Gregory. 2010. *JUnit in Action, 2/e*. Greenwich, CT: Manning Publications.

Weinberg, G. 1971. *The Psychology of Computer Programming*. New York: Van Nostrand.

Williams, L., R. R. Kessler, W. Cunningham, and R. Jeffries. 2000. "Strengthening the Case for Pair Programming." *IEEE Software* 17 (4): 19–25. doi:10.1109/52.854064.

4

Requirements engineering

Objectives

The objective of this chapter is to introduce software requirements and to explain the processes involved in discovering and documenting these requirements. When you have read the chapter, you will:

- understand the concepts of user and system requirements and why these requirements should be written in different ways;
- understand the differences between functional and non-functional software requirements;
- understand the main requirements engineering activities of elicitation, analysis, and validation, and the relationships between these activities;
- understand why requirements management is necessary and how it supports other requirements engineering activities.

Contents

4.1 Functional and non-functional requirements
4.2 Requirements engineering processes
4.3 Requirements elicitation
4.4 Requirements specification
4.5 Requirements validation
4.6 Requirements change

The requirements for a system are the descriptions of the services that a system should provide and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analyzing, documenting and checking these services and constraints is called requirements engineering (RE).

The term *requirement* is not used consistently in the software industry. In some cases, a requirement is simply a high-level, abstract statement of a service that a system should provide or a constraint on a system. At the other extreme, it is a detailed, formal definition of a system function. Davis (Davis 1993) explains why these differences exist:

> *If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not predefined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization's needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system*[†].

Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between these different levels of description. I distinguish between them by using the term *user requirements* to mean the high-level abstract requirements and *system requirements* to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

1. User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate. The user requirements may vary from broad statements of the system features required to detailed, precise descriptions of the system functionality.
2. System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

Different kinds of requirement are needed to communicate information about a system to different types of reader. Figure 4.1 illustrates the distinction between user and system requirements. This example from the mental health care patient information system (Mentcare) shows how a user requirement may be expanded into several system requirements. You can see from Figure 4.1 that the user requirement is quite

[†]Davis, A. M. 1993. Software Requirements: Objects, Functions and States. Englewood Cliffs, NJ: Prentice-Hall.

User requirements definition

1. The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g. 10mg, 20mg, etc.) separate reports shall be created for each dose unit.
1.5 Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

Figure 4.1 User and system requirements

general. The system requirements provide more specific information about the services and functions of the system that is to be implemented.

You need to write requirements at different levels of detail because different types of readers use them in different ways. Figure 4.2 shows the types of readers of the user and system requirements. The readers of the user requirements are not usually concerned with how the system will be implemented and may be managers who are not interested in the detailed facilities of the system. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation.

The different types of document readers shown in Figure 4.2 are examples of system stakeholders. As well as users, many other people have some kind of interest in the system. System stakeholders include anyone who is affected by the system in some way and so anyone who has a legitimate interest in it. Stakeholders range from end-users of a system through managers to external stakeholders such as regulators,

Figure 4.2 Readers of different types of requirements specification

Feasibility studies

A feasibility study is a short, focused study that should take place early in the RE process. It should answer three key questions: (1) Does the system contribute to the overall objectives of the organization? (2) Can the system be implemented within schedule and budget using current technology? and (3) Can the system be integrated with other systems that are used?

If the answer to any of these questions is no, you should probably not go ahead with the project.

http://software-engineering-book.com/web/feasibility-study/

who certify the acceptability of the system. For example, system stakeholders for the Mentcare system include:

1. Patients whose information is recorded in the system and relatives of these patients.
2. Doctors who are responsible for assessing and treating patients.
3. Nurses who coordinate the consultations with doctors and administer some treatments.
4. Medical receptionists who manage patients' appointments.
5. IT staff who are responsible for installing and maintaining the system.
6. A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
7. Health care managers who obtain management information from the system.
8. Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

Requirements engineering is usually presented as the first stage of the software engineering process. However, some understanding of the system requirements may have to be developed before a decision is made to go ahead with the procurement or development of a system. This early-stage RE establishes a high-level view of what the system might do and the benefits that it might provide. These may then be considered in a feasibility study, which tries to assess whether or not the system is technically and financially feasible. The results of that study help management decide whether or not to go ahead with the procurement or development of the system.

In this chapter, I present a "traditional" view of requirements rather than requirements in agile processes, which I discussed in Chapter 3. For the majority of large systems, it is still the case that there is a clearly identifiable requirements engineering phase before implementation of the system begins. The outcome is a requirements document, which may be part of the system development contract. Of course, subsequent changes are made to the requirements, and user requirements may be expanded into

more detailed system requirements. Sometimes an agile approach of concurrently eliciting the requirements as the system is developed may be used to add detail and to refine the user requirements.

4.1 Functional and non-functional requirements

Software system requirements are often classified as functional or non-functional requirements:

1. *Functional requirements* These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.
2. *Non-functional requirements* These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole rather than individual system features or services.

In reality, the distinction between different types of requirements is not as clear-cut as these simple definitions suggest. A user requirement concerned with security, such as a statement limiting access to authorized users, may appear to be a non-functional requirement. However, when developed in more detail, this requirement may generate other requirements that are clearly functional, such as the need to include user authentication facilities in the system.

This shows that requirements are not independent and that one requirement often generates or constrains other requirements. The system requirements therefore do not just specify the services or the features of the system that are required; they also specify the necessary functionality to ensure that these services/features are delivered effectively.

4.1.1 Functional requirements

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user requirements, functional requirements should be written in natural language so that system users and managers can understand them. Functional system requirements expand the user requirements and are written for system developers. They should describe the system functions, their inputs and outputs, and exceptions in detail.

Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems. For example, here are examples of functional

Domain requirements

Domain requirements are derived from the application domain of the system rather than from the specific needs of system users. They may be new functional requirements in their own right, constrain existing functional requirements, or set out how particular computations must be carried out.

The problem with domain requirements is that software engineers may not understand the characteristics of the domain in which the system operates. This means that these engineers may not know whether or not a domain requirement has been missed out or conflicts with other requirements.

http://software-engineering-book.com/web/domain-requirements/

requirements for the Mentcare system, used to maintain information about patients receiving treatment for mental health problems:

1. A user shall be able to search the appointments lists for all clinics.
2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

These user requirements define specific functionality that should be included in the system. The requirements show that functional requirements may be written at different levels of detail (contrast requirements 1 and 3).

Functional requirements, as the name suggests, have traditionally focused on what the system should do. However, if an organization decides that an existing off-the-shelf system software product can meet its needs, then there is very little point in developing a detailed functional specification. In such cases, the focus should be on the development of information requirements that specify the information needed for people to do their work. Information requirements specify the information needed and how it is to be delivered and organized. Therefore, an information requirement for the Mentcare system might specify what information is to be included in the list of patients expected for appointments that day.

Imprecision in the requirements specification can lead to disputes between customers and software developers. It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs.

For example, the first Mentcare system requirement in the above list states that a user shall be able to search the appointments lists for all clinics. The rationale for this requirement is that patients with mental health problems are sometimes confused. They may have an appointment at one clinic but actually go to a different clinic. If they have an appointment, they will be recorded as having attended, regardless of the clinic.

A medical staff member specifying a search requirement may expect "search" to mean that, given a patient name, the system looks for that name in all appointments at all clinics. However, this is not explicit in the requirement. System developers may interpret the requirement so that it is easier to implement. Their search function may require the user to choose a clinic and then carry out the search of the patients who attended that clinic. This involves more user input and so takes longer to complete the search.

Ideally, the functional requirements specification of a system should be both complete and consistent. Completeness means that all services and information required by the user should be defined. Consistency means that requirements should not be contradictory.

In practice, it is only possible to achieve requirements consistency and completeness for very small software systems. One reason is that it is easy to make mistakes and omissions when writing specifications for large, complex systems. Another reason is that large systems have many stakeholders, with different backgrounds and expectations. Stakeholders are likely to have different—and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are originally specified, and the inconsistent requirements may only be discovered after deeper analysis or during system development.

4.1.2 Non-functional requirements

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users. These non-functional requirements usually specify or constrain characteristics of the system as a whole. They may relate to emergent system properties such as reliability, response time, and memory use. Alternatively, they may define constraints on the system implementation, such as the capabilities of I/O devices or the data representations used in interfaces with other systems.

Non-functional requirements are often more critical than individual functional requirements. System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly.

While it is often possible to identify which system components implement specific functional requirements (e.g., there may be formatting components that implement reporting requirements), this is often more difficult with non-functional requirements. The implementation of these requirements may be spread throughout the system, for two reasons:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance requirements are met in an embedded system, you may have to organize the system to minimize communications between components.

Figure 4.3 Types of non-functional requirements

2. An individual non-functional requirement, such as a security requirement, may generate several, related functional requirements that define new system services that are required if the non-functional requirement is to be implemented. In addition, it may also generate requirements that constrain existing requirements; for example, it may limit access to information in the system.

Nonfunctional requirements arise through user needs because of budget constraints, organizational policies, the need for interoperability with other software or hardware systems, or external factors such as safety regulations or privacy legislation. Figure 4.3 is a classification of non-functional requirements. You can see from this diagram that the non-functional requirements may come from required characteristics of the software (product requirements), the organization developing the software (organizational requirements), or external sources:

1. *Product requirements* These requirements specify or constrain the runtime behavior of the software. Examples include performance requirements for how fast the system must execute and how much memory it requires; reliability requirements that set out the acceptable failure rate; security requirements; and usability requirements.

2. *Organizational requirements* These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organizations. Examples include operational process requirements that define how the system will be used; development process requirements that specify the

PRODUCT REQUIREMENT
The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 08:30–17:30). Downtime within normal working hours shall not exceed 5 seconds in any one day.

ORGANIZATIONAL REQUIREMENT
Users of the Mentcare system shall identify themselves using their health authority identity card.

EXTERNAL REQUIREMENT
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Figure 4.4 Examples of possible non-functional requirements for the Mentcare system

programming language; the development environment or process standards to be used; and environmental requirements that specify the operating environment of the system.

3. *External requirements* This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a nuclear safety authority; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will be acceptable to its users and the general public.

Figure 4.4 shows examples of product, organizational, and external requirements that could be included in the Mentcare system specification. The product requirement is an availability requirement that defines when the system has to be available and the allowed downtime each day. It says nothing about the functionality of the Mentcare system and clearly identifies a constraint that has to be considered by the system designers.

The organizational requirement specifies how users authenticate themselves to the system. The health authority that operates the system is moving to a standard authentication procedure for all software where, instead of users having a login name, they swipe their identity card through a reader to identify themselves. The external requirement is derived from the need for the system to conform to privacy legislation. Privacy is obviously a very important issue in health care systems, and the requirement specifies that the system should be developed in accordance with a national privacy standard.

A common problem with non-functional requirements is that stakeholders propose requirements as general goals, such as ease of use, the ability of the system to recover from failure, or rapid user response. Goals set out good intentions but cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered. For example, the following system goal is typical of how a manager might express usability requirements:

The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized.

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Megabytes/Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Figure 4.5 Metrics for specifying non-functional requirements

I have rewritten this to show how the goal could be expressed as a "testable" non-functional requirement. It is impossible to objectively verify the system goal, but in the following description you can at least include software instrumentation to count the errors made by users when they are testing the system.

Medical staff shall be able to use all the system functions after two hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.

Whenever possible, you should write non-functional requirements quantitatively so that they can be objectively tested. Figure 4.5 shows metrics that you can use to specify non-functional system properties. You can measure these characteristics when the system is being tested to check whether or not the system has met its non-functional requirements.

In practice, customers for a system often find it difficult to translate their goals into measurable requirements. For some goals, such as maintainability, there are no simple metrics that can be used. In other cases, even when quantitative specification is possible, customers may not be able to relate their needs to these specifications. They don't understand what some number defining the reliability (for example) means in terms of their everyday experience with computer systems. Furthermore, the cost of objectively verifying measurable, non-functional requirements can be very high, and the customers paying for the system may not think these costs are justified.

Non-functional requirements often conflict and interact with other functional or non-functional requirements. For example, the identification requirement in Figure 4.4 requires a card reader to be installed with each computer that connects to the system. However, there may be another requirement that requests mobile access to the system from doctors' or nurses' tablets or smartphones. These are not normally

equipped with card readers so, in these circumstances, some alternative identification method may have to be supported.

It is difficult to separate functional and non-functional requirements in the requirements document. If the non-functional requirements are stated separately from the functional requirements, the relationships between them may be hard to understand. However, you should, ideally, highlight requirements that are clearly related to emergent system properties, such as performance or reliability. You can do this by putting them in a separate section of the requirements document or by distinguishing them, in some way, from other system requirements.

Non-functional requirements such as reliability, safety, and confidentiality requirements are particularly important for critical systems. I cover these dependability requirements in Part 2, which describes ways of specifying reliability, safety, and security requirements.

4.2 Requirements engineering processes

As I discussed in Chapter 2, requirements engineering involves three key activities. These are discovering requirements by interacting with stakeholders (elicitation and analysis); converting these requirements into a standard form (specification); and checking that the requirements actually define the system that the customer wants (validation). I have shown these as sequential processes in Figure 2.4. However, in practice, requirements engineering is an iterative process in which the activities are interleaved.

Figure 4.6 shows this interleaving. The activities are organized as an iterative process around a spiral. The output of the RE process is a system requirements document. The amount of time and effort devoted to each activity in an iteration depends on the stage of the overall process, the type of system being developed, and the budget that is available.

Early in the process, most effort will be spent on understanding high-level business and non-functional requirements, and the user requirements for the system. Later in the process, in the outer rings of the spiral, more effort will be devoted to eliciting and understanding the non-functional requirements and more detailed system requirements.

This spiral model accommodates approaches to development where the requirements are developed to different levels of detail. The number of iterations around the spiral can vary so that the spiral can be exited after some or all of the user requirements have been elicited. Agile development can be used instead of prototyping so that the requirements and the system implementation are developed together.

In virtually all systems, requirements change. The people involved develop a better understanding of what they want the software to do; the organization buying the system changes; and modifications are made to the system's hardware, software, and organizational environment. Changes have to be managed to understand the impact on other requirements and the cost and system implications of making the change. I discuss this process of requirements management in Section 4.6.

Figure 4.6 A spiral view of the requirements engineering process

4.3 Requirements elicitation

The aims of the requirements elicitation process are to understand the work that stakeholders do and how they might use a new system to help support that work. During requirements elicitation, software engineers work with stakeholders to find out about the application domain, work activities, the services and system features that stakeholders want, the required performance of the system, hardware constraints, and so on.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

1. Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.

Figure 4.7 The requirements elicitation and analysis process

2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.

3. Different stakeholders, with diverse requirements, may express their requirements in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.

4. Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.

5. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

A process model of the elicitation and analysis process is shown in Figure 4.7. Each organization will have its own version or instantiation of this general model, depending on local factors such as the expertise of the staff, the type of system being developed, and the standards used.

The process activities are:

1. *Requirements discovery and understanding* This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.

2. *Requirements classification and organization* This activity takes the unstructured collection of requirements, groups related requirements and organizes them into coherent clusters.

3. *Requirements prioritization and negotiation* Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts

Viewpoints

A viewpoint is a way of collecting and organizing a set of requirements from a group of stakeholders who have something in common. Each viewpoint therefore includes a set of system requirements. Viewpoints might come from end-users, managers, or others. They help identify the people who can provide information about their requirements and structure the requirements for analysis.

http://www.software-engineering-book.com/web/viewpoints/

through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.

4. *Requirements documentation* The requirements are documented and input into the next round of the spiral. An early draft of the software requirements documents may be produced at this stage, or the requirements may simply be maintained informally on whiteboards, wikis, or other shared spaces.

Figure 4.7 shows that requirements elicitation and analysis is an iterative process with continual feedback from each activity to other activities. The process cycle starts with requirements discovery and ends with the requirements documentation. The analyst's understanding of the requirements improves with each round of the cycle. The cycle ends when the requirements document has been produced.

To simplify the analysis of requirements, it is helpful to organize and group the stakeholder information. One way of doing so is to consider each stakeholder group to be a viewpoint and to collect all requirements from that group into the viewpoint. You may also include viewpoints to represent domain requirements and constraints from other systems. Alternatively, you can use a model of the system architecture to identify subsystems and to associate requirements with each subsystem.

Inevitably, different stakeholders have different views on the importance and priority of requirements, and sometimes these views are conflicting. If some stakeholders feel that their views have not been properly considered, then they may deliberately attempt to undermine the RE process. Therefore, it is important that you organize regular stakeholder meetings. Stakeholders should have the opportunity to express their concerns and agree on requirements compromises.

At the requirements documentation stage, it is important that you use simple language and diagrams to describe the requirements. This makes it possible for stakeholders to understand and comment on these requirements. To make information sharing easier, it is best to use a shared document (e.g., on Google Docs or Office 365) or a wiki that is accessible to all interested stakeholders.

4.3.1 Requirements elicitation techniques

Requirements elicitation involves meeting with stakeholders of different kinds to discover information about the proposed system. You may supplement this information

with knowledge of existing systems and their usage and information from documents of various kinds. You need to spend time understanding how people work, what they produce, how they use other systems, and how they may need to change to accommodate a new system.

There are two fundamental approaches to requirements elicitation:

1. Interviewing, where you talk to people about what they do.
2. Observation or ethnography, where you watch people doing their job to see what artifacts they use, how they use them, and so on.

You should use a mix of interviewing and observation to collect information and, from that, you derive the requirements, which are then the basis for further discussions.

4.3.1.1 Interviewing

Formal or informal interviews with system stakeholders are part of most requirements engineering processes. In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. Requirements are derived from the answers to these questions. Interviews may be of two types:

1. Closed interviews, where the stakeholder answers a predefined set of questions.
2. Open interviews, in which there is no predefined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develops a better understanding of their needs.

In practice, interviews with stakeholders are normally a mixture of both of these. You may have to obtain the answer to certain questions, but these usually lead to other issues that are discussed in a less structured way. Completely open-ended discussions rarely work well. You usually have to ask some questions to get started and to keep the interview focused on the system to be developed.

Interviews are good for getting an overall understanding of what stakeholders do, how they might interact with the new system, and the difficulties that they face with current systems. People like talking about their work, and so they are usually happy to get involved in interviews. However, unless you have a system prototype to demonstrate, you should not expect stakeholders to suggest specific and detailed requirements. Everyone finds it difficult to visualize what a system might be like. You need to analyze the information collected and to generate the requirements from this.

Eliciting domain knowledge through interviews can be difficult, for two reasons:

1. All application specialists use jargon specific to their area of work. It is impossible for them to discuss domain requirements without using this terminology. They normally use words in a precise and subtle way that requirements engineers may misunderstand.

2. Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning. For example, for a librarian, it goes without saying that all acquisitions are catalogued before they are added to the library. However, this may not be obvious to the interviewer, and so it isn't taken into account in the requirements.

Interviews are not an effective technique for eliciting knowledge about organizational requirements and constraints because there are subtle power relationships between the different people in the organization. Published organizational structures rarely match the reality of decision making in an organization, but interviewees may not wish to reveal the actual rather than the theoretical structure to a stranger. In general, most people are generally reluctant to discuss political and organizational issues that may affect the requirements.

To be an effective interviewer, you should bear two things in mind:

1. You should be open-minded, avoid preconceived ideas about the requirements, and willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, then you should be willing to change your mind about the system.
2. You should prompt the interviewee to get discussions going by using a springboard question or a requirements proposal, or by working together on a prototype system. Saying to people "tell me what you want" is unlikely to result in useful information. They find it much easier to talk in a defined context rather than in general terms.

Information from interviews is used along with other information about the system from documentation describing business processes or existing systems, user observations, and developer experience. Sometimes, apart from the information in the system documents, the interview information may be the only source of information about the system requirements. However, interviewing on its own is liable to miss essential information, and so it should be used in conjunction with other requirements elicitation techniques.

4.3.1.2 Ethnography

Software systems do not exist in isolation. They are used in a social and organizational environment, and software system requirements may be generated or constrained by that environment. One reason why many software systems are delivered but never used is that their requirements do not take proper account of how social and organizational factors affect the practical operation of the system. It is therefore very important that, during the requirements engineering process, you try to understand the social and organizational issues that affect the use of the system.

Ethnography is an observational technique that can be used to understand operational processes and help derive requirements for software to support these processes. An analyst immerses himself or herself in the working environment where

the system will be used. The day-to-day work is observed, and notes are made of the actual tasks in which participants are involved. The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.

People often find it very difficult to articulate details of their work because it is second nature to them. They understand their own work but may not understand its relationship to other work in the organization. Social and organizational factors that affect the work, but that are not obvious to individuals, may only become clear when noticed by an unbiased observer. For example, a workgroup may self-organize so that members know of each other's work and can cover for each other if someone is absent. This may not be mentioned during an interview as the group might not see it as an integral part of their work.

Suchman (Suchman 1983) pioneered the use of ethnography to study office work. She found that actual work practices were far richer, more complex, and more dynamic than the simple models assumed by office automation systems. The difference between the assumed and the actual work was the most important reason why these office systems had no significant effect on productivity. Crabtree (Crabtree 2003) discusses a wide range of studies since then and describes, in general, the use of ethnography in systems design. In my own research, I have investigated methods of integrating ethnography into the software engineering process by linking it with requirements engineering methods (Viller and Sommerville 2000) and documenting patterns of interaction in cooperative systems (Martin and Sommerville 2004).

Ethnography is particularly effective for discovering two types of requirements:

1. Requirements derived from the way in which people actually work, rather than the way in which business process definitions say they ought to work. In practice, people never follow formal processes. For example, air traffic controllers may switch off a conflict alert system that detects aircraft with intersecting flight paths, even though normal control procedures specify that it should be used. The conflict alert system is sensitive and issues audible warnings even when planes are far apart. Controllers may find these distracting and prefer to use other strategies to ensure that planes are not on conflicting flight paths.

2. Requirements derived from cooperation and awareness of other people's activities. For example, air traffic controllers (ATCs) may use an awareness of other controlles' work to predict the number of aircraft that will be entering their control sector. They then modify their control strategies depending on that predicted workload. Therefore, an automated ATC system should allow controllers in a sector to have some visibility of the work in adjacent sectors.

Ethnography can be combined with the development of a system prototype (Figure 4.8). The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer. He or she should then look for the answers to these questions during the next phase of the system study (Sommerville et al. 1993).

Figure 4.8 Ethnography and prototyping for requirements analysis

Ethnography is helpful to understand existing systems, but this understanding does not always help with innovation. Innovation is particularly relevant for new product development. Commentators have suggested that Nokia used ethnography to discover how people used their phones and developed new phone models on that basis; Apple, on the other hand, ignored current use and revolutionized the mobile phone industry with the introduction of the iPhone.

Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques. However, because of its focus on the end-user, this approach is not effective for discovering broader organizational or domain requirements or for suggestion innovations. You therefore have to use ethnography as one of a number of techniques for requirements elicitation.

4.3.2 Stories and scenarios

People find it easier to relate to real-life examples than abstract descriptions. They are not good at telling you the system requirements. However, they may be able to describe how they handle particular situations or imagine things that they might do in a new way of working. Stories and scenarios are ways of capturing this kind of information. You can then use these when interviewing groups of stakeholders to discuss the system with other stakeholders and to develop more specific system requirements.

Stories and scenarios are essentially the same thing. They are a description of how the system can be used for some particular task. They describe what people do, what information they use and produce, and what systems they may use in this process. The difference is in the ways that descriptions are structured and in the level of detail presented. Stories are written as narrative text and present a high-level description of system use; scenarios are usually structured with specific information collected such as inputs and outputs. I find stories to be effective in setting out the "big picture." Parts of stories can then be developed in more detail and represented as scenarios.

Figure 4.9 is an example of a story that I developed to understand the requirements for the iLearn digital learning environment that I introduced in Chapter 1. This story describes a situation in a primary (elementary) school where the teacher is using the environment to support student projects on the fishing industry. You can see this is a very high-level description. Its purpose is to facilitate discussion of how the iLearn system might be used and to act as a starting point for eliciting the requirements for that system.

Photo sharing in the classroom

Jack is a primary school teacher in Ullapool (a village in northern Scotland). He has decided that a class project should be focused on the fishing industry in the area, looking at the history, development, and economic impact of fishing. As part of this project, pupils are asked to gather and share reminiscences from relatives, use newspaper archives, and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCRAN (a history resources site) to access newspaper archives and photographs. However, Jack also needs a photo-sharing site because he wants pupils to take and comment on each other's photos and to upload scans of old photographs that they may have in their families.

Jack sends an email to a primary school teachers' group, which he is a member of, to see if anyone can recommend an appropriate system. Two teachers reply, and both suggest that he use KidsTakePics, a photo-sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account. He uses the iLearn setup service to add KidsTakePics to the services seen by the pupils in his class so that when they log in, they can immediately use the system to upload photos from their mobile devices and class computers.

Figure 4.9 A user story for the iLearn system

The advantage of stories is that everyone can easily relate to them. We found this approach to be particularly useful to get information from a wider community than we could realistically interview. We made the stories available on a wiki and invited teachers and students from across the country to comment on them.

These high-level stories do not go into detail about a system, but they can be developed into more specific scenarios. Scenarios are descriptions of example user interaction sessions. I think that it is best to present scenarios in a structured way rather than as narrative text. User stories used in agile methods such as Extreme Programming, are actually narrative scenarios rather than general stories to help elicit requirements.

A scenario starts with an outline of the interaction. During the elicitation process, details are added to create a complete description of that interaction. At its most general, a scenario may include:

1. A description of what the system and users expect when the scenario starts.
2. A description of the normal flow of events in the scenario.
3. A description of what can go wrong and how resulting problems can be handled.
4. Information about other activities that might be going on at the same time.
5. A description of the system state when the scenario ends.

As an example of a scenario, Figure 4.10 describes what happens when a student uploads photos to the KidsTakePics system, as explained in Figure 4.9. The key difference between this system and other systems is that a teacher moderates the uploaded photos to check that they are suitable for sharing.

You can see this is a much more detailed description than the story in Figure 4.9, and so it can be used to propose requirements for the iLearn system. Like stories, scenarios can be used to facilitate discussions with stakeholders who sometimes may have different ways of achieving the same result.

Uploading photos to KidsTakePics

Initial assumption: A user or a group of users have one or more digital photographs to be uploaded to the picture-sharing site. These photos are saved on either a tablet or a laptop computer. They have successfully logged on to KidsTakePics.

Normal: The user chooses to upload photos and is prompted to select the photos to be uploaded on the computer and to select the project name under which the photos will be stored. Users should also be given the option of inputting keywords that should be associated with each uploaded photo. Uploaded photos are named by creating a conjunction of the user name with the filename of the photo on the local computer.

On completion of the upload, the system automatically sends an email to the project moderator, asking them to check new content, and generates an on-screen message to the user that this checking has been done.

What can go wrong: No moderator is associated with the selected project. An email is automatically generated to the school administrator asking them to nominate a project moderator. Users should be informed of a possible delay in making their photos visible.

Photos with the same name have already been uploaded by the same user. The user should be asked if he or she wishes to re-upload the photos with the same name, rename the photos, or cancel the upload. If users choose to re-upload the photos, the originals are overwritten. If they choose to rename the photos, a new name is automatically generated by adding a number to the existing filename.

Other activities: The moderator may be logged on to the system and may approve photos as they are uploaded.

System state on completion: User is logged on. The selected photos have been uploaded and assigned a status "awaiting moderation." Photos are visible to the moderator and to the user who uploaded them.

Figure 4.10 Scenario for uploading photos in KidsTakePics

4.4 Requirements specification

Requirements specification is the process of writing down the user and system requirements in a requirements document. Ideally, the user and system requirements should be clear, unambiguous, easy to understand, complete, and consistent. In practice, this is almost impossible to achieve. Stakeholders interpret the requirements in different ways, and there are often inherent conflicts and inconsistencies in the requirements.

User requirements are almost always written in natural language supplemented by appropriate diagrams and tables in the requirements document. System requirements may also be written in natural language, but other notations based on forms, graphical, or mathematical system models can also be used. Figure 4.11 summarizes possible notations for writing system requirements.

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge. Ideally, they should specify only the external behavior of the system. The requirements document should not include details of the system architecture or design. Consequently, if you are writing user requirements, you should not use software jargon, structured notations, or formal notations. You should write user requirements in natural language, with simple tables, forms, and intuitive diagrams.

Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system. UML (unified modeling language) use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want, and they are reluctant to accept it as a system contract. (I discuss this approach, in Chapter 10, which covers system dependability.)

Figure 4.11 Notations for writing system requirements

System requirements are expanded versions of the user requirements that software engineers use as the starting point for the system design. They add detail and explain how the system should provide the user requirements. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system.

Ideally, the system requirements should only describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. However, at the level of detail required to completely specify a complex software system, it is neither possible nor desirable to exclude all design information. There are several reasons for this:

1. You may have to design an initial architecture of the system to help structure the requirements specification. The system requirements are organized according to the different subsystems that make up the system. We did this when we were defining the requirements for the iLearn system, where we proposed the architecture shown in Figure 1.8.

2. In most cases, systems must interoperate with existing systems, which constrain the design and impose requirements on the new system.

3. The use of a specific architecture to satisfy non-functional requirements, such as N-version programming to achieve reliability, discussed in Chapter 11, may be necessary. An external regulator who needs to certify that the system is safe may specify that an architectural design that has already been certified should be used.

4.4.1 Natural language specification

Natural language has been used to write requirements for software since the 1950s. It is expressive, intuitive, and universal. It is also potentially vague and ambiguous, and its interpretation depends on the background of the reader. As a result, there

3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow, so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

Figure 4.12 Example requirements for the insulin pump software system

have been many proposals for alternative ways to write requirements. However, none of these proposals has been widely adopted, and natural language will continue to be the most widely used way of specifying system and software requirements.

To minimize misunderstandings when writing natural language requirements, I recommend that you follow these simple guidelines:

1. Invent a standard format and ensure that all requirement definitions adhere to that format. Standardizing the format makes omissions less likely and requirements easier to check. I suggest that, wherever possible, you should write the requirement in one or two sentences of natural language.
2. Use language consistently to distinguish between mandatory and desirable requirements. Mandatory requirements are requirements that the system must support and are usually written using "shall." Desirable requirements are not essential and are written using "should."
3. Use text highlighting (bold, italic, or color) to pick out key parts of the requirement.
4. Do not assume that readers understand technical, software engineering language. It is easy for words such as "architecture" and "module" to be misunderstood. Wherever possible, you should avoid the use of jargon, abbreviations, and acronyms.
5. Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included and who proposed the requirement (the requirement source), so that you know whom to consult if the requirement has to be changed. Requirements rationale is particularly useful when requirements are changed, as it may help decide what changes would be undesirable.

Figure 4.12 illustrates how these guidelines may be used. It includes two requirements for the embedded software for the automated insulin pump, introduced in Chapter 1. Other requirements for this embedded system are defined in the insulin pump requirements document, which can be downloaded from the book's web pages.

4.4.2 Structured specifications

Structured natural language is a way of writing system requirements where requirements are written in a standard way rather than as free-form text. This approach maintains most of the expressiveness and understandability of natural language but

Problems with using natural language for requirements specification

The flexibility of natural language, which is so useful for specification, often causes problems. There is scope for writing unclear requirements, and readers (the designers) may misinterpret requirements because they have a different background to the user. It is easy to amalgamate several requirements into a single sentence, and structuring natural language requirements can be difficult.

http://software-engineering-book.com/web/natural-language/

ensures that some uniformity is imposed on the specification. Structured language notations use templates to specify system requirements. The specification may use programming language constructs to show alternatives and iteration, and may highlight key elements using shading or different fonts.

The Robertsons (Robertson and Robertson 2013), in their book on the VOLERE requirements engineering method, recommend that user requirements be initially written on cards, one requirement per card. They suggest a number of fields on each card, such as the requirements rationale, the dependencies on other requirements, the source of the requirements, and supporting materials. This is similar to the approach used in the example of a structured specification shown in Figure 4.13.

To use a structured approach to specifying system requirements, you define one or more standard templates for requirements and represent these templates as structured forms. The specification may be structured around the objects manipulated by the system, the functions performed by the system, or the events processed by the system. An example of a form-based specification, in this case, one that defines how to calculate the dose of insulin to be delivered when the blood sugar is within a safe band, is shown in Figure 4.13.

When a standard format is used for specifying functional requirements, the following information should be included:

1. A description of the function or entity being specified.
2. A description of its inputs and the origin of these inputs.
3. A description of its outputs and the destination of these outputs.
4. Information about the information needed for the computation or other entities in the system that are required (the "requires" part).
5. A description of the action to be taken.
6. If a functional approach is used, a precondition setting out what must be true before the function is called, and a postcondition specifying what is true after the function is called.
7. A description of the side effects (if any) of the operation.

Using structured specifications removes some of the problems of natural language specification. Variability in the specification is reduced, and requirements are organized

Insulin Pump/Control Software/SRS/3.3.2

Function	Compute insulin dose: Safe sugar level.
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.
Inputs	Current sugar reading (r2), the previous two readings (r0 and r1).
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose—the dose in insulin to be delivered.
Destination	Main control loop.
Action:	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered. (see Figure 4.14)
Requires	Two previous readings so that the rate of change of sugar level can be computed.
Precondition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Postcondition	r0 is replaced by r1 then r1 is replaced by r2.
Side effects	None.

Figure 4.13 The structured specification of a requirement for an insulin pump

more effectively. However, it is still sometimes difficult to write requirements in a clear and unambiguous way, particularly when complex computations (e.g., how to calculate the insulin dose) are to be specified.

To address this problem, you can add extra information to natural language requirements, for example, by using tables or graphical models of the system. These can show how computations proceed, how the system state changes, how users interact with the system, and how sequences of actions are performed.

Tables are particularly useful when there are a number of possible alternative situations and you need to describe the actions to be taken for each of these. The insulin pump bases its computations of the insulin requirement on the rate of change of blood sugar levels. The rates of change are computed using the current and previous readings. Figure 4.14 is a tabular description of how the rate of change of blood sugar is used to calculate the amount of insulin to be delivered.

Figure 4.14 The tabular specification of computation in an insulin pump

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 − r1) < (r1 − r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing r2 > r1 & ((r2 − r1) ≥ (r1 − r0))	CompDose = round ((r2 − r1)/4) If rounded result = 0 then CompDose = MinimumDose

Figure 4.15 Use cases for the Mentcare system

4.4.3 Use cases

Use cases are a way of describing interactions between users and a system using a graphical model and structured text. They were first introduced in the Objectory method (Jacobsen et al. 1993) and have now become a fundamental feature of the Unified Modeling Language (UML). In their simplest form, a use case identifies the actors involved in an interaction and names the type of interaction. You then add additional information describing the interaction with the system. The additional information may be a textual description or one or more graphical models such as the UML sequence or state charts (see Chapter 5).

Use cases are documented using a high-level use case diagram. The set of use cases represents all of the possible interactions that will be described in the system requirements. Actors in the process, who may be human or other systems, are represented as stick figures. Each class of interaction is represented as a named ellipse. Lines link the actors with the interaction. Optionally, arrowheads may be added to lines to show how the interaction is initiated. This is illustrated in Figure 4.15, which shows some of the use cases for the Mentcare system.

Use cases identify the individual interactions between the system and its users or other systems. Each use case should be documented with a textual description. These can then be linked to other models in the UML that will develop the scenario in more detail. For example, a brief description of the Setup Consultation use case from Figure 4.15 might be:

> *Setup consultation allows two or more doctors, working in different offices, to view the same patient record at the same time. One doctor initiates the consultation by choosing the people involved from a dropdown menu of doctors who are online. The patient record is then displayed on their screens, but only the initiating doctor can edit the record. In addition, a text chat window is created*

to help coordinate actions. It is assumed that a phone call for voice communication can be separately arranged.

The UML is a standard for object-oriented modeling, so use cases and use case-based elicitation are used in the requirements engineering process. However, my experience with use cases is that they are too fine-grained to be useful in discussing requirements. Stakeholders don't understand the term *use case*; they don't find the graphical model to be useful, and they are often not interested in a detailed description of each and every system interaction. Consequently, I find use cases to be more helpful in systems design than in requirements engineering. I discuss use cases further in Chapter 5, which shows how they are used alongside other system models to document a system design.

Some people think that each use case is a single, low-level interaction scenario. Others, such as Stevens and Pooley (Stevens and Pooley 2006), suggest that each use case includes a set of related, low-level scenarios. Each of these scenarios is a single thread through the use case. Therefore, there would be a scenario for the normal interaction plus scenarios for each possible exception. In practice, you can use them in either way.

4.4.4 The software requirements document

The software requirements document (sometimes called the software requirements specification or SRS) is an official statement of what the system developers should implement. It may include both the user requirements for a system and a detailed specification of the system requirements. Sometimes the user and system requirements are integrated into a single description. In other cases, the user requirements are described in an introductory chapter in the system requirements specification.

Requirements documents are essential when systems are outsourced for development, when different teams develop different parts of the system, and when a detailed analysis of the requirements is mandatory. In other circumstances, such as software product or business system development, a detailed requirements document may not be needed.

Agile methods argue that requirements change so rapidly that a requirements document is out of date as soon as it is written, so the effort is largely wasted. Rather than a formal document, agile approaches often collect user requirements incrementally and write these on cards or whiteboards as short user stories. The user then prioritizes these stories for implementation in the next increment of the system.

For business systems where requirements are unstable, I think that this approach is a good one. However, I think that it is still useful to write a short supporting document that defines the business and dependability requirements for the system; it is easy to forget the requirements that apply to the system as a whole when focusing on the functional requirements for the next system release.

The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software. Figure 4.16 shows possible users of the document and how they use it.

Figure 4.16 Users of a requirements document

The diversity of possible users means that the requirements document has to be a compromise. It has to describe the requirements for customers, define the requirements in precise detail for developers and testers, as well as include information about future system evolution. Information on anticipated changes helps system designers to avoid restrictive design decisions and maintenance engineers to adapt the system to new requirements.

The level of detail that you should include in a requirements document depends on the type of system that is being developed and the development process used. Critical systems need detailed requirements because safety and security have to be analyzed in detail to find possible requirements errors. When the system is to be developed by a separate company (e.g., through outsourcing), the system specifications need to be detailed and precise. If an in-house, iterative development process is used, the requirements document can be less detailed. Details can be added to the requirements and ambiguities resolved during development of the system.

Figure 4.17 shows one possible organization for a requirements document that is based on an IEEE standard for requirements documents (IEEE 1998). This standard is a generic one that can be adapted to specific uses. In this case, the standard has been extended to include information about predicted system evolution. This information helps the maintainers of the system and allows designers to include support for future system features.

Chapter	Description
Preface	This defines the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This describes the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This defines the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter presents a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This describes the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This chapter includes graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This describes the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These provide detailed, specific information that is related to the application being developed—for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Figure 4.17 The structure of a requirements document

Naturally, the information included in a requirements document depends on the type of software being developed and the approach to development that is to be used. A requirements document with a structure like that shown in Figure 4.17 might be produced for a complex engineering system that includes hardware and software developed by different companies. The requirements document is likely to be long and detailed. It is therefore important that a comprehensive table of contents and document index be included so that readers can easily find the information they need.

By contrast, the requirements document for an in-house software product will leave out many of detailed chapters suggested above. The focus will be on defining the user requirements and high-level, nonfunctional system requirements. The system designers and programmers use their judgment to decide how to meet the outline user requirements for the system.

Requirements document standards

A number of large organizations, such as the U.S. Department of Defense and the IEEE, have defined standards for requirements documents. These are usually very generic but are nevertheless useful as a basis for developing more detailed organizational standards. The U.S. Institute of Electrical and Electronic Engineers (IEEE) is one of the best-known standards providers, and they have developed a standard for the structure of requirements documents. This standard is most appropriate for systems such as military command and control systems that have a long lifetime and are usually developed by a group of organizations.

http://software-engineering-book.com/web/requirements-standard/

4.5 Requirements validation

Requirements validation is the process of checking that requirements define the system that the customer really wants. It overlaps with elicitation and analysis, as it is concerned with finding problems with the requirements. Requirements validation is critically important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.

The cost of fixing a requirements problem by making a system change is usually much greater than repairing design or coding errors. A change to the requirements usually means that the system design and implementation must also be changed. Furthermore, the system must then be retested.

During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

1. *Validity checks* These check that the requirements reflect the real needs of system users. Because of changing circumstances, the user requirements may have changed since they were originally elicited.
2. *Consistency checks* Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.
3. *Completeness checks* The requirements document should include requirements that define all functions and the constraints intended by the system user.
4. *Realism checks* By using knowledge of existing technologies, the requirements should be checked to ensure that they can be implemented within the proposed budget for the system. These checks should also take account of the budget and schedule for the system development.
5. *Verifiability* To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

Requirements reviews

A requirements review is a process in which a group of people from the system customer and the system developer read the requirements document in detail and check for errors, anomalies, and inconsistencies. Once these have been detected and recorded, it is then up to the customer and the developer to negotiate how the identified problems should be solved.

http://software-engineering-book.com/web/requirements-reviews/

A number of requirements validation techniques can be used individually or in conjunction with one another:

1. *Requirements reviews* The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
2. *Prototyping* This involves developing an executable model of a system and using this with end-users and customers to see if it meets their needs and expectations. Stakeholders experiment with the system and feed back requirements changes to the development team.
3. *Test-case generation* Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of test-driven development.

You should not underestimate the problems involved in requirements validation. Ultimately, it is difficult to show that a set of requirements does in fact meet a user's needs. Users need to picture the system in operation and imagine how that system would fit into their work. It is hard even for skilled computer professionals to perform this type of abstract analysis and harder still for system users.

As a result, you rarely find all requirements problems during the requirements validation process. Inevitably, further requirements changes will be needed to correct omissions and misunderstandings after agreement has been reached on the requirements document.

4.6 Requirements change

The requirements for large software systems are always changing. One reason for the frequent changes is that these systems are often developed to address "wicked" problems—problems that cannot be completely defined (Rittel and Webber 1973). Because the problem cannot be fully defined, the software requirements are bound to

Figure 4.18
Requirements evolution

be incomplete. During the software development process, the stakeholders' understanding of the problem is constantly changing (Figure 4.18). The system requirements must then evolve to reflect this changed problem understanding.

Once a system has been installed and is regularly used, new requirements inevitably emerge. This is partly a consequence of errors and omissions in the original requirements that have to be corrected. However, most changes to system requirements arise because of changes to the business environment of the system:

1. The business and technical environment of the system always changes after installation. New hardware may be introduced and existing hardware updated. It may be necessary to interface the system with other systems. Business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that require system compliance.

2. The people who pay for a system and the users of that system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements, and, after delivery, new features may have to be added for user support if the system is to meet its goals.

3. Large systems usually have a diverse stakeholder community, with stakeholders having different requirements. Their priorities may be conflicting or contradictory. The final system requirements are inevitably a compromise, and some stakeholders have to be given priority. With experience, it is often discovered that the balance of support given to different stakeholders has to be changed and the requirements re-prioritized.

As requirements are evolving, you need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You therefore need a formal process for making change proposals and linking these to system requirements. This process of "requirements management" should start as soon as a draft version of the requirements document is available.

Agile development processes have been designed to cope with requirements that change during the development process. In these processes, when a user proposes a requirements change, this change does not go through a formal change management

Enduring and volatile requirements

Some requirements are more susceptible to change than others. Enduring requirements are the requirements that are associated with the core, slow-to-change activities of an organization. Enduring requirements are associated with fundamental work activities. Volatile requirements are more likely to change. They are usually associated with supporting activities that reflect how the organization does its work rather than the work itself.

http://software-engineering-book.com/web/changing-requirements/

process. Rather, the user has to prioritize that change and, if it is high priority, decide what system features that were planned for the next iteration should be dropped for the change to be implemented.

The problem with this approach is that users are not necessarily the best people to decide on whether or not a requirements change is cost-effective. In systems with multiple stakeholders, changes will benefit some stakeholders and not others. It is often better for an independent authority, who can balance the needs of all stakeholders, to decide on the changes that should be accepted.

4.6.1 Requirements management planning

Requirements management planning is concerned with establishing how a set of evolving requirements will be managed. During the planning stage, you have to decide on a number of issues:

1. *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and used in traceability assessments.
2. *A change management process* This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
3. *Traceability policies* These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.
4. *Tool support* Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to shared spreadsheets and simple database systems.

Requirements management needs automated support, and the software tools for this should be chosen during the planning phase. You need tool support for:

1. *Requirements storage* The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.

Figure 4.19
Requirements change management

2. *Change management* The process of change management (Figure 4.19) is simplified if active tool support is available. Tools can keep track of suggested changes and responses to these suggestions.

3. *Traceability management* As discussed above, tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

For small systems, you do not need to use specialized requirements management tools. Requirements management can be supported using shared web documents, spreadsheets, and databases. However, for larger systems, more specialized tool support, using systems such as DOORS (IBM 2013), makes it much easier to keep track of a large number of changing requirements.

4.6.2 Requirements change management

Requirements change management (Figure 4.19) should be applied to all proposed changes to a system's requirements after the requirements document has been approved. Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation. The advantage of using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way.

There are three principal stages to a change management process:

1. *Problem analysis and change specification* The process starts with an identified requirements problem or, sometimes, with a specific change proposal. During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.

2. *Change analysis and costing* The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made as to whether or not to proceed with the requirements change.

Requirements traceability

You need to keep track of the relationships between requirements, their sources, and the system design so that you can analyze the reasons for proposed changes and the impact that these changes are likely to have on other parts of the system. You need to be able to trace how a change ripples its way through the system. Why?

http://software-engineering-book.com/web/traceability/

3. *Change implementation* The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

If a new requirement has to be urgently implemented, there is always a temptation to change the system and then retrospectively modify the requirements document. This almost inevitably leads to the requirements specification and the system implementation getting out of step. Once system changes have been made, it is easy to forget to include these changes in the requirements document. In some circumstances, emergency changes to a system have to be made. In those cases, it is important that you update the requirements document as soon as possible in order to include the revised requirements.

KEY POINTS

- Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- Non-functional requirements often constrain the system being developed and the development process being used. These might be product requirements, organizational requirements, or external requirements. They often relate to the emergent properties of the system and therefore apply to the system as a whole.
- The requirements engineering process includes requirements elicitation, requirements specification, requirements validation, and requirements management.
- Requirements elicitation is an iterative process that can be represented as a spiral of activities—requirements discovery, requirements classification and organization, requirements negotiation, and requirements documentation.

- Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.
- The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.
- Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism, and verifiability.
- Business, organizational, and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.

FURTHER READING

"Integrated Requirements Engineering: A Tutorial." This is a tutorial paper that discusses requirements engineering activities and how these can be adapted to fit with modern software engineering practice. (I. Sommerville, *IEEE Software,* 22(1), January–February 2005) http://dx.doi.org/10.1109/MS.2005.13.

"Research Directions in Requirements Engineering." This is a good survey of requirements engineering research that highlights future research challenges in the area to address issues such as scale and agility. (B. H. C. Cheng and J. M. Atlee, *Proc. Conf. on Future of Software Engineering,* IEEE Computer Society, 2007) http://dx.doi.org/10.1109/FOSE.2007.17.

Mastering the Requirements Process, 3rd ed. A well-written, easy-to-read book that is based on a particular method (VOLERE) but that also includes lots of good general advice about requirements engineering. (S. Robertson and J. Robertson, 2013, Addison-Wesley).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/requirements-and-design/

Requirements document for the insulin pump:

http://software-engineering-book.com/case-studies/insulin-pump/

Mentcare system requirements information:

http://software-engineering-book.com/case-studies/mentcare-system/

EXERCISES

4.1. Identify and briefly describe four types of requirements that may be defined for a computer-based system.

4.2. Discover ambiguities or omissions in the following statement of the requirements for part of a drone system intended for search and recovery:

The drone, a quad chopper, will be very useful in search and recovery operations, especially in remote areas or in extreme weather conditions. It will click high-resolution images. It will fly according to a path preset by a ground operator, but will be able to avoid obstacles on its own, returning to its original path whenever possible. The drone will also be able to identify various objects and match them to the target it is looking for.

4.3. Rewrite the above description using the structured approach described in this chapter. Resolve the identified ambiguities in a sensible way.

4.4. Write a set of non-functional requirements for the drone system, setting out its expected safety and response time.

4.5. Using the technique suggested here, where natural language descriptions are presented in a standard format, write plausible user requirements for the following functions:

An unattended petrol (gas) pump system that includes a credit card reader. The customer swipes the card through the reader, then specifies the amount of fuel required. The fuel is delivered and the customer's account debited.

The cash-dispensing function in a bank ATM.

In an Internet banking system, a facility that allows customers to transfer funds from one account held with the bank to another account with the same bank.

4.6. Suggest how an engineer responsible for drawing up a system requirements specification might keep track of the relationships between functional and non-functional requirements.

4.7. Using your knowledge of how an ATM is used, develop a set of use cases that could serve as a basis for understanding the requirements for an ATM system.

4.8. To minimize mistakes during a requirements review, an organization decides to allocate two scribes to document the review session. Explain how this can be done.

4.9. When emergency changes have to be made to systems, the system software may have to be modified before changes to the requirements have been approved. Suggest a model of a process for making these modifications that will ensure that the requirements document and the system implementation do not become inconsistent.

4.10. You have taken a job with a software user who has contracted your previous employer to develop a system for them. You discover that your company's interpretation of the requirements is different from the interpretation taken by your previous employer. Discuss what you

should do in such a situation. You know that the costs to your current employer will increase if the ambiguities are not resolved. However, you also have a responsibility of confidentiality to your previous employer.

REFERENCES

Crabtree, A. 2003. *Designing Collaborative Systems: A Practical Guide to Ethnography*. London: Springer-Verlag.

Davis, A. M. 1993. *Software Requirements: Objects, Functions and States*. Englewood Cliffs, NJ: Prentice-Hall.

IBM. 2013. "Rational Doors Next Generation: Requirements Engineering for Complex Systems." https://jazz.net/products/rational-doors-next-generation/

IEEE. 1998. "IEEE Recommended Practice for Software Requirements Specifications." In *IEEE Software Engineering Standards Collection*. Los Alamitos, CA: IEEE Computer Society Press.

Jacobsen, I., M. Christerson, P. Jonsson, and G. Overgaard. 1993. *Object-Oriented Software Engineering*. Wokingham, UK: Addison-Wesley.

Martin, D., and I. Sommerville. 2004. "Patterns of Cooperative Interaction: Linking Ethnomethodology and Design." *ACM Transactions on Computer-Human Interaction* 11 (1) (March 1): 59–89. doi:10.1145/972648.972651.

Rittel, H., and M. Webber. 1973. "Dilemmas in a General Theory of Planning." *Policy Sciences* 4: 155–169. doi:10.1007/BF01405730.

Robertson, S., and J. Robertson. 2013. *Mastering the Requirements Process, 3rd ed.* Boston: Addison-Wesley.

Sommerville, I., T. Rodden, P. Sawyer, R. Bentley, and M. Twidale. 1993. "Integrating Ethnography into the Requirements Engineering Process." In *RE' 93*, 165–173. San Diego, CA: IEEE Computer Society Press. doi:10.1109/ISRE.1993.324821.

Stevens, P., and R. Pooley. 2006. *Using UML: Software Engineering with Objects and Components, 2nd ed.* Harlow, UK: Addison-Wesley.

Suchman, L. 1983. "Office Procedures as Practical Action: Models of Work and System Design." *ACM Transactions on Office Information Systems* 1 (3): 320–328. doi:10.1145/357442.357445.

Viller, S., and I. Sommerville. 2000. "Ethnographically Informed Analysis for Software Engineers." *Int. J. of Human-Computer Studies* 53 (1): 169–196. doi:10.1006/ijhc.2000.0370.

5

System modeling

Objectives

The aim of this chapter is to introduce system models that may be developed as part of requirements engineering and system design processes. When you have read the chapter, you will:

- understand how graphical models can be used to represent software systems and why several types of model are needed to fully represent a system;
- understand the fundamental system modeling perspectives of context, interaction, structure, and behavior;
- understand the principal diagram types in the Unified Modeling Language (UML) and how these diagrams may be used in system modeling;
- have been introduced to model-driven engineering, where an executable system is automatically generated from structural and behavioral models.

Contents

5.1 Context models
5.2 Interaction models
5.3 Structural models
5.4 Behavioral models
5.5 Model-driven engineering

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. System modeling now usually means representing a system using some kind of graphical notation based on diagram types in the Unified Modeling Language (UML). However, it is also possible to develop formal (mathematical) models of a system, usually as a detailed system specification. I cover graphical modeling using the UML here, and formal modeling is briefly discussed in Chapter 10.

Models are used during the requirements engineering process to help derive the detailed requirements for a system, during the design process to describe the system to engineers implementing the system, and after implementation to document the system's structure and operation. You may develop models of both the existing system and the system to be developed:

1. Models of the existing system are used during requirements engineering. They help clarify what the existing system does, and they can be used to focus a stakeholder discussion on its strengths and weaknesses.

2. Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation. If you use a model-driven engineering process (Brambilla, Cabot, and Wimmer 2012), you can generate a complete or partial system implementation from system models.

It is important to understand that a system model is not a complete representation of system. It purposely leaves out detail to make it easier to understand. A model is an abstraction of the system being studied rather than an alternative representation of that system. A representation of a system should maintain all the information about the entity being represented. An abstraction deliberately simplifies a system design and picks out the most salient characteristics. For example, the PowerPoint slides that accompany this book are an abstraction of the book's key points. However, if the book were translated from English into Italian, this would be an alternative *representation*. The translator's intention would be to maintain all the information as it is presented in English.

You may develop different models to represent the system from different perspectives. For example:

1. An external perspective, where you model the context or environment of the system.

2. An interaction perspective, where you model the interactions between a system and its environment, or between the components of a system.

3. A structural perspective, where you model the organization of a system or the structure of the data processed by the system.

4. A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

The Unified Modeling Language

The Unified Modeling Language (UML) is a set of 13 different diagram types that may be used to model software systems. It emerged from work in the 1990s on object-oriented modeling, where similar object-oriented notations were integrated to create the UML. A major revision (UML 2) was finalized in 2004. The UML is universally accepted as the standard approach for developing models of software systems. Variants, such as SysML, have been proposed for more general system modeling.

http://software-engineering-book.com/web/uml/

When developing system models, you can often be flexible in the way that the graphical notation is used. You do not always need to stick rigidly to the details of a notation. The detail and rigor of a model depend on how you intend to use it. There are three ways in which graphical models are commonly used:

1. As a way to stimulate and focus discussion about an existing or proposed system. The purpose of the model is to stimulate and focus discussion among the software engineers involved in developing the system. The models may be incomplete (as long as they cover the key points of the discussion), and they may use the modeling notation informally. This is how models are normally used in agile modeling (Ambler and Jeffries 2002).
2. As a way of documenting an existing system. When models are used as documentation, they do not have to be complete, as you may only need to use models to document some parts of a system. However, these models have to be correct—they should use the notation correctly and be an accurate description of the system.
3. As a detailed system description that can be used to generate a system implementation. Where models are used as part of a model-based development process, the system models have to be both complete and correct. They are used as a basis for generating the source code of the system, and you therefore have to be very careful not to confuse similar symbols, such as stick and block arrowheads, that may have different meanings.

In this chapter, I use diagrams defined in the Unified Modeling Language (UML) (Rumbaugh, Jacobson, and Booch 2004; Booch, Rumbaugh, and Jacobson 2005), which has become a standard language for object-oriented modeling. The UML has 13 diagram types and so supports the creation of many different types of system model. However, a survey (Erickson and Siau 2007) showed that most users of the UML thought that five diagram types could represent the essentials of a system. I therefore concentrate on these five UML diagram types here:

1. *Activity diagrams,* which show the activities involved in a process or in data processing.
2. *Use case diagrams,* which show the interactions between a system and its environment.
3. *Sequence diagrams,* which show interactions between actors and the system and between system components.
4. *Class diagrams,* which show the object classes in the system and the associations between these classes.
5. *State diagrams,* which show how the system reacts to internal and external events.

5.1 Context models

At an early stage in the specification of a system, you should decide on the system boundaries, that is, on what is and is not part of the system being developed. This involves working with system stakeholders to decide what functionality should be included in the system and what processing and operations should be carried out in the system's operational environment. You may decide that automated support for some business processes should be implemented in the software being developed but that other processes should be manual or supported by different systems. You should look at possible overlaps in functionality with existing systems and decide where new functionality should be implemented. These decisions should be made early in the process to limit the system costs and the time needed for understanding the system requirements and design.

In some cases, the boundary between a system and its environment is relatively clear. For example, where an automated system is replacing an existing manual or computerized system, the environment of the new system is usually the same as the existing system's environment. In other cases, there is more flexibility, and you decide what constitutes the boundary between the system and its environment during the requirements engineering process.

For example, say you are developing the specification for the Mentcare patient information system. This system is intended to manage information about patients attending mental health clinics and the treatments that have been prescribed. In developing the specification for this system, you have to decide whether the system should focus exclusively on collecting information about consultations (using other systems to collect personal information about patients) or whether it should also collect personal patient information. The advantage of relying on other systems for patient information is that you avoid duplicating data. The major disadvantage, however, is that using other systems may make it slower to access information, and if these systems are unavailable, then it may be impossible to use the Mentcare system.

In some situations, the user base for a system is very diverse, and users have a wide range of different system requirements. You may decide not to define

Figure 5.1 The context of the Mentcare system

boundaries explicitly but instead to develop a configurable system that can be adapted to the needs of different users. This was the approach that we adopted in the iLearn systems, introduced in Chapter 1. There, users range from very young children who can't read through to young adults, their teachers, and school administrators. Because these groups need different system boundaries, we specified a configuration system that would allow the boundaries to be specified when the system was deployed.

The definition of a system boundary is not a value-free judgment. Social and organizational concerns may mean that the position of a system boundary may be determined by nontechnical factors. For example, a system boundary may be deliberately positioned so that the complete analysis process can be carried out on one site; it may be chosen so that a particularly difficult manager need not be consulted; and it may be positioned so that the system cost is increased and the system development division must therefore expand to design and implement the system.

Once some decisions on the boundaries of the system have been made, part of the analysis activity is the definition of that context and the dependencies that a system has on its environment. Normally, producing a simple architectural model is the first step in this activity.

Figure 5.1 is a context model that shows the Mentcare system and the other systems in its environment. You can see that the Mentcare system is connected to an appointments system and a more general patient record system with which it shares data. The system is also connected to systems for management reporting and hospital admissions, and a statistics system that collects information for research. Finally, it makes use of a prescription system to generate prescriptions for patients' medication.

Context models normally show that the environment includes several other automated systems. However, they do not show the types of relationships between the systems in the environment and the system that is being specified. External systems might produce data for or consume data from the system. They might share data with the system, or they might be connected directly, through a network or not connected at all. They might be physically co-located or located in separate buildings. All of

Figure 5.2 A process model of involuntary detention

these relations may affect the requirements and design of the system being defined and so must be taken into account. Therefore, simple context models are used along with other models, such as business process models. These describe human and automated processes in which particular software systems are used.

UML activity diagrams may be used to show the business processes in which systems are used. Figure 5.2 is a UML activity diagram that shows where the Mentcare system is used in an important mental health care process—involuntary detention.

Sometimes, patients who are suffering from mental health problems may be a danger to others or to themselves. They may therefore have to be detained against their will in a hospital so that treatment can be administered. Such detention is subject to strict legal safeguards—for example, the decision to detain a patient must be regularly reviewed so that people are not held indefinitely without good reason. One critical function of the Mentcare system is to ensure that such safeguards are implemented and that the rights of patients are respected.

UML activity diagrams show the activities in a process and the flow of control from one activity to another. The start of a process is indicated by a filled circle, the end by a filled circle inside another circle. Rectangles with round corners represent activities, that is, the specific subprocesses that must be carried out. You may include objects in activity charts. Figure 5.2 shows the systems that are used to support different subprocesses within the involuntary detection process. I have shown that these are separate systems by using the UML stereotype feature where the type of entity in the box between chevrons is shown.

Arrows represent the flow of work from one activity to another, and a solid bar indicates activity coordination. When the flow from more than one activity leads to a

Figure 5.3 Transfer-data use case

solid bar, then all of these activities must be complete before progress is possible. When the flow from a solid bar leads to a number of activities, these may be executed in parallel. Therefore, in Figure 5.2, the activities to inform social care and the patient's next of kin, as well as to update the detention register, may be concurrent.

Arrows may be annotated with guards (in square brackets) that specify when that flow is followed. In Figure 5.2, you can see guards showing the flows for patients who are dangerous and not dangerous to society. Patients who are dangerous to society must be detained in a secure facility. However, patients who are suicidal and are a danger to themselves may be admitted to an appropriate ward in a hospital, where they can be kept under close supervision.

5.2 Interaction models

All systems involve interaction of some kind. This can be user interaction, which involves user inputs and outputs; interaction between the software being developed and other systems in its environment; or interaction between the components of a software system. User interaction modeling is important as it helps to identify user requirements. Modeling system-to-system interaction highlights the communication problems that may arise. Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.

This section discusses two related approaches to interaction modeling:

1. Use case modeling, which is mostly used to model interactions between a system and external agents (human users or other systems).
2. Sequence diagrams, which are used to model interactions between system components, although external agents may also be included.

Use case models and sequence diagrams present interactions at different levels of detail and so may be used together. For example, the details of the interactions involved in a high-level use case may be documented in a sequence diagram. The UML also includes communication diagrams that can be used to model interactions. I don't describe this diagram type because communication diagrams are simply an alternative representation of sequence diagrams.

5.2.1 Use case modeling

Use case modeling was originally developed by Ivar Jacobsen in the 1990s (Jacobsen et al. 1993), and a UML diagram type to support use case modeling is part of the

Mentcare system: Transfer data	
Actors	Medical receptionist, Patient records system (PRS)
Description	A receptionist may transfer data from the Mentcare system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

Figure 5.4 Tabular description of the Transfer-data use case

UML. A use case can be taken as a simple description of what a user expects from a system in that interaction. I have discussed use cases for requirements elicitation in Chapter 4. As I said in Chapter 4, I find use case models to be more useful in the early stages of system design rather than in requirements engineering.

Each use case represents a discrete task that involves external interaction with a system. In its simplest form, a use case is shown as an ellipse, with the actors involved in the use case represented as stick figures. Figure 5.3 shows a use case from the Mentcare system that represents the task of uploading data from the Mentcare system to a more general patient record system. This more general system maintains summary data about a patient rather than data about each consultation, which is recorded in the Mentcare system.

Notice that there are two actors in this use case—the operator who is transferring the data and the patient record system. The stick figure notation was originally developed to cover human interaction, but it is also used to represent other external systems and hardware. Formally, use case diagrams should use lines without arrows as arrows in the UML indicate the direction of flow of messages. Obviously, in a use case, messages pass in both directions. However, the arrows in Figure 5.3 are used informally to indicate that the medical receptionist initiates the transaction and data is transferred to the patient record system.

Use case diagrams give a simple overview of an interaction, and you need to add more detail for complete interaction description. This detail can either be a simple textual description, a structured description in a table, or a sequence diagram. You choose the most appropriate format depending on the use case and the level of detail that you think is required in the model. I find a standard tabular format to be the most useful. Figure 5.4 shows a tabular description of the "Transfer data" use case.

Composite use case diagrams show a number of different use cases. Sometimes it is possible to include all possible interactions within a system in a single composite use case diagram. However, this may be impossible because of the number of use cases. In such cases, you may develop several diagrams, each of which shows related use cases. For example, Figure 5.5 shows all of the use cases in the Mentcare system

Figure 5.5 Use cases involving the role "Medical receptionist"

in which the actor "Medical Receptionist" is involved. Each of these should be accompanied by a more detailed description.

The UML includes a number of constructs for sharing all or part of a use case in other use case diagrams. While these constructs can sometimes be helpful for system designers, my experience is that many people, especially end-users, find them difficult to understand. For this reason, these constructs are not described here.

5.2.2 Sequence diagrams

Sequence diagrams in the UML are primarily used to model the interactions between the actors and the objects in a system and the interactions between the objects themselves. The UML has a rich syntax for sequence diagrams, which allows many different kinds of interaction to be modeled. As space does not allow covering all possibilities here, the focus will be on the basics of this diagram type.

As the name implies, a sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance. Figure 5.6 is an example of a sequence diagram that illustrates the basics of the notation. This diagram models the interactions involved in the View patient information use case, where a medical receptionist can see some patient information.

The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these. Annotated arrows indicate interactions between objects. The rectangle on the dotted lines indicates the lifeline of the object concerned (i.e., the time that object instance is involved in the computation). You read the sequence of interactions from top to bottom. The annotations on the arrows indicate the calls to the objects, their parameters, and the return values. This example also shows the notation used to denote alternatives. A box named alt is used with the

Figure 5.6 Sequence diagram for View patient information

conditions indicated in square brackets, with alternative interaction options separated by a dotted line.

You can read Figure 5.6 as follows:

1. The medical receptionist triggers the ViewInfo method in an instance P of the PatientInfo object class, supplying the patient's identifier, PID to identify the required information. P is a user interface object, which is displayed as a form showing patient information.
2. The instance P calls the database to return the information required, supplying the receptionist's identifier to allow security checking. (At this stage, it is not important where the receptionist's UID comes from.)
3. The database checks with an authorization system that the receptionist is authorized for this action.
4. If authorized, the patient information is returned and is displayed on a form on the user's screen. If authorization fails, then an error message is returned. The box denoted by "alt" in the top-left corner is a choice box indicating that one of the contained interactions will be executed. The condition that selects the choice is shown in square brackets.

Figure 5.7 is a further example of a sequence diagram from the same system that illustrates two additional features. These are the direct communication between the actors in the system and the creation of objects as part of a sequence of operations. In this example, an object of type Summary is created to hold the summary data that is

Figure 5.7 Sequence diagram for Transfer Data

to be uploaded to a national PRS (patient records system). You can read this diagram as follows:

1. The receptionist logs on to the PRS.
2. Two options are available (as shown in the "alt" box). These allow the direct transfer of updated patient information from the Mentcare database to the PRS and the transfer of summary health data from the Mentcare database to the PRS.
3. In each case, the receptionist's permissions are checked using the authorization system.

4. Personal information may be transferred directly from the user interface object to the PRS. Alternatively, a summary record may be created from the database, and that record is then transferred.

5. On completion of the transfer, the PRS issues a status message and the user logs off.

Unless you are using sequence diagrams for code generation or detailed documentation, you don't have to include every interaction in these diagrams. If you develop system models early in the development process to support requirements engineering and high-level design, there will be many interactions that depend on implementation decisions. For example, in Figure 5.7 the decision on how to get the user identifier to check authorization is one that can be delayed. In an implementation, this might involve interacting with a User object. As this is not important at this stage, you do not need to include it in the sequence diagram.

5.3 Structural models

Structural models of software display the organization of a system in terms of the components that make up that system and their relationships. Structural models may be static models, which show the organization of the system design, or dynamic models, which show the organization of the system when it is executing. These are not the same things—the dynamic organization of a system as a set of interacting threads may be very different from a static model of the system components.

You create structural models of a system when you are discussing and designing the system architecture. These can be models of the overall system architecture or more detailed models of the objects in the system and their relationships.

In this section, I focus on the use of class diagrams for modeling the static structure of the object classes in a software system. Architectural design is an important topic in software engineering, and UML component, package, and deployment diagrams may all be used when presenting architectural models. I cover architectural modeling in Chapters 6 and 17.

5.3.1 Class diagrams

Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes. Loosely, an object class can be thought of as a general definition of one kind of system object. An association is a link between classes indicating that some relationship exists between these classes. Consequently, each class may have to have some knowledge of its associated class.

When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a

Figure 5.8 UML Classes and association

Figure 5.9 Classes and associations in the Mentcare system

prescription, or a doctor. As an implementation is developed, you define implementation objects to represent data that is manipulated by the system. In this section, the focus is on the modeling of real-world objects as part of the requirements or early software design processes. A similar approach is used for data structure modeling.

Class diagrams in the UML can be expressed at different levels of detail. When you are developing a model, the first stage is usually to look at the world, identify the essential objects, and represent these as classes. The simplest way of writing these diagrams is to write the class name in a box. You can also note the existence of an association by drawing a line between classes. For example, Figure 5.8 is a simple class diagram showing two classes, Patient and Patient Record, with an association between them. At this stage, you do not need to say what the association is.

Figure 5.9 develops the simple class diagram in Figure 5.8 to show that objects of class Patient are also involved in relationships with a number of other classes. In this example, I show that you can name associations to give the reader an indication of the type of relationship that exists.

Figures 5.8 and 5.9, shows an important feature of class diagrams—the ability to show how many objects are involved in the association. In Figure 5.8 each end of the association is annotated with a 1, meaning that there is a 1:1 relationship between objects of these classes. That is, each patient has exactly one record, and each record maintains information about exactly one patient.

As you can see from Figure 5.9, other multiplicities are possible. You can define that an exact number of objects are involved (e.g., 1..4) or, by using a *, indicate that there are an indefinite number of objects involved in the association. For example, the (1..*) multiplicity in Figure 5.9 on the relationship between Patient and Condition shows that a patient may suffer from several conditions and that the same condition may be associated with several patients.

Figure 5.10 A Consultation class

At this level of detail, class diagrams look like semantic data models. Semantic data models are used in database design. They show the data entities, their associated attributes, and the relations between these entities (Hull and King 1987). The UML does not include a diagram type for database modeling, as it models data using objects and their relationships. However, you can use the UML to represent a semantic data model. You can think of entities in a semantic data model as simplified object classes (they have no operations), attributes as object class attributes, and relations as named associations between object classes.

When showing the associations between classes, it is best to represent these classes in the simplest possible way, without attributes or operations. To define objects in more detail, you add information about their attributes (the object's characteristics) and operations (the object's functions). For example, a Patient object has the attribute Address, and you may include an operation called ChangeAddress, which is called when a patient indicates that he or she has moved from one address to another.

In the UML, you show attributes and operations by extending the simple rectangle that represents a class. I illustrate this in Figure 5.10 that shows an object representing a consultation between doctor and patient:

1. The name of the object class is in the top section.
2. The class attributes are in the middle section. This includes the attribute names and, optionally, their types. I don't show the types in Figure 5.10.
3. The operations (called methods in Java and other OO programming languages) associated with the object class are in the lower section of the rectangle. I show some but not all operations in Figure 5.10.

In the example shown in Figure 5.10, it is assumed that doctors record voice notes that are transcribed later to record details of the consultation. To prescribe medication, the doctor involved must use the Prescribe method to generate an electronic prescription.

Figure 5.11 A generalization hierarchy

5.3.2 Generalization

Generalization is an everyday technique that we use to manage complexity. Rather than learn the detailed characteristics of everything that we experience, we learn about general classes (animals, cars, houses, etc.) and learn the characteristics of these classes. We then reuse knowledge by classifying things and focus on the differences between them and their class. For example, squirrels and rats are members of the class "rodents," and so share the characteristics of rodents. General statements apply to all class members; for example, all rodents have teeth for gnawing.

When you are modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization and class creation. This means that common information will be maintained in one place only. This is good design practice as it means that, if changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change. You can make the changes at the most general level. In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.

The UML has a specific type of association to denote generalization, as illustrated in Figure 5.11. The generalization is shown as an arrowhead pointing up to the more general class. This indicates that general practitioners and hospital doctors can be generalized as doctors and that there are three types of Hospital Doctor: those who have just graduated from medical school and have to be supervised (Trainee Doctor); those who can work unsupervised as part of a consultant's team (Registered Doctor); and consultants, who are senior doctors with full decision-making responsibilities.

In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes. The lower-level classes are subclasses that inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.

Figure 5.12 A generalization hierarchy with added detail

For example, all doctors have a name and phone number, and all hospital doctors have a staff number and carry a pager. General practitioners don't have these attributes, as they work independently, but they have an individual practice name and address. Figure 5.12 shows part of the generalization hierarchy, which I have extended with class attributes, for the class Doctor. The operations associated with the class Doctor are intended to register and de-register that doctor with the Mentcare system.

5.3.3 Aggregation

Objects in the real world are often made up of different parts. For example, a study pack for a course may be composed of a book, PowerPoint slides, quizzes, and recommendations for further reading. Sometimes in a system model, you need to illustrate this. The UML provides a special type of association between classes called aggregation, which means that one object (the whole) is composed of other objects (the parts). To define aggregation, a diamond shape is added to the link next to the class that represents the whole.

Figure 5.13 shows that a patient record is an aggregate of Patient and an indefinite number of Consultations. That is, the record maintains personal patient information as well as an individual record for each consultation with a doctor.

Figure 5.13 The aggregation association

Data flow diagrams

Data-flow diagrams (DFDs) are system models that show a functional perspective where each transformation represents a single function or process. DFDs are used to show how data flows through a sequence of processing steps. For example, a processing step could be the filtering of duplicate records in a customer database. The data is transformed at each step before moving on to the next stage. These processing steps or transformations represent software processes or functions, where data-flow diagrams are used to document a software design. Activity diagrams in the UML may be used to represent DFDs.

http://software-engineering-book.com/web/dfds/

5.4 Behavioral models

Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment. These stimuli may be either data or events:

1. Data becomes available that has to be processed by the system. The availability of the data triggers the processing.
2. An event happens that triggers system processing. Events may have associated data, although this is not always the case.

Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing. Their processing involves a sequence of actions on that data and the generation of an output. For example, a phone billing system will accept information about calls made by a customer, calculate the costs of these calls, and generate a bill for that customer.

By contrast, real-time systems are usually event-driven, with limited data processing. For example, a landline phone switching system responds to events such as "handset activated" by generating a dial tone, pressing keys on a handset by capturing the phone number, and so on.

5.4.1 Data-driven modeling

Data-driven models show the sequence of actions involved in processing input data and generating an associated output. They can be used during the analysis of requirements as they show end-to-end processing in a system. That is, they show the entire sequence of actions that takes place from an initial input being processed to the corresponding output, which is the system's response.

Data-driven models were among the first graphical software models. In the 1970s, structured design methods used data-flow diagrams (DFDs) as a way to illustrate the

Figure 5.14 An activity model of the insulin pump's operation

processing steps in a system. Data-flow models are useful because tracking and documenting how data associated with a particular process moves through the system help analysts and designers understand what is going on in the process. DFDs are simple and intuitive and so are more accessible to stakeholders than some other types of model. It is usually possible to explain them to potential system users who can then participate in validating the model.

Data-flow diagrams can be represented in the UML using the activity diagram type, described in Section 5.1. Figure 5.14 is a simple activity diagram that shows the chain of processing involved in the insulin pump software. You can see the processing steps, represented as activities (rounded rectangles), and the data flowing between these steps, represented as objects (rectangles).

An alternative way of showing the sequence of processing in a system is to use UML sequence diagrams. You have seen how these diagrams can be used to model interaction, but if you draw these so that messages are only sent from left to right, then they show the sequential data processing in the system. Figure 5.15 illustrates this, using a sequence model of processing an order and sending it to a supplier. Sequence models highlight objects in a system, whereas data-flow diagrams highlight the operations or activities. In practice, nonexperts seem to find data-flow diagrams more intuitive, but engineers prefer sequence diagrams.

Figure 5.15 Order processing

Figure 5.16 A state diagram of a microwave oven

5.4.2 Event-driven modeling

Event-driven modeling shows how a system responds to external and internal events. It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another. For example, a system controlling a valve may move from a state "Valve open" to a state "Valve closed" when an operator command (the stimulus) is received. This view of a system is particularly appropriate for real-time systems. Event-driven modeling is used extensively when designing and documenting real-time systems (Chapter 21).

The UML supports event-based modeling using state diagrams, which are based on Statecharts (Harel 1987). State diagrams show system states and events that cause transitions from one state to another. They do not show the flow of data within the system but may include additional information on the computations carried out in each state.

I use an example of control software for a very simple microwave oven to illustrate event-driven modeling (Figure 5.16). Real microwave ovens are much more complex than this system, but the simplified system is easier to understand. This simple oven has a switch to select full or half power, a numeric keypad to input the cooking time, a start/stop button, and an alphanumeric display.

Figure 5.17 A state model of the Operation state

I have assumed that the sequence of actions in using the microwave is as follows:

1. Select the power level (either half power or full power).
2. Input the cooking time using a numeric keypad.
3. Press **Start** and the food is cooked for the given time.

For safety reasons, the oven should not operate when the door is open, and, on completion of cooking, a buzzer is sounded. The oven has a simple display that is used to display various alerts and warning messages.

In UML state diagrams, rounded rectangles represent system states. They may include a brief description (following "do") of the actions taken in that state. The labeled arrows represent stimuli that force a transition from one state to another. You can indicate start and end states using filled circles, as in activity diagrams.

From Figure 5.16, you can see that the system starts in a waiting state and responds initially to either the full-power or the half-power button. Users can change their minds after selecting one of these and may press the other button. The time is set and, if the door is closed, the Start button is enabled. Pushing this button starts the oven operation, and cooking takes place for the specified time. This is the end of the cooking cycle, and the system returns to the waiting state.

The problem with state-based modeling is that the number of possible states increases rapidly. For large system models, therefore, you need to hide detail in the models. One way to do this is by using the notion of a "superstate" that encapsulates a number of separate states. This superstate looks like a single state on a high-level model but is then expanded to show more detail on a separate diagram. To illustrate this concept, consider the **Operation** state in Figure 5.16. This is a superstate that can be expanded, as shown in Figure 5.17.

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows "Half power."
Full power	The oven power is set to 600 watts. The display shows "Full power."
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows "Not ready."
Enabled	Oven operation is enabled. Interior oven light is off. Display shows "Ready to cook."
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows "Cooking complete" while buzzer is sounding.
Stimulus	**Description**
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.

Figure 5.18 States and stimuli for the microwave oven

The Operation state includes a number of substates. It shows that operation starts with a status check and that if any problems are discovered an alarm is indicated and operation is disabled. Cooking involves running the microwave generator for the specified time; on completion, a buzzer is sounded. If the door is opened during operation, the system moves to the disabled state, as shown in Figure 5.17.

State models of a system provide an overview of event processing, but you normally have to extend this with a more detailed description of the stimuli and the system states. You may use a table to list the states and events that stimulate state transitions along with a description of each state and event. Figure 5.18 shows a tabular description of each state and how the stimuli that force state transitions are generated.

5.4.3 Model-driven engineering

Model-driven engineering (MDE) is an approach to software development whereby models rather than programs are the principal outputs of the development process

(Brambilla, Cabot, and Wimmer 2012). The programs that execute on a hardware/software platform are generated automatically from the models. Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

Model-driven engineering was developed from the idea of model-driven architecture (MDA). This was proposed by the Object Management Group (OMG) as a new software development paradigm (Mellor, Scott, and Weise 2004). MDA focuses on the design and implementation stages of software development, whereas MDE is concerned with all aspects of the software engineering process. Therefore, topics such as model-based requirements engineering, software processes for model-based development, and model-based testing are part of MDE but are not considered in MDA.

MDA as an approach to system engineering has been adopted by a number of large companies to support their development processes. This section focuses on the use of MDA for software implementation rather than discuss more general aspects of MDE. The take-up of more general model-driven engineering has been slow, and few companies have adopted this approach throughout their software development life cycle. In his blog, den Haan discusses possible reasons why MDE has not been widely adopted (den Haan 2011).

5.5 Model-driven architecture

Model-driven architecture (Mellor, Scott, and Weise 2004; Stahl and Voelter 2006) is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system. Here, models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.

The MDA method recommends that three types of abstract system model should be produced:

1. *A computation independent model (CIM)* CIMs model the important domain abstractions used in a system and so are sometimes called domain models. You may develop several different CIMs, reflecting different views of the system. For example, there may be a security CIM in which you identify important security abstractions such as an asset, and a role and a patient record CIM, in which you describe abstractions such as patients and consultations.

2. *A platform-independent model (PIM)* PIMs model the operation of the system without reference to its implementation. A PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.

Figure 5.19 MDA transformations

3. *Platform-specific models (PSM)* PSMs are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail. So, the first level PSM could be middleware-specific but database-independent. When a specific database has been chosen, a database-specific PSM can then be generated.

Model-based engineering allows engineers to think about systems at a high level of abstraction, without concern for the details of their implementation. This reduces the likelihood of errors, speeds up the design and implementation process, and allows for the creation of reusable, platform-independent application models. By using powerful tools, system implementations can be generated for different platforms from the same model. Therefore, to adapt the system to some new platform technology, you write a model translator for that platform. When this is available, all platform-independent models can then be rapidly re-hosted on the new platform.

Fundamental to MDA is the notion that transformations between models can be defined and applied automatically by software tools, as illustrated in Figure 5.19. This diagram also shows a final level of automatic transformation where a transformation is applied to the PSM to generate the executable code that will run on the designated software platform. Therefore, in principle at least, executable software can be generated from a high-level system model.

In practice, completely automated translation of models to code is rarely possible. The translation of high-level CIM to PIM models remains a research problem, and for production systems, human intervention, illustrated using a stick figure in Figure 5.19, is normally required. A particularly difficult problem for automated model transformation is the need to link the concepts used in different CIMS. For example, the concept of a role in a security CIM that includes role-driven access control may have to be mapped onto the concept of a staff member in a hospital CIM. Only a person who understands both security and the hospital environment can make this mapping.

Figure 5.20 Multiple platform-specific models

The translation of platform-independent to platform-specific models is a simpler technical problem. Commercial tools and open-source tools (Koegel 2012) are available that provide translators from PIMS to common platforms such as Java and J2EE. These use an extensive library of platform-specific rules and patterns to convert a PIM to a PSM. There may be several PSMs for each PIM in the system. If a software system is intended to run on different platforms (e.g., J2EE and .NET), then, in principle, you only have to maintain a single PIM. The PSMs for each platform are automatically generated (Figure 5.20).

Although MDA support tools include platform-specific translators, these sometimes only offer partial support for translating PIMS to PSMs. The execution environment for a system is more than the standard execution platform, such as J2EE or Java. It also includes other application systems, specific application libraries that may be created for a company, external services, and user interface libraries.

These vary from one company to another, so off-the-shelf tool support is not available that takes these into account. Therefore, when MDA is introduced into an organization, special-purpose translators may have to be created to make use of the facilities available in the local environment. This is one reason why many companies have been reluctant to take on model-driven approaches to development. They do not want to develop or maintain their own tools or to rely on small software companies, who may go out of business, for tool development. Without these specialist tools, model-based development requires additional manual coding which reduces the cost-effectiveness of this approach.

I believe that there are several other reasons why MDA has not become a mainstream approach to software development.

1. Models are a good way of facilitating discussions about a software design. However, it does not always follow that the abstractions that are useful for discussions are the right abstractions for implementation. You may decide to use a completely different implementation approach that is based on the reuse of off-the-shelf application systems.

2. For most complex systems, implementation is not the major problem—requirements engineering, security and dependability, integration with legacy

Executable UML

The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible. To achieve this, you have to be able to construct graphical models with clearly defined meanings that can be compiled to executable code. You also need a way of adding information to graphical models about the ways in which the operations defined in the model are implemented. This is possible using a subset of UML 2, called Executable UML or xUML (Mellor and Balcer 2002).

http://software-engineering-book.com/web/xuml/

systems and testing are all more significant. Consequently, the gains from the use of MDA are limited.

3. The arguments for platform independence are only valid for large, long-lifetime systems, where the platforms become obsolete during a system's lifetime. For software products and information systems that are developed for standard platforms, such as Windows and Linux, the savings from the use of MDA are likely to be outweighed by the costs of its introduction and tooling.

4. The widespread adoption of agile methods over the same period that MDA was evolving has diverted attention away from model-driven approaches.

The success stories for MDA (OMG 2012) have mostly come from companies that are developing systems products, which include both hardware and software. The software in these products has a long lifetime and may have to be modified to reflect changing hardware technologies. The domain of application (automotive, air traffic control, etc.) is often well understood and so can be formalized in a CIM.

Hutchinson and his colleagues (Hutchinson, Rouncefield, and Whittle 2012) report on the industrial use of MDA, and their work confirms that successes in the use of model-driven development have been in systems products. Their assessment suggests that companies have had mixed results when adopting this approach, but the majority of users report that using MDA has increased productivity and reduced maintenance costs. They found that MDA was particularly useful in facilitating reuse, and this led to major productivity improvements.

There is an uneasy relationship between agile methods and model-driven architecture. The notion of extensive up-front modeling contradicts the fundamental ideas in the agile manifesto and I suspect that few agile developers feel comfortable with model-driven engineering. Ambler, a pioneer in the development of agile methods, suggests that some aspects of MDA can be used in agile processes (Ambler 2004) but considers automated code generation to be impractical. However, Zhang and Patel report on Motorola's success in using agile development with automated code generation (Zhang and Patel 2011).

KEY POINTS

- A model is an abstract view of a system that deliberately ignores some system details. Complementary system models can be developed to show the system's context, interactions, structure, and behavior.
- Context models show how a system that is being modeled is positioned in an environment with other systems and processes. They help define the boundaries of the system to be developed.
- Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.
- Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.
- Behavioral models are used to describe the dynamic behavior of an executing system. This behavior can be modeled from the perspective of the data processed by the system or by the events that stimulate responses from a system.
- Activity diagrams may be used to model the processing of data, where each activity represents one process step.
- State diagrams are used to model a system's behavior in response to internal or external events.
- Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.

FURTHER READING

Any of the introductory books on the UML provide more information about the notation than I can cover here. UML has only changed slightly in the last few years, so although some of these books are almost 10 years old, they are still relevant.

Using UML: Software Engineering with Objects and Components, 2nd ed. This book is a short, readable introduction to the use of the UML in system specification and design. I think that it is excellent for learning and understanding the UML notation, although it is less comprehensive than the complete descriptions of UML found in the UML reference manual. (P. Stevens with R. Pooley, Addison-Wesley, 2006)

Model-driven Software Engineering in Practice. This is quite a comprehensive book on model-driven approaches with a focus on model-driven design and implementation. As well as the UML, it also covers the development of domain-specific modeling languages. (M. Brambilla, J. Cabot, and M. Wimmer. Morgan Claypool, 2012)

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/requirements-and-design/

EXERCISES

5.1. Scope creep can be defined as a continuous increase in the scope of a project that can significantly increase project cost. Explain how a proper model of the system context can help prevent scope creeps.

5.2. The way in which a system boundary is defined and an appropriate context model is created may have serious implications on the complexity and cost of a project. Give two examples where this may be applicable.

5.3. You have been asked to develop a system that will help with planning large-scale events and parties such as weddings, graduation celebrations, and birthday parties. Using an activity diagram, model the process context for such a system that shows the activities involved in planning a party (booking a venue, organizing invitations, etc.) and the system elements that might be used at each stage.

5.4. For the Mentcare system, propose a set of use cases that illustrates the interactions between a doctor, who sees patients and prescribes medicine and treatments, and the Mentcare system.

5.5. Develop a sequence diagram showing the interactions involved when a student registers for a course in a university. Courses may have limited enrollment, so the registration process must include checks that places are available. Assume that the student accesses an electronic course catalog to find out about available courses.

5.6. Look carefully at how messages and mailboxes are represented in the email system that you use. Model the object classes that might be used in the system implementation to represent a mailbox and an email message.

5.7. Based on your experience with a bank ATM, draw an activity diagram that models the data processing involved when a customer withdraws cash from the machine.

5.8. Draw a sequence diagram for the same system. Explain why you might want to develop both activity and sequence diagrams when modeling the behavior of a system.

5.9. Draw state diagrams of the control software for:

- an automatic washing machine that has different programs for different types of clothes;
- the software for a DVD player;
- the control software for the camera on your mobile phone. Ignore the flash if you have one on your phone.

5.10. In principle, it is possible to generate working programs from a high-level model without manual intervention when using model-driven architectures. Discuss some of the current challenges that stand in the way of the existence of completely automated translation tools.

REFERENCES

Ambler, S. W. 2004. *The Object Primer: Agile Model-Driven Development with UML 2.0, 3rd ed*. Cambridge, UK: Cambridge University Press.

Ambler, S. W., and R. Jeffries. 2002. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons.

Booch, G., J. Rumbaugh, and I. Jacobson. 2005. *The Unified Modeling Language User Guide, 2nd ed*. Boston: Addison-Wesley.

Brambilla, M., J. Cabot, and M. Wimmer. 2012. *Model-Driven Software Engineering in Practice*. San Rafael, CA: Morgan Claypool.

Den Haan, J. 2011. "Why There Is No Future for Model Driven Development." http://www.theenterprisearchitect.eu/archive/2011/01/25/why-there-is-no-future-for-model-driven-development/

Erickson, J,, and K Siau. 2007. "Theoretical and Practical Complexity of Modeling Methods." *Comm. ACM* 50 (8): 46–51. doi:10.1145/1278201.1278205.

Harel, D. 1987. "Statecharts: A Visual Formalism for Complex Systems." *Sci. Comput. Programming* 8 (3): 231–274. doi:10.1016/0167-6423(87)90035-9.

Hull, R., and R King. 1987. "Semantic Database Modeling: Survey, Applications and Research Issues." *ACM Computing Surveys* 19 (3): 201–260. doi:10.1145/45072.45073.

Hutchinson, J., M. Rouncefield, and J. Whittle. 2012. "Model-Driven Engineering Practices in Industry." In *34th Int. Conf. on Software Engineering*, 633–642. doi:10.1145/1985793.1985882.

Jacobsen, I., M. Christerson, P. Jonsson, and G. Overgaard. 1993. *Object-Oriented Software Engineering*. Wokingham, UK: Addison-Wesley.

Koegel, M. 2012. "EMF Tutorial: What Every Eclipse Developer Should Know about EMF." http://eclipsesource.com/blogs/tutorials/emf-tutorial/

Mellor, S. J., and M. J. Balcer. 2002. *Executable UML*. Boston: Addison-Wesley.

Mellor, S. J., K. Scott, and D. Weise. 2004. *MDA Distilled: Principles of Model-Driven Architecture*. Boston: Addison-Wesley.

OMG. 2012. "Model-Driven Architecture: Success Stories." http://www.omg.org/mda/products_success.htm

Rumbaugh, J., I. Jacobson, and G Booch. 2004. *The Unified Modelling Language Reference Manual, 2nd ed.* Boston: Addison-Wesley.

Stahl, T., and M. Voelter. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. New York: John Wiley & Sons.

Zhang, Y., and S. Patel. 2011. "Agile Model-Driven Development in Practice." *IEEE Software* 28 (2): 84–91. doi:10.1109/MS.2010.85.

6

Architectural design

Objectives

The objective of this chapter is to introduce the concepts of software architecture and architectural design. When you have read the chapter, you will:

- understand why the architectural design of software is important;
- understand the decisions that have to be made about the software architecture during the architectural design process;
- have been introduced to the idea of Architectural patterns, well-tried ways of organizing software architectures that can be reused in system designs;
- understand how Application-Specific Architectural patterns may be used in transaction processing and language processing systems.

Contents

6.1 Architectural design decisions
6.2 Architectural views
6.3 Architectural patterns
6.4 Application architectures

Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system. In the model of the software development process that I described in Chapter 2, architectural design is the first stage in the software design process. It is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them. The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

In agile processes, it is generally accepted that an early stage of an agile development process should focus on designing an overall system architecture. Incremental development of architectures is not usually successful. Refactoring components in response to changes is usually relatively easy. However, refactoring the system architecture is expensive because you may need to modify most system components to adapt them to the architectural changes.

To help you understand what I mean by system architecture, look at Figure 6.1. This diagram shows an abstract model of the architecture for a packing robot system. This robotic system can pack different kinds of objects. It uses a vision component to pick out objects on a conveyor, identify the type of object, and select the right kind of packaging. The system then moves objects from the delivery conveyor to be packaged. It places packaged objects on another conveyor. The architectural model shows these components and the links between them.

In practice, there is a significant overlap between the processes of requirements engineering and architectural design. Ideally, a system specification should not

Figure 6.1 The architecture of a packing robot control system

include any design information. This ideal is unrealistic, however, except for very small systems. You need to identify the main architectural components as these reflect the high-level features of the system. Therefore, as part of the requirements engineering process, you might propose an abstract system architecture where you associate groups of system functions or features with large-scale components or subsystems. You then use this decomposition to discuss the requirements and more detailed features of the system with stakeholders.

You can design software architectures at two levels of abstraction, which I call *architecture in the small* and *architecture in the large:*

1. *Architecture in the small* is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components. This chapter is mostly concerned with program architectures.

2. *Architecture in the large* is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems may be distributed over different computers, which may be owned and managed by different companies. (I cover architecture in the large in Chapters 17 and 18.)

Software architecture is important because it affects the performance, robustness, distributability, and maintainability of a system (Bosch 2000). As Bosch explains, individual components implement the functional system requirements, but the dominant influence on the non-functional system characteristics is the system's architecture. Chen et al. (Chen, Ali Babar, and Nuseibeh 2013) confirmed this in a study of "architecturally significant requirements" where they found that non-functional requirements had the most significant effect on the system's architecture.

Bass et al. (Bass, Clements, and Kazman 2012) suggest that explicitly designing and documenting software architecture has three advantages:

1. *Stakeholder communication* The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.

2. *System analysis* Making the system architecture explicit at an early stage in the system development requires some analysis. Architectural design decisions have a profound effect on whether or not the system can meet critical requirements such as performance, reliability, and maintainability.

3. *Large-scale reuse* An architectural model is a compact, manageable description of how a system is organized and how the components interoperate. The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse. As I explain in Chapter 15, product-line architectures are an approach to reuse where the same architecture is reused across a range of related systems.

System architectures are often modeled informally using simple block diagrams, as in Figure 6.1. Each box in the diagram represents a component. Boxes within boxes indicate that the component has been decomposed to subcomponents. Arrows mean that data and or control signals are passed from component to component in the direction of the arrows. You can see many examples of this type of architectural model in Booch's handbook of software architecture (Booch 2014).

Block diagrams present a high-level picture of the system structure, which people from different disciplines, who are involved in the system development process, can readily understand. In spite of their widespread use, Bass et al. (Bass, Clements, and Kazman 2012) dislike informal block diagrams for describing an architecture. They claim that these informal diagrams are poor architectural representations, as they show neither the type of the relationships among system components nor the components' externally visible properties.

The apparent contradictions between architectural theory and industrial practice arise because there are two ways in which an architectural model of a program is used:

1. *As a way of encouraging discussions about the system design* A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail. The architectural model identifies the key components that are to be developed so that managers can start assigning people to plan the development of these systems.

2. *As a way of documenting an architecture that has been designed* The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections. The argument for such a model is that such a detailed architectural description makes it easier to understand and evolve the system.

Block diagrams are a good way of supporting communications between the people involved in the software design process. They are intuitive, and domain experts and software engineers can relate to them and participate in discussions about the system. Managers find them helpful in planning the project. For many projects, block diagrams are the only architectural description.

Ideally, if the architecture of a system is to be documented in detail, it is better to use a more rigorous notation for architectural description. Various architectural description languages (Bass, Clements, and Kazman 2012) have been developed for this purpose. A more detailed and complete description means that there is less scope for misunderstanding the relationships between the architectural components. However, developing a detailed architectural description is an expensive and time-consuming process. It is practically impossible to know whether or not it is cost-effective, so this approach is not widely used.

Figure 6.2 Architectural design decisions

6.1 Architectural design decisions

Architectural design is a creative process in which you design a system organization that will satisfy the functional and non-functional requirements of a system. There is no formulaic architectural design process. It depends on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system. Consequently, I think it is best to consider architectural design as a series of decisions to be made rather than a sequence of activities.

During the architectural design process, system architects have to make a number of structural decisions that profoundly affect the system and its development process. Based on their knowledge and experience, they have to consider the fundamental questions shown in Figure 6.2.

Although each software system is unique, systems in the same application domain often have similar architectures that reflect the fundamental concepts of the domain. For example, application product lines are applications that are built around a core architecture with variants that satisfy specific customer requirements. When designing a system architecture, you have to decide what your system and broader application classes have in common, and decide how much knowledge from these application architectures you can reuse.

For embedded systems and apps designed for personal computers and mobile devices, you do not have to design a distributed architecture for the system. However, most large systems are distributed systems in which the system software is distributed across many different computers. The choice of distribution architecture is a

key decision that affects the performance and reliability of the system. This is a major topic in its own right that I cover in Chapter 17.

The architecture of a software system may be based on a particular Architectural pattern or style (these terms have come to mean the same thing). An Architectural pattern is a description of a system organization (Garlan and Shaw 1993), such as a client–server organization or a layered architecture. Architectural patterns capture the essence of an architecture that has been used in different software systems. You should be aware of common patterns, where they can be used, and their strengths and weaknesses when making decisions about the architecture of a system. I cover several frequently used patterns in Section 6.3.

Garlan and Shaw's notion of an architectural style covers questions 4 to 6 in the list of fundamental architectural questions shown in Figure 6.2. You have to choose the most appropriate structure, such as client–server or layered structuring, that will enable you to meet the system requirements. To decompose structural system units, you decide on a strategy for decomposing components into subcomponents. Finally, in the control modeling process, you develop a general model of the control relationships between the various parts of the system and make decisions about how the execution of components is controlled.

Because of the close relationship between non-functional system characteristics and software architecture, the choice of architectural style and structure should depend on the non-functional requirements of the system:

1. *Performance* If performance is a critical requirement, the architecture should be designed to localize critical operations within a small number of components, with these components deployed on the same computer rather than distributed across the network. This may mean using a few relatively large components rather than small, finer-grain components. Using large components reduces the number of component communications, as most of the interactions between related system features take place within a component. You may also consider runtime system organizations that allow the system to be replicated and executed on different processors.

2. *Security* If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers and a high level of security validation applied to these layers.

3. *Safety* If safety is a critical requirement, the architecture should be designed so that safety-related operations are co-located in a single component or in a small number of components. This reduces the costs and problems of safety validation and may make it possible to provide related protection systems that can safely shut down the system in the event of failure.

4. *Availability* If availability is a critical requirement, the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system. I describe fault-tolerant system architectures for high-availability systems in Chapter 11.

5. *Maintainability* If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may readily be changed. Producers of data should be separated from consumers, and shared data structures should be avoided.

Obviously, there is potential conflict between some of these architectures. For example, using large components improves performance, and using small, fine-grain components improves maintainability. If both performance and maintainability are important system requirements, however, then some compromise must be found. You can sometimes do this by using different Architectural patterns or styles for separate parts of the system. Security is now almost always a critical requirement, and you have to design an architecture that maintains security while also satisfying other non-functional requirements.

Evaluating an architectural design is difficult because the true test of an architecture is how well the system meets its functional and non-functional requirements when it is in use. However, you can do some evaluation by comparing your design against reference architectures or generic Architectural patterns. Bosch's description (Bosch 2000) of the non-functional characteristics of some Architectural patterns can help with architectural evaluation.

6.2 Architectural views

I explained in the introduction to this chapter that architectural models of a software system can be used to focus discussion about the software requirements or design. Alternatively, they may be used to document a design so that it can be used as a basis for more detailed design and implementation of the system. In this section, I discuss two issues that are relevant to both of these:

1. What views or perspectives are useful when designing and documenting a system's architecture?
2. What notations should be used for describing architectural models?

It is impossible to represent all relevant information about a system's architecture in a single diagram, as a graphical model can only show one view or perspective of the system. It might show how a system is decomposed into modules, how the runtime processes interact, or the different ways in which system components are distributed across a network. Because all of these are useful at different times, for both design and documentation, you usually need to present multiple views of the software architecture.

There are different opinions as to what views are required. Krutchen (Krutchen 1995) in his well-known 4+1 view model of software architecture, suggests that there should

Figure 6.3 Architectural views

be four fundamental architectural views, which can be linked through common use cases or scenarios (Figure 6.3). He suggests the following views:

1. *A logical view,* which shows the key abstractions in the system as objects or object classes. It should be possible to relate the system requirements to entities in this logical view.
2. *A process view,* which shows how, at runtime, the system is composed of interacting processes. This view is useful for making judgments about non-functional system characteristics such as performance and availability.
3. *A development view,* which shows how the software is decomposed for development; that is, it shows the breakdown of the software into components that are implemented by a single developer or development team. This view is useful for software managers and programmers.
4. *A physical view,* which shows the system hardware and how software components are distributed across the processors in the system. This view is useful for systems engineers planning a system deployment.

Hofmeister et al. (Hofmeister, Nord, and Soni 2000) suggest the use of similar views but add to this the notion of a conceptual view. This view is an abstract view of the system that can be the basis for decomposing high-level requirements into more detailed specifications, help engineers make decisions about components that can be reused, and represent a product line (discussed in Chapter 15) rather than a single system. Figure 6.1, which describes the architecture of a packing robot, is an example of a conceptual system view.

In practice, conceptual views of a system's architecture are almost always developed during the design process. They are used to explain the system architecture to stakeholders and to inform architectural decision making. During the design process, some of the other views may also be developed when different aspects of the system are discussed, but it is rarely necessary to develop a complete description from all perspectives. It may also be possible to associate Architectural patterns, discussed in the next section, with the different views of a system.

There are differing views about whether or not software architects should use the UML for describing and documenting software architectures. A survey in 2006 (Lange, Chaudron, and Muskens 2006) showed that, when the UML was used, it was mostly applied in an informal way. The authors of that paper argued that this was a bad thing.

I disagree with this view. The UML was designed for describing object-oriented systems, and, at the architectural design stage, you often want to describe systems at a higher level of abstraction. Object classes are too close to the implementation to be useful for architectural description. I don't find the UML to be useful during the design process itself and prefer informal notations that are quicker to write and that can be easily drawn on a whiteboard. The UML is of most value when you are documenting an architecture in detail or using model-driven development, as discussed in Chapter 5.

A number of researchers (Bass, Clements, and Kazman 2012) have proposed the use of more specialized architectural description languages (ADLs) to describe system architectures. The basic elements of ADLs are components and connectors, and they include rules and guidelines for well-formed architectures. However, because ADLs are specialist languages, domain and application specialists find it hard to understand and use ADLs. There may be some value in using domain-specific ADLs as part of model-driven development, but I do not think they will become part of mainstream software engineering practice. Informal models and notations, such as the UML, will remain the most commonly used ways of documenting system architectures.

Users of agile methods claim that detailed design documentation is mostly unused. It is, therefore, a waste of time and money to develop these documents. I largely agree with this view, and I think that, except for critical systems, it is not worth developing a detailed architectural description from Krutchen's four perspectives. You should develop the views that are useful for communication and not worry about whether or not your architectural documentation is complete.

6.3 Architectural patterns

The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems has been adopted in a number of areas of software engineering. The trigger for this was the publication of a book on object-oriented design patterns (Gamma et al. 1995). This prompted the development of other types of patterns, such as patterns for organizational design (Coplien and Harrison 2004), usability patterns (Usability Group 1998), patterns of cooperative interaction (Martin and Sommerville 2004), and configuration management patterns (Berczuk and Appleton 2002).

Architectural patterns were proposed in the 1990s under the name "architectural styles" (Shaw and Garlan 1996). A very detailed five-volume series of handbooks on pattern-oriented software architecture was published between 1996 and 2007 (Buschmann et al. 1996; Schmidt et al. 2000; Buschmann, Henney, and Schmidt 2007a, 2007b; Kircher and Jain 2004).

In this section, I introduce Architectural patterns and briefly describe a selection of Architectural patterns that are commonly used. Patterns may be described in a standard way (Figures 6.4 and 6.5) using a mixture of narrative description and diagrams.

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.5.
Example	Figure 6.6 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways, with changes made in one representation shown in all of them.
Disadvantages	May involve additional code and code complexity when the data model and interactions are simple.

Figure 6.4 The Model-View-Controller (MVC) pattern

For more detailed information about patterns and their use, you should refer to the published pattern handbooks.

You can think of an Architectural pattern as a stylized, abstract description of good practice, which has been tried and tested in different systems and environments. So, an Architectural pattern should describe a system organization that has been successful in previous systems. It should include information on when it is and is not appropriate to use that pattern, and details on the pattern's strengths and weaknesses.

Figure 6.4 describes the well-known Model-View-Controller pattern. This pattern is the basis of interaction management in many web-based systems and is supported by most language frameworks. The stylized pattern description includes the pattern

Figure 6.5 The organization of the Model-View-Controller

Figure 6.6 Web application architecture using the MVC pattern

name, a brief description, a graphical model, and an example of the type of system where the pattern is used. You should also include information about when the pattern should be used and its advantages and disadvantages.

Graphical models of the architecture associated with the MVC pattern are shown in Figures 6.5 and 6.6. These present the architecture from different views: Figure 6.5 is a conceptual view, and Figure 6.6 shows a runtime system architecture when this pattern is used for interaction management in a web-based system.

In this short space, it is impossible to describe all of the generic patterns that can be used in software development. Instead, I present some selected examples of patterns that are widely used and that capture good architectural design principles.

6.3.1 Layered architecture

The notions of separation and independence are fundamental to architectural design because they allow changes to be localized. The MVC pattern, shown in Figure 6.4, separates elements of a system, allowing them to change independently. For example, adding a new view or changing an existing view can be done without any changes to the underlying data in the model. The Layered Architecture pattern is another way of achieving separation and independence. This pattern is shown in Figure 6.7. Here, the system functionality is organized into separate layers, and each layer only relies on the facilities and services offered by the layer immediately beneath it.

This layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users. The architecture is also changeable and portable. If its interface is unchanged, a new layer with extended functionality can replace an existing layer

Name	Layered architecture
Description	Organizes the system into layers, with related functionality associated with each layer. A layer provides services to the layer above it, so the lowest level layers represent core services that are likely to be used throughout the system. See Figure 6.8.
Example	A layered model of a digital learning system to support learning of all subjects in schools (Figure 6.9).
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multilevel security.
Advantages	Allows replacement of entire layers as long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult, and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

Figure 6.7 The Layered Architecture pattern

without changing other parts of the system. Furthermore, when layer interfaces change or new facilities are added to a layer, only the adjacent layer is affected. As layered systems localize machine dependencies, this makes it easier to provide multi-platform implementations of an application system. Only the machine-dependent layers need be reimplemented to take account of the facilities of a different operating system or database.

Figure 6.8 is an example of a layered architecture with four layers. The lowest layer includes system support software—typically, database and operating system support. The next layer is the application layer, which includes the components concerned with the application functionality and utility components used by other application components.

The third layer is concerned with user interface management and providing user authentication and authorization, with the top layer providing user interface facilities. Of course, the number of layers is arbitrary. Any of the layers in Figure 6.6 could be split into two or more layers.

Figure 6.8 A generic layered architecture

Figure 6.9 The architecture of the iLearn system

Figure 6.9 shows that the iLearn digital learning system, introduced in Chapter 1, has a four-layer architecture that follows this pattern. You can see another example of the Layered Architecture pattern in Figure 6.19 (Section 6.4, which shows the organization of the Mentcare system.

6.3.2 Repository architecture

The layered architecture and MVC patterns are examples of patterns where the view presented is the conceptual organization of a system. My next example, the Repository pattern (Figure 6.10), describes how a set of interacting components can share data.

Figure 6.10 The Repository pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.11 is an example of an IDE where the components use a repository of system design information. Each software tool generates information, which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent; they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

Figure 6.11 A repository architecture for an IDE

The majority of systems that use large amounts of data are organized around a shared database or repository. This model is therefore suited to applications in which data is generated by one component and used by another. Examples of this type of system include command and control systems, management information systems, Computer-Aided Design (CAD) systems, and interactive development environments for software.

Figure 6.11 illustrates a situation in which a repository might be used. This diagram shows an IDE that includes different tools to support model-driven development. The repository in this case might be a version-controlled environment (as discussed in Chapter 25) that keeps track of changes to software and allows rollback to earlier versions.

Organizing tools around a repository is an efficient way of sharing large amounts of data. There is no need to transmit data explicitly from one component to another. However, components must operate around an agreed repository data model. Inevitably, this is a compromise between the specific needs of each tool, and it may be difficult or impossible to integrate new components if their data models do not fit the agreed schema. In practice, it may be difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralized repository, this involves maintaining multiple copies of data. Keeping these consistent and up to date adds more overhead to the system.

In the repository architecture shown in Figure 6.11, the repository is passive and control is the responsibility of the components using the repository. An alternative approach, which has been derived for artificial intelligence (AI) systems, uses a "blackboard" model that triggers components when particular data become available. This is appropriate when the data in the repository is unstructured. Decisions about which tool is to be activated can only be made when the data has been analyzed. This model was introduced by Nii (Nii 1986), and Bosch (Bosch 2000) includes a good discussion of how this style relates to system quality attributes.

6.3.3 Client–server architecture

The Repository pattern is concerned with the static structure of a system and does not show its runtime organization. My next example, the Client–Server pattern (Figure 6.12), illustrates a commonly used runtime organization for distributed

Name	Client–server
Description	In a client–server architecture, the system is presented as a set of services, with each service delivered by a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.13 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure and so is susceptible to denial-of-service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. Management problems may arise if servers are owned by different organizations.

Figure 6.12 The Client–Server pattern

systems. A system that follows the Client–Server pattern is organized as a set of services and associated servers, and clients that access and use the services. The major components of this model are:

1. A set of servers that offer services to other components. Examples of servers include print servers that offer printing services, file servers that offer file management services, and a compile server that offers programming language compilation services. Servers are software components, and several servers may run on the same computer.
2. A set of clients that call on the services offered by servers. There will normally be several instances of a client program executing concurrently on different computers.
3. A network that allows the clients to access these services. Client–server systems are usually implemented as distributed systems, connected using Internet protocols.

Client–server architectures are usually thought of as distributed systems architectures, but the logical model of independent services running on separate servers can be implemented on a single computer. Again, an important benefit is separation and independence. Services and servers can be changed without affecting other parts of the system.

Clients may have to know the names of the available servers and the services they provide. However, servers do not need to know the identity of clients or how many clients are accessing their services. Clients access the services provided by a server through remote procedure calls using a request–reply protocol (such as http), where a client makes a request to a server and waits until it receives a reply from that server.

Figure 6.13 A client–server architecture for a film library

Figure 6.13 is an example of a system that is based on the client–server model. This is a multiuser, web-based system for providing a film and photograph library. In this system, several servers manage and display the different types of media. Video frames need to be transmitted quickly and in synchrony but at relatively low resolution. They may be compressed in a store, so the video server can handle video compression and decompression in different formats. Still pictures, however, must be maintained at a high resolution, so it is appropriate to maintain them on a separate server.

The catalog must be able to deal with a variety of queries and provide links into the web information system that include data about the film and video clips, and an e-commerce system that supports the sale of photographs, film, and video clips. The client program is simply an integrated user interface, constructed using a web browser, to access these services.

The most important advantage of the client–server model is that it is a distributed architecture. Effective use can be made of networked systems with many distributed processors. It is easy to add a new server and integrate it with the rest of the system or to upgrade servers transparently without affecting other parts of the system. I cover distributed architectures in Chapter 17, where I explain the client–server model and its variants in more detail.

6.3.4 Pipe and filter architecture

My final example of a general Architectural pattern is the Pipe and Filter pattern (Figure 6.14). This is a model of the runtime organization of a system where functional transformations process their inputs and produce outputs. Data flows from one to another and is transformed as it moves through the sequence. Each processing step is implemented as a transform. Input data flows through these transforms until converted to output. The transformations may execute sequentially or in parallel. The data can be processed by each transform item by item or in a single batch.

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.15 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data-processing applications (both batch and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse architectural components that use incompatible data structures.

Figure 6.14 The Pipe and Filter pattern

The name "pipe and filter" comes from the original Unix system where it was possible to link processes using "pipes." These passed a text stream from one process to another. Systems that conform to this model can be implemented by combining Unix commands, using pipes and the control facilities of the Unix shell. The term *filter* is used because a transformation "filters out" the data it can process from its input data stream.

Variants of this pattern have been in use since computers were first used for automatic data processing. When transformations are sequential with data processed in batches, this pipe and filter architectural model becomes a batch sequential model, a common architecture for data-processing systems such as billing systems. The architecture of an embedded system may also be organized as a process pipeline, with each process executing concurrently. I cover use of this pattern in embedded systems in Chapter 21.

An example of this type of system architecture, used in a batch processing application, is shown in Figure 6.15. An organization has issued invoices to customers. Once a week, payments that have been made are reconciled with the invoices. For

Figure 6.15 An example of the pipe and filter architecture

Architectural patterns for control

There are specific Architectural patterns that reflect commonly used ways of organizing control in a system. These include centralized control, based on one component calling other components, and event-based control, where the system reacts to external events.

http://software-engineering-book.com/web/archpatterns/

those invoices that have been paid, a receipt is issued. For those invoices that have not been paid within the allowed payment time, a reminder is issued.

Pipe and filter systems are best suited to batch processing systems and embedded systems where there is limited user interaction. Interactive systems are difficult to write using the pipe and filter model because of the need for a stream of data to be processed. While simple textual input and output can be modeled in this way, graphical user interfaces have more complex I/O formats and a control strategy that is based on events such as mouse clicks or menu selections. It is difficult to implement this as a sequential stream that conforms to the pipe and filter model.

6.4 Application architectures

Application systems are intended to meet a business or an organizational need. All businesses have much in common—they need to hire people, issue invoices, keep accounts, and so on. Businesses operating in the same sector use common sector-specific applications. Therefore, as well as general business functions, all phone companies need systems to connect and meter calls, manage their network and issue bills to customers. Consequently, the application systems used by these businesses also have much in common.

These commonalities have led to the development of software architectures that describe the structure and organization of particular types of software systems. Application architectures encapsulate the principal characteristics of a class of systems. For example, in real-time systems, there might be generic architectural models of different system types, such as data collection systems or monitoring systems. Although instances of these systems differ in detail, the common architectural structure can be reused when developing new systems of the same type.

The application architecture may be reimplemented when developing new systems. However, for many business systems, application architecture reuse is implicit when generic application systems are configured to create a new application. We see this in the widespread use of Enterprise Resource Planning (ERP) systems and off-the-shelf configurable application systems, such as systems for accounting and stock control. These systems have a standard architecture and components. The components are configured and adapted to create a specific business application.

Application architectures

There are several examples of application architectures on the book's website. These include descriptions of batch data-processing systems, resource allocation systems, and event-based editing systems.

http://software-engineering-book.com/web/apparch/

For example, a system for supply chain management can be adapted for different types of suppliers, goods, and contractual arrangements.

As a software designer, you can use models of application architectures in a number of ways:

1. *As a starting point for the architectural design process* If you are unfamiliar with the type of application that you are developing, you can base your initial design on a generic application architecture. You then specialize this for the specific system that is being developed.
2. *As a design checklist* If you have developed an architectural design for an application system, you can compare this with the generic application architecture. You can check that your design is consistent with the generic architecture.
3. *As a way of organizing the work of the development team* The application architectures identify stable structural features of the system architectures, and in many cases, it is possible to develop these in parallel. You can assign work to group members to implement different components within the architecture.
4. *As a means of assessing components for reuse* If you have components you might be able to reuse, you can compare these with the generic structures to see whether there are comparable components in the application architecture.
5. *As a vocabulary for talking about applications* If you are discussing a specific application or trying to compare applications, then you can use the concepts identified in the generic architecture to talk about these applications.

There are many types of application system, and, in some cases, they may seem to be very different. However, superficially dissimilar applications may have much in common and thus share an abstract application architecture. I illustrate this by describing the architectures of two types of application:

1. *Transaction processing applications* Transaction processing applications are database-centered applications that process user requests for information and update the information in a database. These are the most common types of interactive business systems. They are organized in such a way that user actions can't interfere with each other and the integrity of the database is maintained. This class of system includes interactive banking systems, e-commerce systems, information systems, and booking systems.

Figure 6.16 The structure of transaction processing applications

2. *Language processing systems* Language processing systems are systems in which the user's intentions are expressed in a formal language, such as a programming language. The language processing system processes this language into an internal format and then interprets this internal representation. The best-known language processing systems are compilers, which translate high-level language programs into machine code. However, language processing systems are also used to interpret command languages for databases and information systems, and markup languages such as XML.

I have chosen these particular types of system because a large number of web-based business systems are transaction processing systems, and all software development relies on language processing systems.

6.4.1 Transaction processing systems

Transaction processing systems are designed to process user requests for information from a database, or requests to update a database (Lewis, Bernstein, and Kifer 2003). Technically, a database transaction is part of a sequence of operations and is treated as a single unit (an atomic unit). All of the operations in a transaction have to be completed before the database changes are made permanent. This ensures that failure of operations within a transaction does not lead to inconsistencies in the database.

From a user perspective, a transaction is any coherent sequence of operations that satisfies a goal, such as "find the times of flights from London to Paris." If the user transaction does not require the database to be changed, then it may not be necessary to package this as a technical database transaction.

An example of a database transaction is a customer request to withdraw money from a bank account using an ATM. This involves checking the customer account balance to see if sufficient funds are available, modifying the balance by the amount withdrawn and sending commands to the ATM to deliver the cash. Until all of these steps have been completed, the transaction is incomplete and the customer accounts database is not changed.

Transaction processing systems are usually interactive systems in which users make asynchronous requests for service. Figure 6.16 illustrates the conceptual architectural structure of transaction processing applications. First, a user makes a request to the system through an I/O processing component. The request is processed by some application-specific logic. A transaction is created and passed to a transaction manager, which is usually embedded in the database management system. After the transaction manager has ensured that the transaction is properly completed, it signals to the application that processing has finished.

Transaction processing systems may be organized as a "pipe and filter" architecture, with system components responsible for input, processing, and output. For

Figure 6.17 The software architecture of an ATM system

example, consider a banking system that allows customers to query their accounts and withdraw cash from an ATM. The system is composed of two cooperating software components—the ATM software and the account processing software in the bank's database server. The input and output components are implemented as software in the ATM, and the processing component is part of the bank's database server. Figure 6.17 shows the architecture of this system, illustrating the functions of the input, process, and output components.

6.4.2 Information systems

All systems that involve interaction with a shared database can be considered to be transaction-based information systems. An information system allows controlled access to a large base of information, such as a library catalog, a flight timetable, or the records of patients in a hospital. Information systems are almost always web-based systems, where the user interface is implemented in a web browser.

Figure 6.18 presents a very general model of an information system. The system is modeled using a layered approach (discussed in Section 6.3) where the top layer

Figure 6.18 Layered information system architecture

Figure 6.19 The architecture of the Mentcare system

supports the user interface and the bottom layer is the system database. The user communications layer handles all input and output from the user interface, and the information retrieval layer includes application-specific logic for accessing and updating the database. The layers in this model can map directly onto servers in a distributed Internet-based system.

As an example of an instantiation of this layered model, Figure 6.19 shows the architecture of the Mentcare system. Recall that this system maintains and manages details of patients who are consulting specialist doctors about mental health problems. I have added detail to each layer in the model by identifying the components that support user communications and information retrieval and access:

1. The top layer is a browser-based user interface.
2. The second layer provides the user interface functionality that is delivered through the web browser. It includes components to allow users to log in to the system and checking components that ensure that the operations they use are allowed by their role. This layer includes form and menu management components that present information to users, and data validation components that check information consistency.
3. The third layer implements the functionality of the system and provides components that implement system security, patient information creation and updating, import and export of patient data from other databases, and report generators that create management reports.
4. Finally, the lowest layer, which is built using a commercial database management system, provides transaction management and persistent data storage.

Information and resource management systems are sometimes also transaction processing systems. For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer. In an e-commerce

Figure 6.20 The architecture of a language processing system

system, the application-specific layer includes additional functionality supporting a "shopping cart" in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

The organization of servers in these systems usually reflects the four-layer generic model presented in Figure 6.18. These systems are often implemented as distributed systems with a multitier client server/architecture

1. The web server is responsible for all user communications, with the user interface implemented using a web browser;
2. The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests;
3. The database server moves information to and from the database and handles transaction management.

Using multiple servers allows high throughput and makes it possible to handle thousands of transactions per minute. As demand increases, servers can be added at each level to cope with the extra processing involved.

6.4.3 Language processing systems

Language processing systems translate one language into an alternative representation of that language and, for programming languages, may also execute the resulting code. Compilers translate a programming language into machine code. Other language processing systems may translate an XML data description into commands to query a database or to an alternative XML representation. Natural language processing systems may translate one natural language to another, for example, French to Norwegian.

A possible architecture for a language processing system for a programming language is illustrated in Figure 6.20. The source language instructions define the

Figure 6.21 A repository architecture for a language processing system

program to be executed, and a translator converts these into instructions for an abstract machine. These instructions are then interpreted by another component that fetches the instructions for execution and executes them using (if necessary) data from the environment. The output of the process is the result of interpreting the instructions on the input data.

For many compilers, the interpreter is the system hardware that processes machine instructions, and the abstract machine is a real processor. However, for dynamically typed languages, such as Ruby or Python, the interpreter is a software component.

Programming language compilers that are part of a more general programming environment have a generic architecture (Figure 6.21) that includes the following components:

1. A lexical analyzer, which takes input language tokens and converts them into an internal form.
2. A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
3. A syntax analyzer, which checks the syntax of the language being translated. It uses a defined grammar of the language and builds a syntax tree.
4. A syntax tree, which is an internal structure representing the program being compiled.
5. A semantic analyzer, which uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
6. A code generator, which "walks" the syntax tree and generates abstract machine code.

Other components might also be included that analyze and transform the syntax tree to improve efficiency and remove redundancy from the generated machine code.

Reference architectures

Reference architectures capture important features of system architectures in a domain. Essentially, they include everything that might be in an application architecture, although, in reality, it is very unlikely that any individual application would include all the features shown in a reference architecture. The main purpose of reference architectures is to evaluate and compare design proposals, and to educate people about architectural characteristics in that domain.

http://software-engineering-book.com/web/refarch/

In other types of language processing system, such as a natural language translator, there will be additional components such as a dictionary. The output of the system is translation of the input text.

Figure 6.21 illustrates how a language processing system can be part of an integrated set of programming support tools. In this example, the symbol table and syntax tree act as a central information repository. Tools or tool fragments communicate through it. Other information that is sometimes embedded in tools, such as the grammar definition and the definition of the output format for the program, have been taken out of the tools and put into the repository. Therefore, a syntax-directed editor can check that the syntax of a program is correct as it is being typed. A program formatter can create listings of the program that highlight different syntactic elements and are therefore easier to read and understand.

Alternative Architectural patterns may be used in a language processing system (Garlan and Shaw 1993). Compilers can be implemented using a composite of a repository and a pipe and filter model. In a compiler architecture, the symbol table is a repository for shared data. The phases of lexical, syntactic, and semantic analysis are organized sequentially, as shown in Figure 6.22, and communicate through the shared symbol table.

This pipe and filter model of language compilation is effective in batch environments where programs are compiled and executed without user interaction; for example, in the translation of one XML document to another. It is less effective when a compiler is integrated with other language processing tools such as a structured editing system, an interactive debugger, or a program formatter. In this situation, changes from one component need to be reflected immediately in other components. It is better to organize the system around a repository, as shown in Figure 6.21 if you are implementing a general, language-oriented programming environment.

Figure 6.22 A pipe and filter compiler architecture

KEY POINTS

- A software architecture is a description of how a software system is organized. Properties of a system such as performance, security, and availability are influenced by the architecture used.
- Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used, and the ways in which the architecture should be documented and evaluated.
- Architectures may be documented from several different perspectives or views. Possible views include a conceptual view, a logical view, a process view, a development view, and a physical view.
- Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used, and point out its advantages and disadvantages.
- Commonly used Architectural patterns include model-view-controller, layered architecture, repository, client–server, and pipe and filter.
- Generic models of application systems architectures help us understand the operation of applications, compare applications of the same type, validate application system designs, and assess large-scale components for reuse.
- Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users. Information systems and resource management systems are examples of transaction processing systems.
- Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.

FURTHER READING

Software Architecture: Perspectives on an Emerging Discipline. This was the first book on software architecture and has a good discussion on different architectural styles that is still relevant. (M. Shaw and D. Garlan, 1996, Prentice-Hall).

"The Golden Age of Software Architecture." This paper surveys the development of software architecture from its beginnings in the 1980s through to its usage in the 21st century. There is not a lot of technical content, but it is an interesting historical overview. (M. Shaw and P. Clements, *IEEE Software,* 21 (2), March–April 2006) http://doi.dx.org/10.1109/MS.2006.58.

Software Architecture in Practice (3rd ed.). This is a practical discussion of software architectures that does not oversell the benefits of architectural design. It provides a clear business rationale, explaining why architectures are important. (L. Bass, P. Clements, and R. Kazman, 2012, Addison-Wesley).

Handbook of Software Architecture. This is a work in progress by Grady Booch, one of the early evangelists for software architecture. He has been documenting the architectures of a range of software systems so that you can see reality rather than academic abstraction. Available on the web and intended to appear as a book. (G. Booch, 2014) http://www.handbookofsoftwarearchitecture.com/

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/requirements-and-design/

EXERCISES

6.1. When describing a system, explain why you may have to start the design of the system architecture before the requirements specification is complete.

6.2. You have been asked to prepare and deliver a presentation to a nontechnical manager to justify the hiring of a system architect for a new project. Write a list of bullet points setting out the key points in your presentation in which you explain the importance of software architecture.

6.3. Performance and security may pose to be conflicting non-functional requirements when architecting software systems. Make an argument in support of this statement.

6.4. Draw diagrams showing a conceptual view and a process view of the architectures of the following systems:

A ticket machine used by passengers at a railway station.

A computer-controlled video conferencing system that allows video, audio, and computer data to be visible to several participants at the same time.

A robot floor-cleaner that is intended to clean relatively clear spaces such as corridors. The cleaner must be able to sense walls and other obstructions.

6.5. A software system will be built to allow drones to autonomously herd cattle in farms. These drones can be remotely controlled by human operators. Explain how multiple architectural patterns can fit together to help build this kind of system.

6.6. Suggest an architecture for a system (such as iTunes) that is used to sell and distribute music on the Internet. What Architectural patterns are the basis for your proposed architecture?

6.7. An information system is to be developed to maintain information about assets owned by a utility company such as buildings, vehicles, and equipment. It is intended that this will be

updatable by staff working in the field using mobile devices as new asset information becomes available. The company has several existing asset databases that should be integrated through this system. Design a layered architecture for this asset management system based on the generic information system architecture shown in Figure 6.18.

6.8. Using the generic model of a language processing system presented here, design the architecture of a system that accepts natural language commands and translates these into database queries in a language such as SQL.

6.9. Using the basic model of an information system, as presented in Figure 6.18, suggest the components that might be part of an information system that allows users to view box office events, available tickets and prices, and to eventually buy tickets.

6.10. Should there be a separate profession of 'software architect' whose role is to work independently with a customer to design the software system architecture? A separate software company would then implement the system. What might be the difficulties of establishing such a profession?

REFERENCES

Bass, L., P. Clements, and R. Kazman. 2012. *Software Architecture in Practice (3rd ed.)*. Boston: Addison-Wesley.

Berczuk, S. P., and B. Appleton. 2002. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Boston: Addison-Wesley.

Booch, G. 2014. "Handbook of Software Architecture." http://handbookofsoftwarearchitecture.com/

Bosch, J. 2000. *Design and Use of Software Architectures*. Harlow, UK: Addison-Wesley.

Buschmann, F., K. Henney, and D. C. Schmidt. 2007a. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. New York: John Wiley & Sons.

———. 2007b. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. New York: John Wiley & Sons.

Buschmann, F., R. Meunier, H. Rohnert, and P. Sommerlad. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. New York: John Wiley & Sons.

Chen, L., M. Ali Babar, and B. Nuseibeh. 2013. "Characterizing Architecturally Significant Requirements." *IEEE Software* 30 (2): 38–45. doi:10.1109/MS.2012.174.

Coplien, J. O., and N. B. Harrison. 2004. *Organizational Patterns of Agile Software Development*. Englewood Cliffs, NJ: Prentice-Hall.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Garlan, D., and M. Shaw. 1993. "An Introduction to Software Architecture." In *Advances in Software Engineering and Knowledge Engineering*, edited by V. Ambriola and G. Tortora, 2:1–39. London: World Scientific Publishing Co.

Hofmeister, C., R. Nord, and D. Soni. 2000. *Applied Software Architecture*. Boston: Addison-Wesley.

Kircher, M., and P. Jain. 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. New York: John Wiley & Sons.

Krutchen, P. 1995. "The 4+1 View Model of Software Architecture." *IEEE Software* 12 (6): 42–50. doi:10.1109/52.469759.

Lange, C. F. J., M. R. V. Chaudron, and J. Muskens. 2006. "UML Software Architecture and Design Description." *IEEE Software* 23 (2): 40–46. doi:10.1109/MS.2006.50.

Lewis, P. M., A. J. Bernstein, and M. Kifer. 2003. *Databases and Transaction Processing: An Application-Oriented Approach*. Boston: Addison-Wesley.

Martin, D., and I. Sommerville. 2004. "Patterns of Cooperative Interaction: Linking Ethnomethodology and Design." *ACM Transactions on Computer-Human Interaction* 11 (1) (March 1): 59–89. doi:10.1145/972648.972651.

Nii, H. P. 1986. "Blackboard Systems, Parts 1 and 2." *AI Magazine* 7 (2 and 3): 38–53 and 62–69. http://www.aaai.org/ojs/index.php/aimagazine/article/view/537/473

Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann. 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons.

Shaw, M., and D. Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice-Hall.

Usability Group. 1998. "Usability Patterns". University of Brighton. http://www.it.bton.ac.uk/Research/patterns/home.html

7

Design and implementation

Objectives

The objectives of this chapter are to introduce object-oriented software design using the UML and highlight important implementation concerns. When you have read this chapter, you will:

- understand the most important activities in a general, object-oriented design process;
- understand some of the different models that may be used to document an object-oriented design;
- know about the idea of design patterns and how these are a way of reusing design knowledge and experience;
- have been introduced to key issues that have to be considered when implementing software, including software reuse and open-source development.

Contents

7.1 Object-oriented design using the UML
7.2 Design patterns
7.3 Implementation issues
7.4 Open-source development

Software design and implementation is the stage in the software engineering process at which an executable software system is developed. For some simple systems, software engineering means software design and implementation and all other software engineering activities are merged with this process. However, for large systems, software design and implementation is only one of a number of software engineering processes (requirements engineering, verification and validation, etc.).

Software design and implementation activities are invariably interleaved. Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements. Implementation is the process of realizing the design as a program. Sometimes there is a separate design stage, and this design is modeled and documented. At other times, a design is in the programmer's head or roughly sketched on a whiteboard or sheets of paper. Design is about how to solve a problem, so there is always a design process. However, it isn't always necessary or appropriate to describe the design in detail using the UML or other design description language.

Design and implementation are closely linked, and you should normally take implementation issues into account when developing a design. For example, using the UML to document a design may be the right thing to do if you are programming in an object-oriented language such as Java or C#. It is less useful, I think, if you are developing using a dynamically typed language like Python. There is no point in using the UML if you are implementing your system by configuring an off-the-shelf package. As I discussed in Chapter 3, agile methods usually work from informal sketches of the design and leave design decisions to programmers.

One of the most important implementation decisions that has to be made at an early stage of a software project is whether to build or to buy the application software. For many types of application, it is now possible to buy off-the-shelf application systems that can be adapted and tailored to the users' requirements. For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It is usually cheaper and faster to use this approach rather than developing a new system in a conventional programming language.

When you develop an application system by reusing an off-the-shelf product, the design process focuses on how to configure the system product to meet the application requirements. You don't develop design models of the system, such as models of the system objects and their interactions. I discuss this reuse-based approach to development in Chapter 15.

I assume that most readers of this book have had experience of program design and implementation. This is something that you acquire as you learn to program and master the elements of a programming language like Java or Python. You will have probably learned about good programming practice in the programming languages that you have studied, as well as how to debug programs that you have developed. Therefore, I don't cover programming topics here. Instead, this chapter has two aims:

1. To show how system modeling and architectural design (covered in Chapters 5 and 6) are put into practice in developing an object-oriented software design.

2. To introduce important implementation issues that are not usually covered in programming books. These include software reuse, configuration management and open-source development.

As there are a vast number of different development platforms, the chapter is not biased toward any particular programming language or implementation technology. Therefore, I have presented all examples using the UML rather than a programming language such as Java or Python.

7.1 Object-oriented design using the UML

An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. The representation of the state is private and cannot be accessed directly from outside the object. Object-oriented design processes involve designing object classes and the relationships between these classes. These classes define the objects in the system and their interactions. When the design is realized as an executing program, the objects are created dynamically from these class definitions.

Objects include both data and operations to manipulate that data. They may therefore be understood and modified as stand-alone entities. Changing the implementation of an object or adding services should not affect other system objects. Because objects are associated with things, there is often a clear mapping between real-world entities (such as hardware components) and their controlling objects in the system. This improves the understandability, and hence the maintainability, of the design.

To develop a system design from concept to detailed, object-oriented design, you need to:

1. Understand and define the context and the external interactions with the system.
2. Design the system architecture.
3. Identify the principal objects in the system.
4. Develop design models.
5. Specify interfaces.

Like all creative activities, design is not a clear-cut, sequential process. You develop a design by getting ideas, proposing solutions, and refining these solutions as information becomes available. You inevitably have to backtrack and retry when problems arise. Sometimes you explore options in detail to see if they work; at other times you ignore details until late in the process. Sometimes you use notations, such as the UML, precisely to clarify aspects of the design; at other times, notations are used informally to stimulate discussions.

I explain object-oriented software design by developing a design for part of the embedded software for the wilderness weather station that I introduced in Chapter 1. Wilderness weather stations are deployed in remote areas. Each weather station

Figure 7.1 System context for the weather station

records local weather information and periodically transfers this to a weather information system, using a satellite link.

7.1.1 System context and interactions

The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment. This is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment. As I discussed in Chapter 5, understanding the context also lets you establish the boundaries of the system.

Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems. In this case, you need to decide how functionality is distributed between the control system for all of the weather stations and the embedded software in the weather station itself.

System context models and interaction models present complementary views of the relationships between a system and its environment:

1. A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.

2. An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

The context model of a system may be represented using associations. Associations simply show that there are some relationships between the entities involved in the association. You can document the environment of the system using a simple block diagram, showing the entities in the system and their associations. Figure 7.1 shows that the systems in the environment of each weather station are a weather information system, an onboard satellite system, and a control system. The cardinality information on the link shows that there is a single control system but several weather stations, one satellite, and one general weather information system.

When you model the interactions of a system with its environment, you should use an abstract approach that does not include too much detail. One way to do this is to use a use case model. As I discussed in Chapters 4 and 5, each use case represents

Weather station use cases

Report weather–send weather data to the weather information system

Report status–send status information to the weather information system

Restart–if the weather station is shut down, restart the system

Shutdown–shut down the weather station

Reconfigure–reconfigure the weather station software

Powersave–put the weather station into power-saving mode

Remote control–send control commands to any weather station subsystem

http://software-engineering-book.com/web/ws-use-cases/

an interaction with the system. Each possible interaction is named in an ellipse, and the external entity involved in the interaction is represented by a stick figure.

The use case model for the weather station is shown in Figure 7.2. This shows that the weather station interacts with the weather information system to report weather data and the status of the weather station hardware. Other interactions are with a control system that can issue specific weather station control commands. The stick figure is used in the UML to represent other systems as well as human users.

Each of these use cases should be described in structured natural language. This helps designers identify objects in the system and gives them an understanding of what the system is intended to do. I use a standard format for this description that clearly identifies what information is exchanged, how the interaction is initiated, and so on. As I explain in Chapter 21, embedded systems are often modeled by describing

Figure 7.2 Weather station use cases

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Data	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum and average wind speeds; the total rainfall; and the wind direction as sampled at 5-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour, but this frequency may differ from one station to another and may be modified in future.

Figure 7.3 Use case description—Report weather

how they respond to internal or external stimuli. Therefore, the stimuli and associated responses should be listed in the description. Figure 7.3 shows the description of the Report weather use case from Figure 7.2 that is based on this approach.

7.1.2 Architectural design

Once the interactions between the software system and the system's environment have been defined, you use this information as a basis for designing the system architecture. Of course, you need to combine this knowledge with your general knowledge of the principles of architectural design and with more detailed domain knowledge. You identify the major components that make up the system and their interactions. You may then design the system organization using an architectural pattern such as a layered or client–server model.

The high-level architectural design for the weather station software is shown in Figure 7.4. The weather station is composed of independent subsystems that communicate

Figure 7.4 High-level architecture of weather station

Figure 7.5 Architecture of data collection system

by broadcasting messages on a common infrastructure, shown as **Communication** link in Figure 7.4. Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them. This "listener model" is a commonly used architectural style for distributed systems.

When the communications subsystem receives a control command, such as shutdown, the command is picked up by each of the other subsystems, which then shut themselves down in the correct way. The key benefit of this architecture is that it is easy to support different configurations of subsystems because the sender of a message does not need to address the message to a particular subsystem.

Figure 7.5 shows the architecture of the data collection subsystem, which is included in Figure 7.4. The **Transmitter** and **Receiver** objects are concerned with managing communications, and the **WeatherData** object encapsulates the information that is collected from the instruments and transmitted to the weather information system. This arrangement follows the producer–consumer pattern, discussed in Chapter 21.

7.1.3 Object class identification

By this stage in the design process, you should have some ideas about the essential objects in the system that you are designing. As your understanding of the design develops, you refine these ideas about the system objects. The use case description helps to identify objects and operations in the system. From the description of the Report weather use case, it is obvious that you will need to implement objects representing the instruments that collect weather data and an object representing the summary of the weather data. You also usually need a high-level system object or objects that encapsulate the system interactions defined in the use cases. With these objects in mind, you can start to identify the general object classes in the system.

As object-oriented design evolved in the 1980s, various ways of identifying object classes in object-oriented systems were suggested:

1. Use a grammatical analysis of a natural language description of the system to be constructed. Objects and attributes are nouns; operations or services are verbs (Abbott 1983).

2. Use tangible entities (things) in the application domain such as aircraft, roles such as manager, events such as request, interactions such as meetings, locations

Figure 7.6 Weather station objects

such as offices, organizational units such as companies, and so on (Wirfs-Brock, Wilkerson, and Weiner 1990).

3. Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn. As each scenario is analyzed, the team responsible for the analysis must identify the required objects, attributes, and operations (Beck and Cunningham 1989).

In practice, you have to use several knowledge sources to discover object classes. Object classes, attributes, and operations that are initially identified from the informal system description can be a starting point for the design. Information from application domain knowledge or scenario analysis may then be used to refine and extend the initial objects. This information can be collected from requirements documents, discussions with users, or analyses of existing systems. As well as the objects representing entities external to the system, you may also have to design "implementation objects" that are used to provide general services such as searching and validity checking.

In the wilderness weather station, object identification is based on the tangible hardware in the system. I don't have space to include all the system objects here, but I have shown five object classes in Figure 7.6. The Ground thermometer, Anemometer, and Barometer objects are application domain objects, and the WeatherStation and WeatherData objects have been identified from the system description and the scenario (use case) description:

1. The WeatherStation object class provides the basic interface of the weather station with its environment. Its operations are based on the interactions shown in Figure 7.3. I use a single object class, and it includes all of these interactions. Alternatively, you could design the system interface as several different classes, with one class per interaction.

2. The WeatherData object class is responsible for processing the report weather command. It sends the summarized data from the weather station instruments to the weather information system.

3. The Ground thermometer, Anemometer, and Barometer object classes are directly related to instruments in the system. They reflect tangible hardware entities in the system and the operations are concerned with controlling that hardware. These objects operate autonomously to collect data at the specified frequency and store the collected data locally. This data is delivered to the WeatherData object on request.

You use knowledge of the application domain to identify other objects, attributes. and services:

1. Weather stations are often located in remote places and include various instruments that sometimes go wrong. Instrument failures should be reported automatically. This implies that you need attributes and operations to check the correct functioning of the instruments.

2. There are many remote weather stations, so each weather station should have its own identifier so that it can be uniquely identified in communications.

3. As weather stations are installed at different times, the types of instrument may be different. Therefore, each instrument should also be uniquely identified, and a database of instrument information should be maintained.

At this stage in the design process, you should focus on the objects themselves, without thinking about how these objects might be implemented. Once you have identified the objects, you then refine the object design. You look for common features and then design the inheritance hierarchy for the system. For example, you may identify an Instrument superclass, which defines the common features of all instruments, such as an identifier, and get and test operations. You may also add new attributes and operations to the superclass, such as an attribute that records how often data should be collected.

7.1.4 Design models

Design or system models, as I discussed in Chapter 5, show the objects or object classes in a system. They also show the associations and relationships between these entities. These models are the bridge between the system requirements and the implementation of a system. They have to be abstract so that unnecessary detail doesn't hide the relationships between them and the system requirements. However, they also have to include enough detail for programmers to make implementation decisions.

The level of detail that you need in a design model depends on the design process used. Where there are close links between requirements engineers, designers and programmers, then abstract models may be all that are required. Specific design decisions may be made as the system is implemented, with problems resolved through informal discussions. Similarly, if agile development is used, outline design models on a whiteboard may be all that is required.

However, if a plan-based development process is used, you may need more detailed models. When the links between requirements engineers, designers, and programmers are indirect (e.g., where a system is being designed in one part of an organization but implemented elsewhere), then precise design descriptions are needed for communication. Detailed models, derived from the high-level abstract models, are used so that all team members have a common understanding of the design.

An important step in the design process, therefore, is to decide on the design models that you need and the level of detail required in these models. This depends on the type of system that is being developed. A sequential data-processing system is quite different from an embedded real-time system, so you need to use different types of design models. The UML supports 13 different types of models, but, as I discussed in Chapter 5, many of these models are not widely used. Minimizing the number of models that are produced reduces the costs of the design and the time required to complete the design process.

When you use the UML to develop a design, you should develop two kinds of design model:

1. *Structural models,* which describe the static structure of the system using object classes and their relationships. Important relationships that may be documented at this stage are generalization (inheritance) relationships, uses/used-by relationships, and composition relationships.
2. *Dynamic models,* which describe the dynamic structure of the system and show the expected runtime interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the state changes triggered by these object interactions.

I think three UML model types are particularly useful for adding detail to use case and architectural models:

1. *Subsystem models,* which show logical groupings of objects into coherent subsystems. These are represented using a form of class diagram with each subsystem shown as a package with enclosed objects. Subsystem models are structural models.
2. *Sequence models,* which show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic models.
3. *State machine models,* which show how objects change their state in response to events. These are represented in the UML using state diagrams. State machine models are dynamic models.

A subsystem model is a useful static model that shows how a design is organized into logically related groups of objects. I have already shown this type of model in Figure 7.4 to present the subsystems in the weather mapping system. As well as subsystem models, you may also design detailed object models, showing the objects in the systems and their associations (inheritance, generalization, aggregation, etc.). However, there is a danger

Figure 7.7 Sequence diagram describing data collection

in doing too much modeling. You should not make detailed decisions about the implementation that are really best left until the system is implemented.

Sequence models are dynamic models that describe, for each mode of interaction, the sequence of object interactions that take place. When documenting a design, you should produce a sequence model for each significant interaction. If you have developed a use case model, then there should be a sequence model for each use case that you have identified.

Figure 7.7 is an example of a sequence model, shown as a UML sequence diagram. This diagram shows the sequence of interactions that take place when an external system requests the summarized data from the weather station. You read sequence diagrams from top to bottom:

1. The **SatComms** object receives a request from the weather information system to collect a weather report from a weather station. It acknowledges receipt of this request. The stick arrowhead on the sent message indicates that the external system does not wait for a reply but can carry on with other processing.

2. **SatComms** sends a message to WeatherStation, via a satellite link, to create a summary of the collected weather data. Again, the stick arrowhead indicates that SatComms does not suspend itself waiting for a reply.

3. **WeatherStation** sends a message to a Commslink object to summarize the weather data. In this case, the squared-off style of arrowhead indicates that the instance of the WeatherStation object class waits for a reply.

4. **Commslink** calls the summarize method in the object WeatherData and waits for a reply.

Figure 7.8 Weather station state diagram

5. The weather data summary is computed and returned to **WeatherStation** via the **Commslink** object.

6. **WeatherStation** then calls the **SatComms** object to transmit the summarized data to the weather information system, through the satellite communications system.

The **SatComms** and **WeatherStation** objects may be implemented as concurrent processes, whose execution can be suspended and resumed. The **SatComms** object instance listens for messages from the external system, decodes these messages, and initiates weather station operations.

Sequence diagrams are used to model the combined behavior of a group of objects, but you may also want to summarize the behavior of an object or a subsystem in response to messages and events. To do this, you can use a state machine model that shows how the object instance changes state depending on the messages that it receives. As I discuss in Chapter 5, the UML includes state diagrams to describe state machine models.

Figure 7.8 is a state diagram for the weather station system that shows how it responds to requests for various services.

You can read this diagram as follows:

1. If the system state is **Shutdown**, then it can respond to a **restart**(), a **reconfigure**() or a **powerSave**() message. The unlabeled arrow with the black blob indicates that the **Shutdown** state is the initial state. A **restart**() message causes a transition to normal operation. Both the **powerSave**() and **reconfigure**() messages cause a transition to a state in which the system reconfigures itself. The state diagram shows that reconfiguration is allowed only if the system has been shut down.

2. In the Running state, the system expects further messages. If a shutdown() message is received, the object returns to the shutdown state.
3. If a reportWeather() message is received, the system moves to the Summarizing state. When the summary is complete, the system moves to a Transmitting state where the information is transmitted to the remote system. It then returns to the Running state.
4. If a signal from the clock is received, the system moves to the Collecting state, where it collects data from the instruments. Each instrument is instructed in turn to collect its data from the associated sensors.
5. If a remoteControl() message is received, the system moves to a controlled state in which it responds to a different set of messages from the remote control room. These are not shown on this diagram.

State diagrams are useful high-level models of a system or an object's operation. However, you don't need a state diagram for all of the objects in the system. Many system objects in a system are simple, and their operation can be easily described without a state model.

7.1.5 Interface specification

An important part of any design process is the specification of the interfaces between the components in the design. You need to specify interfaces so that objects and subsystems can be designed in parallel. Once an interface has been specified, the developers of other objects may assume that interface will be implemented.

Interface design is concerned with specifying the detail of the interface to an object or to a group of objects. This means defining the signatures and semantics of the services that are provided by the object or by a group of objects. Interfaces can be specified in the UML using the same notation as a class diagram. However, there is no attribute section, and the UML stereotype «interface» should be included in the name part. The semantics of the interface may be defined using the object constraint language (OCL). I discuss the use of the OCL in Chapter 16, where I explain how it can be used to describe the semantics of components.

You should not include details of the data representation in an interface design, as attributes are not defined in an interface specification. However, you should include operations to access and update data. As the data representation is hidden, it can be easily changed without affecting the objects that use that data. This leads to a design that is inherently more maintainable. For example, an array representation of a stack may be changed to a list representation without affecting other objects that use the stack. By contrast, you should normally expose the attributes in an object model, as this is the clearest way of describing the essential characteristics of the objects.

There is not a simple 1:1 relationship between objects and interfaces. The same object may have several interfaces, each of which is a viewpoint on the methods that it provides. This is supported directly in Java, where interfaces are declared separately from objects and objects "implement" interfaces. Equally, a group of objects may all be accessed through a single interface.

Figure 7.9 Weather station interfaces

Figure 7.9 shows two interfaces that may be defined for the weather station. The left-hand interface is a reporting interface that defines the operation names that are used to generate weather and status reports. These map directly to operations in the WeatherStation object. The remote control interface provides four operations, which map onto a single method in the WeatherStation object. In this case, the individual operations are encoded in the command string associated with the remoteControl method, shown in Figure 7.6.

7.2 Design patterns

Design patterns were derived from ideas put forward by Christopher Alexander (Alexander 1979), who suggested that there were certain common patterns of building design that were inherently pleasing and effective. The pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings. The pattern is not a detailed specification. Rather, you can think of it as a description of accumulated wisdom and experience, a well-tried solution to a common problem.

A quote from the Hillside Group website (hillside.net/patterns/), which is dedicated to maintaining information about patterns, encapsulates their role in reuse:

> *Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience*[†].

Patterns have made a huge impact on object-oriented software design. As well as being tested solutions to common problems, they have become a vocabulary for talking about a design. You can therefore explain your design by describing the patterns that you have used. This is particularly true for the best known design patterns that were originally described by the "Gang of Four" in their patterns book, published in 1995 (Gamma et al. 1995). Other important pattern descriptions are those published in a series of books by authors from Siemens, a large European technology company (Buschmann et al. 1996; Schmidt et al. 2000; Kircher and Jain 2004; Buschmann, Henney, and Schmidt 2007a, 2007b).

Patterns are a way of reusing the knowledge and experience of other designers. Design patterns are usually associated with object-oriented design. Published patterns often rely on object characteristics such as inheritance and polymorphism to provide generality. However, the general principle of encapsulating experience in a pattern is

[†]The HIllside Group: hillside.net/patterns

Pattern name: Observer

Description: Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.

Problem description: In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.

This pattern may be used in situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

Solution description: This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.

The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.

The UML model of the pattern is shown in Figure 7.12.

Consequences: The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Figure 7.10 The Observer pattern

one that is equally applicable to any kind of software design. For instance, you could have configuration patterns for instantiating reusable application systems.

The Gang of Four defined the four essential elements of design patterns in their book on patterns:

1. A name that is a meaningful reference to the pattern.
2. A description of the problem area that explains when the pattern may be applied.
3. A solution description of the parts of the design solution, their relationships and their responsibilities. This is not a concrete design description. It is a template for a design solution that can be instantiated in different ways. This is often expressed graphically and shows the relationships between the objects and object classes in the solution.
4. A statement of the consequences—the results and trade-offs—of applying the pattern. This can help designers understand whether or not a pattern can be used in a particular situation.

Gamma and his co-authors break down the problem description into motivation (a description of why the pattern is useful) and applicability (a description of situations in which the pattern may be used). Under the description of the solution, they describe the pattern structure, participants, collaborations, and implementation.

To illustrate pattern description, I use the Observer pattern, taken from the Gang of Four's patterns book. This is shown in Figure 7.10. In my description, I use the

Figure 7.11 Multiple displays

four essential description elements and also include a brief statement of what the pattern can do. This pattern can be used in situations where different presentations of an object's state are required. It separates the object that must be displayed from the different forms of presentation. This is illustrated in Figure 7.11, which shows two different graphical presentations of the same dataset.

Graphical representations are normally used to illustrate the object classes in patterns and their relationships. These supplement the pattern description and add detail to the solution description. Figure 7.12 is the representation in UML of the Observer pattern.

To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied. Examples of such problems, documented in the Gang of Four's original patterns book, include:

1. Tell several objects that the state of some other object has changed (Observer pattern).
2. Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).

Figure 7.12 A UML model of the Observer pattern

3. Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
4. Allow for the possibility of extending the functionality of an existing class at runtime (Decorator pattern).

Patterns support high-level, concept reuse. When you try to reuse executable components you are inevitably constrained by detailed design decisions that have been made by the implementers of these components. These range from the particular algorithms that have been used to implement the components to the objects and types in the component interfaces. When these design decisions conflict with your requirements, reusing the component is either impossible or introduces inefficiencies into your system. Using patterns means that you reuse the ideas but can adapt the implementation to suit the system you are developing.

When you start designing a system, it can be difficult to know, in advance, if you will need a particular pattern. Therefore, using patterns in a design process often involves developing a design, experiencing a problem, and then recognizing that a pattern can be used. This is certainly possible if you focus on the 23 general-purpose patterns documented in the original patterns book. However, if your problem is a different one, you may find it difficult to find an appropriate pattern among the hundreds of different patterns that have been proposed.

Patterns are a great idea, but you need experience of software design to use them effectively. You have to recognize situations where a pattern can be applied. Inexperienced programmers, even if they have read the pattern books, will always find it hard to decide whether they can reuse a pattern or need to develop a special-purpose solution.

7.3 Implementation issues

Software engineering includes all of the activities involved in software development from the initial requirements of the system through to maintenance and management of the deployed system. A critical stage of this process is, of course, system implementation, where you create an executable version of the software. Implementation may involve developing programs in high- or low-level programming languages or tailoring and adapting generic, off-the-shelf systems to meet the specific requirements of an organization.

I assume that most readers of this book will understand programming principles and will have some programming experience. As this chapter is intended to offer a language-independent approach, I haven't focused on issues of good programming practice as language-specific examples need to be used. Instead, I introduce some aspects of implementation that are particularly important to software engineering and that are often not covered in programming texts. These are:

1. *Reuse* Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.

Figure 7.13 Software reuse

2. *Configuration management* During the development process, many different versions of each software component are created. If you don't keep track of these versions in a configuration management system, you are liable to include the wrong versions of these components in your system.

3. *Host-target development* Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system). The host and target systems are sometimes of the same type, but often they are completely different.

7.3.1 Reuse

From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language. The only significant reuse or software was the reuse of functions and objects in programming language libraries. However, costs and schedule pressure meant that this approach became increasingly unviable, especially for commercial and Internet-based systems. Consequently, an approach to development based on the reuse of existing software is now the norm for many types of system development. A reuse-based approach is now widely used for web-based systems of all kinds, scientific software, and, increasingly, in embedded systems engineering.

Software reuse is possible at a number of different levels, as shown in Figure 7.13:

1. *The abstraction level* At this level, you don't reuse software directly but rather use knowledge of successful abstractions in the design of your software. Design patterns and architectural patterns (covered in Chapter 6) are ways of representing abstract knowledge for reuse.

2. *The object level* At this level, you directly reuse objects from a library rather than writing the code yourself. To implement this type of reuse, you have to find appropriate libraries and discover if the objects and methods offer the functionality that you need. For example, if you need to process email messages in a Java program, you may use objects and methods from a JavaMail library.

3. *The component level* Components are collections of objects and object classes that operate together to provide related functions and services. You often have to adapt and extend the component by adding some code of your own. An example of component-level reuse is where you build your user interface using a framework. This is a set of general object classes that implement event handling, display management, etc. You add connections to the data to be displayed and write code to define specific display details such as screen layout and colors.

4. *The system level* At this level, you reuse entire application systems. This function usually involves some kind of configuration of these systems. This may be done by adding and modifying code (if you are reusing a software product line) or by using the system's own configuration interface. Most commercial systems are now built in this way where generic application systems systems are adapted and reused. Sometimes this approach may involve integrating several application systems to create a new system.

By reusing existing software, you can develop new systems more quickly, with fewer development risks and at lower cost. As the reused software has been tested in other applications, it should be more reliable than new software. However, there are costs associated with reuse:

1. The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs. You may have to test the software to make sure that it will work in your environment, especially if this is different from its development environment.

2. Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.

3. The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.

4. The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed. Integrating reusable software from different providers can be difficult and expensive because the providers may make conflicting assumptions about how their respective software will be reused.

How to reuse existing knowledge and software should be the first thing you should think about when starting a software development project. You should consider the

Figure 7.14 Configuration management

possibilities of reuse before designing the software in detail, as you may wish to adapt your design to reuse existing software assets. As I discussed in Chapter 2, in a reuse-oriented development process, you search for reusable elements, then modify your requirements and design to make the best use of these.

Because of the importance of reuse in modern software engineering, I devote several chapters in Part 3 of this book to this topic (Chapters 15, 16, and 18).

7.3.2 Configuration management

In software development, change happens all the time, so change management is absolutely essential. When several people are involved in developing a software system, you have to make sure that team members don't interfere with each other's work. That is, if two people are working on a component, their changes have to be coordinated. Otherwise, one programmer may make changes and overwrite the other's work. You also have to ensure that everyone can access the most up-to-date versions of software components; otherwise developers may redo work that has already been done. When something goes wrong with a new version of a system, you have to be able to go back to a working version of the system or component.

Configuration management is the name given to the general process of managing a changing software system. The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system. As shown in Figure 7.14, there are four fundamental configuration management activities:

1. *Version management,* where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers. They stop one developer from overwriting code that has been submitted to the system by someone else.

2. *System integration,* where support is provided to help developers define what versions of components are used to create each version of a system. This

Figure 7.15 Host-target development

description is then used to build a system automatically by compiling and linking the required components.

3. *Problem tracking,* where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

4. *Release management,* where new versions of a software system are released to customers. Release management is concerned with planning the functionality of new releases and organizing the software for distribution.

Software configuration management tools support each of the above activities. These tools are usually installed in an integrated development environment, such as Eclipse. Version management may be supported using a version management system such as Subversion (Pilato, Collins-Sussman, and Fitzpatrick 2008) or Git (Loeliger and McCullough 2012), which can support multi-site, multi-team development. System integration support may be built into the language or rely on a separate toolset such as the GNU build system. Bug tracking or issue tracking systems, such as Bugzilla, are used to report bugs and other issues and to keep track of whether or not these have been fixed. A comprehensive set of tools built around the Git system is available at Github (http://github.com).

Because of its importance in professional software engineering, I discuss change and configuration management in more detail in Chapter 25.

7.3.3 Host-target development

Most professional software development is based on a host-target model (Figure 7.15). Software is developed on one computer (the host) but runs on a separate machine (the target). More generally, we can talk about a development platform (host) and an execution platform (target). A platform is more than just hardware. It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.

Sometimes, the development platform and execution platform are the same, making it possible to develop the software and test it on the same machine. Therefore, if you develop in Java, the target environment is the Java Virtual Machine. In principle, this is the same on every computer, so programs should be portable from one machine to another. However, particularly for embedded systems and mobile systems, the development and the execution platforms are different. You need to either move your developed software to the execution platform for testing or run a simulator on your development machine.

Simulators are often used when developing embedded systems. You simulate hardware devices, such as sensors, and the events in the environment in which the system will be deployed. Simulators speed up the development process for embedded systems as each developer can have his or her own execution platform with no need to download the software to the target hardware. However, simulators are expensive to develop and so are usually available only for the most popular hardware architectures.

If the target system has installed middleware or other software that you need to use, then you need to be able to test the system using that software. It may be impractical to install that software on your development machine, even if it is the same as the target platform, because of license restrictions. If this is the case, you need to transfer your developed code to the execution platform to test the system.

A software development platform should provide a range of tools to support software engineering processes. These may include:

1. An integrated compiler and syntax-directed editing system that allows you to create, edit, and compile code.
2. A language debugging system.
3. Graphical editing tools, such as tools to edit UML models.
4. Testing tools, such as JUnit, that can automatically run a set of tests on a new version of a program.
5. Tools to support refactoring and program visualization.
6. Configuration management tools to manage source code versions and to integrate and build systems.

In addition to these standard tools, your development system may include more specialized tools such as static analyzers (discussed in Chapter 12). Normally, development environments for teams also include a shared server that runs a change and configuration management system and, perhaps, a system to support requirements management.

Software development tools are now usually installed within an integrated development environment (IDE). An IDE is a set of software tools that supports different aspects of software development within some common framework and user interface. Generally, IDEs are created to support development in a specific programming

UML deployment diagrams

UML deployment diagrams show how software components are physically deployed on processors. That is, the deployment diagram shows the hardware and software in the system and the middleware used to connect the different components in the system. Essentially, you can think of deployment diagrams as a way of defining and documenting the target environment.

http://software-engineering-book.com/web/deployment/

language such as Java. The language IDE may be developed specially or may be an instantiation of a general-purpose IDE, with specific language-support tools.

A general-purpose IDE is a framework for hosting software tools that provides data management facilities for the software being developed and integration mechanisms that allow tools to work together. The best-known general-purpose IDE is the Eclipse environment (http://www.eclipse.org). This environment is based on a plug-in architecture so that it can be specialized for different languages, such as Java, and application domains. Therefore, you can install Eclipse and tailor it for your specific needs by adding plug-ins. For example, you may add a set of plug-ins to support networked systems development in Java (Vogel 2013) or embedded systems engineering using C.

As part of the development process, you need to make decisions about how the developed software will be deployed on the target platform. This is straightforward for embedded systems, where the target is usually a single computer. However, for distributed systems, you need to decide on the specific platforms where the components will be deployed. Issues that you have to consider in making this decision are:

1. *The hardware and software requirements of a component* If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.

2. *The availability requirements of the system* High-availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.

3. *Component communications* If there is a lot of intercomponent communication, it is usually best to deploy them on the same platform or on platforms that are physically close to one another. This reduces communications latency—the delay between the time that a message is sent by one component and received by another.

You can document your decisions on hardware and software deployment using UML deployment diagrams, which show how software components are distributed across hardware platforms.

If you are developing an embedded system, you may have to take into account target characteristics, such as its physical size, power capabilities, the need for real-time responses to sensor events, the physical characteristics of actuators and its real-time operating system. I discuss embedded systems engineering in Chapter 21.

7.4 Open-source development

Open-source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process (Raymond 2001). Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish. There was an assumption that the code would be controlled and developed by a small core group, rather than users of the code.

Open-source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code. In principle at least, any contributor to an open-source project may report and fix bugs and propose new features and functionality. However, in practice, successful open-source systems still rely on a core group of developers who control changes to the software.

Open-source software is the backbone of the Internet and software engineering. The Linux operating system is the most widely used server system, as is the open-source Apache web server. Other important and universally used open-source products are Java, the Eclipse IDE, and the mySQL database management system. The Android operating system is installed on millions of mobile devices. Major players in the computer industry such as IBM and Oracle, support the open-source movement and base their software on open-source products. Thousands of other, lesser-known open-source systems and components may also be used.

It is usually cheap or even free to acquire open-source software. You can normally download open-source software without charge. However, if you want documentation and support, then you may have to pay for this, but costs are usually fairly low. The other key benefit of using open-source products is that widely used open-source systems are very reliable. They have a large population of users who are willing to fix problems themselves rather than report these problems to the developer and wait for a new release of the system. Bugs are discovered and repaired more quickly than is usually possible with proprietary software.

For a company involved in software development, there are two open-source issues that have to be considered:

1. Should the product that is being developed make use of open-source components?
2. Should an open-source approach be used for its own software development?

The answers to these questions depend on the type of software that is being developed and the background and experience of the development team.

If you are developing a software product for sale, then time to market and reduced costs are critical. If you are developing software in a domain in which there are high-quality open-source systems available, you can save time and money by using these systems. However, if you are developing software to a specific set of organizational requirements, then using open-source components may not be an option. You may have to integrate your software with existing systems that are incompatible with available

open-source systems. Even then, however, it could be quicker and cheaper to modify the open-source system rather than redevelop the functionality that you need.

Many software product companies are now using an open-source approach to development, especially for specialized systems. Their business model is not reliant on selling a software product but rather on selling support for that product. They believe that involving the open-source community will allow software to be developed more cheaply and more quickly and will create a community of users for the software.

Some companies believe that adopting an open-source approach will reveal confidential business knowledge to their competitors and so are reluctant to adopt this development model. However, if you are working in a small company and you open source your software, this may reassure customers that they will be able to support the software if your company goes out of business.

Publishing the source code of a system does not mean that people from the wider community will necessarily help with its development. Most successful open-source products have been platform products rather than application systems. There are a limited number of developers who might be interested in specialized application systems. Making a software system open source does not guarantee community involvement. There are thousands of open-source projects on Sourceforge and GitHub that have only a handful of downloads. However, if users of your software have concerns about its availability in future, making the software open source means that they can take their own copy and so be reassured that they will not lose access to it.

7.4.1 Open-source licensing

Although a fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code. Legally, the developer of the code (either a company or an individual) owns the code. They can place restrictions on how it is used by including legally binding conditions in an open-source software license (St. Laurent 2004). Some open-source developers believe that if an open-source component is used to develop a new system, then that system should also be open source. Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed-source systems.

Most open-source licenses (Chapman 2010) are variants of one of three general models:

1. The GNU General Public License (GPL). This is a so-called reciprocal license that simplistically means that if you use open-source software that is licensed under the GPL license, then you must make that software open source.

2. The GNU Lesser General Public License (LGPL). This is a variant of the GPL license where you can write components that link to open-source code without having to publish the source of these components. However, if you change the licensed component, then you must publish this as open source.

3. The Berkley Standard Distribution (BSD) License. This is a nonreciprocal license, which means you are not obliged to re-publish any changes or modifications made to

open-source code. You can include the code in proprietary systems that are sold. If you use open-source components, you must acknowledge the original creator of the code. The MIT license is a variant of the BSD license with similar conditions.

Licensing issues are important because if you use open-source software as part of a software product, then you may be obliged by the terms of the license to make your own product open source. If you are trying to sell your software, you may wish to keep it secret. This means that you may wish to avoid using GPL-licensed open-source software in its development.

If you are building software that runs on an open-source platform but that does not reuse open-source components, then licenses are not a problem. However, if you embed open-source software in your software, you need processes and databases to keep track of what's been used and their license conditions. Bayersdorfer (Bayersdorfer 2007) suggests that companies managing projects that use open source should:

1. Establish a system for maintaining information about open-source components that are downloaded and used. You have to keep a copy of the license for each component that was valid at the time the component was used. Licenses may change, so you need to know the conditions that you have agreed to.
2. Be aware of the different types of licenses and understand how a component is licensed before it is used. You may decide to use a component in one system but not in another because you plan to use these systems in different ways.
3. Be aware of evolution pathways for components. You need to know a bit about the open-source project where components are developed to understand how they might change in future.
4. Educate people about open source. It's not enough to have procedures in place to ensure compliance with license conditions. You also need to educate developers about open source and open-source licensing.
5. Have auditing systems in place. Developers, under tight deadlines, might be tempted to break the terms of a license. If possible, you should have software in place to detect and stop this.
6. Participate in the open-source community. If you rely on open-source products, you should participate in the community and help support their development.

The open-source approach is one of several business models for software. In this model, companies release the source of their software and sell add-on services and advice in association with this. They may also sell cloud-based software services—an attractive option for users who do not have the expertise to manage their own open-source system and also specialized versions of their system for particular clients. Open-source is therefore likely to increase in importance as a way of developing and distributing software.

KEY POINTS

- Software design and implementation are interleaved activities. The level of detail in the design depends on the type of system being developed and whether you are using a plan-driven or agile approach.
- The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models, and document the component interfaces.
- A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
- Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.
- When developing software, you should always consider the possibility of reusing existing software, either as components, services, or complete systems.
- Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people is cooperating to develop software.
- Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- Open-source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.

FURTHER READING

Design Patterns: Elements of Reusable Object-oriented Software. This is the original software patterns handbook that introduced software patterns to a wide community. (E. Gamma, R. Helm, R. Johnson and J. Vlissides, Addison-Wesley, 1995).

Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development, 3rd ed. Larman writes clearly on object-oriented design and also discusses use of the UML; this is a good introduction to using patterns in the design process. Although it is more than 10 years old, it remains the best book on this topic that is available. (C. Larman, Prentice-Hall, 2004).

Producing Open Source Software: How to Run a Successful Free Software Project. This book is a comprehensive guide to the background to open-source software, licensing issues, and the practicalities of running an open-source development project. (K. Fogel, O'Reilly Media Inc., 2008).

Further reading on software reuse is suggested in Chapter 15 and on configuration management in Chapter 25.

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/implementation-and-evolution/

More information on the weather information system:

http://software-engineering-book.com/case-studies/wilderness-weather-station/

EXERCISES

7.1. Using the tabular notation shown in Figure 7.3, specify the weather station use cases for Report status and Reconfigure. You should make reasonable assumptions about the functionality that is required here.

7.2. Assume that the Mentcare system is being developed using an object-oriented approach. Draw a use case diagram showing at least six possible use cases for this system.

7.3. Using the UML graphical notation for object classes, design the following object classes, identifying attributes and operations. Use your own experience to decide on the attributes and operations that should be associated with these objects.

- a messaging system on a mobile (cell) phone or tablet
- a printer for a personal computer
- a personal music system
- a bank account
- a library catalogue

7.4. A shape can be classified into 2-D and 3-D. Design an inheritance hierarchy that will include different kinds of 2-D and 3-D shapes. Make sure you identify at least five other classes of shapes.

7.5. Develop the design of the weather station to show the interaction between the data collection subsystem and the instruments that collect weather data. Use sequence diagrams to show this interaction.

7.6. Identify possible objects in the following systems and develop an object-oriented design for them. You may make any reasonable assumptions about the systems when deriving the design.

- A group diary and time management system is intended to support the timetabling of meetings and appointments across a group of co-workers. When an appointment is to be made

that involves a number of people, the system finds a common slot in each of their diaries and arranges the appointment for that time. If no common slots are available, it interacts with the user to rearrange his or her personal diary to make room for the appointment.

- A filling station (gas station) is to be set up for fully automated operation. Drivers swipe their credit card through a reader connected to the pump; the card is verified by communication with a credit company computer, and a fuel limit is established. The driver may then take the fuel required. When fuel delivery is complete and the pump hose is returned to its holster, the driver's credit card account is debited with the cost of the fuel taken. The credit card is returned after debiting. If the card is invalid, the pump returns it before fuel is dispensed.

7.7. Draw a sequence diagram showing the interactions of objects in a group diary system when a group of people are arranging a meeting.

7.8. Draw a UML state diagram showing the possible state changes in either the group diary or the filling station system.

7.9. When code is integrated into a larger system, problems may surface. Explain how configuration management can be useful when handling such problems.

7.10. A small company has developed a specialized software product that it configures specially for each customer. New customers usually have specific requirements to be incorporated into their system, and they pay for these to be developed and integrated with the product. The software company has an opportunity to bid for a new contract, which would more than double its customer base. The new customer wishes to have some involvement in the configuration of the system. Explain why, in these circumstances, it might be a good idea for the company owning the software to make it open source.

REFERENCES

Abbott, R. 1983. "Program Design by Informal English Descriptions." *Comm. ACM* 26 (11): 882–894. doi:10.1145/182.358441.

Alexander, C. 1979. *A Timeless Way of Building*. Oxford, UK: Oxford University Press.

Bayersdorfer, M. 2007. "Managing a Project with Open Source Components." *ACM Interactions* 14 (6): 33–34. doi:10.1145/1300655.1300677.

Beck, K., and W. Cunningham. 1989. "A Laboratory for Teaching Object-Oriented Thinking." In *Proc. OOPSLA' 89 (Conference on Object-Oriented Programming, Systems, Languages and Applications)*, 1–6. ACM Press. doi:10.1145/74878.74879.

Buschmann, F., K. Henney, and D. C. Schmidt. 2007a. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. New York: John Wiley & Sons.

———. 2007b. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. New York: John Wiley & Sons.

Buschmann, F., R. Meunier, H. Rohnert, and P. Sommerlad. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns.* New York: John Wiley & Sons.

Chapman, C. 2010. "A Short Guide to Open-Source and Similar Licences." *Smashing Magazine*. http://www.smashingmagazine.com/2010/03/24/a-short-guide-to-open-source-and-similar-licenses./

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA.: Addison-Wesley*.

Kircher, M., and P. Jain. 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. New York: John Wiley & Sons.

Loeliger, J., and M. McCullough. 2012. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. Sebastopol, CA: O'Reilly & Associates.

Pilato, C., B. Collins-Sussman, and B. Fitzpatrick. 2008. *Version Control with Subversion*. Sebastopol, CA: O'Reilly & Associates.

Raymond, E. S. 2001. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol. CA: O'Reilly & Associates.

Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann. 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons.

St. Laurent, A. 2004. *Understanding Open Source and Free Software Licensing*. Sebastopol, CA: O'Reilly & Associates.

Vogel, L. 2013. *Eclipse IDE: A Tutoria*l. Hamburg, Germany: Vogella Gmbh.

Wirfs-Brock, R., B. Wilkerson, and L. Weiner. 1990. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice-Hall.

8

Software testing

Objectives

The objective of this chapter is to introduce software testing and software testing processes. When you have read the chapter, you will:

- understand the stages of testing from testing during development to acceptance testing by system customers;
- have been introduced to techniques that help you choose test cases that are geared to discovering program defects;
- understand test-first development, where you design tests before writing code and run these tests automatically;
- know about three distinct types of testing—component testing, system testing, and release testing;
- understand the distinctions between development testing and user testing.

Contents

8.1 Development testing
8.2 Test-driven development
8.3 Release testing
8.4 User testing

Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use. When you test software, you execute a program using artificial data. You check the results of the test run for errors, anomalies, or information about the program's non-functional attributes.

When you test software, you are trying to do two things:

1. Demonstrate to the developer and the customer that the software meets its requirements. For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features that will be included in the product release. You may also test combinations of features to check for unwanted interactions between them.

2. Find inputs or input sequences where the behavior of the software is incorrect, undesirable, or does not conform to its specification. These are caused by defects (bugs) in the software. When you test software to find defects, you are trying to root out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations, and data corruption.

The first of these is validation testing, where you expect the system to perform correctly using a set of test cases that reflect the system's expected use. The second is defect testing, where the test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used. Of course, there is no definite boundary between these two approaches to testing. During validation testing, you will find defects in the system; during defect testing, some of the tests will show that the program meets its requirements.

Figure 8.1 shows the differences between validation testing and defect testing. Think of the system being tested as a black box. The system accepts inputs from some input set I and generates outputs in an output set O. Some of the outputs will be erroneous. These are the outputs in set O_e that are generated by the system in response to inputs in the set I_e. The priority in defect testing is to find those inputs in the set I_e because these reveal problems with the system. Validation testing involves testing with correct inputs that are outside I_e. These stimulate the system to generate the expected correct outputs.

Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstance. It is always possible that a test you have overlooked could discover further problems with the system. As Edsger Dijkstra, an early contributor to the development of software engineering, eloquently stated (Dijkstra 1972):

"Testing can only show the presence of errors, not their absence[†]*"*

Testing is part of a broader process of software verification and validation (V & V). Verification and validation are not the same thing, although they are often confused. Barry Boehm, a pioneer of software engineering, succinctly expressed the difference between them (Boehm 1979):

[†]Dijkstra, E. W. 1972. "The Humble Programmer." Comm. ACM 15 (10): 859–66. doi:10.1145/355604.361591

Figure 8.1 An input–output model of program testing

- **Validation:** *Are we building the right product?*
- **Verification:** *Are we building the product right?*

Verification and validation processes are concerned with checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software. These checking processes start as soon as requirements become available and continue through all stages of the development process.

Software verification is the process of checking that the software meets its stated functional and non-functional requirements. Validation is a more general process. The aim of software validation is to ensure that the software meets the customer's expectations. It goes beyond checking conformance with the specification to demonstrating that the software does what the customer expects it to do. Validation is essential because, as I discussed in Chapter 4, statements of requirements do not always reflect the real wishes or needs of system customers and users.

The goal of verification and validation processes is to establish confidence that the software system is "fit for purpose." This means that the system must be good enough for its intended use. The level of required confidence depends on the system's purpose, the expectations of the system users, and the current marketing environment for the system:

1. *Software purpose* The more critical the software, the more important it is that it is reliable. For example, the level of confidence required for software used to control a safety-critical system is much higher than that required for a demonstrator system that prototypes new product ideas.

2. *User expectations* Because of their previous experiences with buggy, unreliable software, users sometimes have low expectations of software quality. They are not surprised when their software fails. When a new system is installed, users

Figure 8.2 Inspections and testing

may tolerate failures because the benefits of use outweigh the costs of failure recovery. However, as a software product becomes more established, users expect it to become more reliable. Consequently, more thorough testing of later versions of the system may be required.

3. *Marketing environment* When a software company brings a system to market, it must take into account competing products, the price that customers are willing to pay for a system, and the required schedule for delivering that system. In a competitive environment, the company may decide to release a program before it has been fully tested and debugged because it wants to be the first into the market. If a software product or app is very cheap, users may be willing to tolerate a lower level of reliability.

As well as software testing, the verification and validation process may involve software inspections and reviews. Inspections and reviews analyze and check the system requirements, design models, the program source code, and even proposed system tests. These are "static" V & V techniques in which you don't need to execute the software to verify it. Figure 8.2 shows that software inspections and testing support V & V at different stages in the software process. The arrows indicate the stages in the process where the techniques may be used.

Inspections mostly focus on the source code of a system, but any readable representation of the software, such as its requirements or a design model, can be inspected. When you inspect a system, you use knowledge of the system, its application domain, and the programming or modeling language to discover errors.

Software inspection has three advantages over testing:

1. During testing, errors can mask (hide) other errors. When an error leads to unexpected outputs, you can never be sure if later output anomalies are due to a new error or are side effects of the original error. Because inspection doesn't involve executing the system, you don't have to worry about interactions between errors. Consequently, a single inspection session can discover many errors in a system.

Figure 8.3 A model of the software testing process

2. Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available. This obviously adds to the system development costs.

3. As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability, and maintainability. You can look for inefficiencies, inappropriate algorithms, and poor programming style that could make the system difficult to maintain and update.

Program inspections are an old idea, and several studies and experiments have shown that inspections are more effective for defect discovery than program testing. Fagan (Fagan 1976) reported that more than 60% of the errors in a program can be detected using informal program inspections. In the Cleanroom process (Prowell et al. 1999), it is claimed that more than 90% of defects can be discovered in program inspections.

However, inspections cannot replace software testing. Inspections are not good for discovering defects that arise because of unexpected interactions between different parts of a program, timing problems, or problems with system performance. In small companies or development groups, it can be difficult and expensive to put together a separate inspection team as all potential team members may also be developers of the software.

I discuss reviews and inspections in more detail in Chapter 24 (Quality Management). Static analysis, where the source text of a program is automatically analyzed to discover anomalies, is explained in Chapter 12. In this chapter, I focus on testing and testing processes.

Figure 8.3 is an abstract model of the traditional testing process, as used in plan-driven development. Test cases are specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested. Test data are the inputs that have been devised to test a system. Test data can sometimes be generated automatically, but automatic test case generation is impossible. People who understand what the system is supposed to do must be involved to specify the expected test results. However, test execution can be automated. The test results are automatically compared with the predicted results, so there is no need for a person to look for errors and anomalies in the test run.

Test planning

Test planning is concerned with scheduling and resourcing all of the activities in the testing process. It involves defining the testing process, taking into account the people and the time available. Usually, a test plan will be created that defines what is to be tested, the predicted testing schedule, and how tests will be recorded. For critical systems, the test plan may also include details of the tests to be run on the software.

http://software-engineering-book.com/web/test-planning/

Typically, a commercial software system has to go through three stages of testing:

1. *Development testing*, where the system is tested during development to discover bugs and defects. System designers and programmers are likely to be involved in the testing process.
2. *Release testing*, where a separate testing team tests a complete version of the system before it is released to users. The aim of release testing is to check that the system meets the requirements of the system stakeholders.
3. *User testing*, where users or potential users of a system test the system in their own environment. For software products, the "user" may be an internal marketing group that decides if the software can be marketed, released and sold. Acceptance testing is one type of user testing where the customer formally tests a system to decide if it should be accepted from the system supplier or if further development is required.

In practice, the testing process usually involves a mixture of manual and automated testing. In manual testing, a tester runs the program with some test data and compares the results to their expectations. They note and report discrepancies to the program developers. In automated testing, the tests are encoded in a program that is run each time the system under development is to be tested. This is faster than manual testing, especially when it involves regression testing—re-running previous tests to check that changes to the program have not introduced new bugs.

Unfortunately, testing can never be completely automated as automated tests can only check that a program does what it is supposed to do. It is practically impossible to use automated testing to test systems that depend on how things look (e.g., a graphical user interface), or to test that a program does not have unanticipated side effects.

8.1 Development testing

Development testing includes all testing activities that are carried out by the team developing the system. The tester of the software is usually the programmer who developed that software. Some development processes use programmer/tester pairs (Cusamano and Selby 1998) where each programmer has an associated tester who

Debugging

Debugging is the process of fixing errors and problems that have been discovered by testing. Using information from the program tests, debuggers use their knowledge of the programming language and the intended outcome of the test to locate and repair the program error. When you are debugging a program, you usually use interactive tools that provide extra information about program execution.

http://software-engineering-book.com/web/debugging/

develops tests and assists with the testing process. For critical systems, a more formal process may be used, with a separate testing group within the development team. This group is responsible for developing tests and maintaining detailed records of test results.

There are three stages of development testing:

1. *Unit testing*, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
2. *Component testing*, where several individual units are integrated to create composite components. Component testing should focus on testing the component interfaces that provide access to the component functions.
3. *System testing*, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

Development testing is primarily a defect testing process, where the aim of testing is to discover bugs in the software. It is therefore usually interleaved with debugging—the process of locating problems with the code and changing the program to fix these problems.

8.1.1 Unit testing

Unit testing is the process of testing program components, such as methods or object classes. Individual functions or methods are the simplest type of component. Your tests should be calls to these routines with different input parameters. You can use the approaches to test-case design discussed in Section 8.1.2 to design the function or method tests.

When you are testing object classes, you should design your tests to provide coverage of all of the features of the object. This means that you should test all operations associated with the object; set and check the value of all attributes associated with the object; and put the object into all possible states. This means that you should simulate all events that cause a state change.

Consider, for example, the weather station object from the example that I discussed in Chapter 7. The attributes and operations of this object are shown in Figure 8.4.

Figure 8.4 The weather station object interface

It has a single attribute, which is its identifier. This is a constant that is set when the weather station is installed. You therefore only need a test that checks if it has been properly set up. You need to define test cases for all of the methods associated with the object such as **reportWeather** and **reportStatus**. Ideally, you should test methods in isolation, but, in some cases, test sequences are necessary. For example, to test the method that shuts down the weather station instruments (**shutdown**), you need to have executed the **restart** method.

Generalization or inheritance makes object class testing more complicated. You can't simply test an operation in the class where it is defined and assume that it will work as expected in all of the subclasses that inherit the operation. The operation that is inherited may make assumptions about other operations and attributes. These assumptions may not be valid in some subclasses that inherit the operation. You therefore have to test the inherited operation everywhere that it is used.

To test the states of the weather station, you can use a state model as discussed in Chapter 7 (Figure 7.8). Using this model, you identify sequences of state transitions that have to be tested and define event sequences to force these transitions. In principle, you should test every possible state transition sequence, although in practice this may be too expensive. Examples of state sequences that should be tested in the weather station include:

Shutdown → Running → Shutdown
Configuring → Running → Testing → Transmitting → Running
Running → Collecting → Running → Summarizing → Transmitting → Running

Whenever possible, you should automate unit testing. In automated unit testing, you make use of a test automation framework, such as JUnit (Tahchiev et al. 2010) to write and run your program tests. Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some graphical unit interface (GUI), on the success or otherwise of the tests. An entire test suite can often be run in a few seconds, so it is possible to execute all tests every time you make a change to the program.

An automated test has three parts:

1. *A setup part*, where you initialize the system with the test case, namely, the inputs and expected outputs.

2. *A call part,* where you call the object or method to be tested.
3. *An assertion part*, where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful; if false, then it has failed.

Sometimes, the object that you are testing has dependencies on other objects that may not have been implemented or whose use slows down the testing process. For example, if an object calls a database, this may involve a slow setup process before it can be used. In such cases, you may decide to use mock objects.

Mock objects are objects with the same interface as the external objects being used that simulate its functionality. For example, a mock object simulating a database may have only a few data items that are organized in an array. They can be accessed quickly, without the overheads of calling a database and accessing disks. Similarly, mock objects can be used to simulate abnormal operations or rare events. For example, if your system is intended to take action at certain times of day, your mock object can simply return those times, irrespective of the actual clock time.

8.1.2 Choosing unit test cases

Testing is expensive and time consuming, so it is important that you choose effective unit test cases. Effectiveness, in this case, means two things:

1. The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
2. If there are defects in the component, these should be revealed by test cases.

You should therefore design two kinds of test case. The first of these should reflect normal operation of a program and should show that the component works. For example, if you are testing a component that creates and initializes a new patient record, then your test case should show that the record exists in a database and that its fields have been set as specified. The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

Two strategies that can be effective in helping you choose test cases are:

1. *Partition testing*, where you identify groups of inputs that have common characteristics and should be processed in the same way. You should choose tests from within each of these groups.
2. *Guideline-based testing*, where you use testing guidelines to choose test cases. These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

Figure 8.5 Equivalence partitioning

The input data and output results of a program can be thought of as members of sets with common characteristics. Examples of these sets are positive numbers, negative numbers, and menu selections. Programs normally behave in a comparable way for all members of a set. That is, if you test a program that does a computation and requires two positive numbers, then you would expect the program to behave in the same way for all positive numbers.

Because of this equivalent behavior, these classes are sometimes called equivalence partitions or domains (Bezier 1990). One systematic approach to test-case design is based on identifying all input and output partitions for a system or component. Test cases are designed so that the inputs or outputs lie within these partitions. Partition testing can be used to design test cases for both systems and components.

In Figure 8.5, the large shaded ellipse on the left represents the set of all possible inputs to the program that is being tested. The smaller unshaded ellipses represent equivalence partitions. A program being tested should process all of the members of an input equivalence partition in the same way.

Output equivalence partitions are partitions within which all of the outputs have something in common. Sometimes there is a 1:1 mapping between input and output equivalence partitions. However, this is not always the case; you may need to define a separate input equivalence partition, where the only common characteristic of the inputs is that they generate outputs within the same output partition. The shaded area in the left ellipse represents inputs that are invalid. The shaded area in the right ellipse represents exceptions that may occur, that is, responses to invalid inputs.

Once you have identified a set of partitions, you choose test cases from each of these partitions. A good rule of thumb for test-case selection is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partition. The reason for this is that designers and programmers tend to consider typical values of inputs when developing a system. You test these by choosing the midpoint of the partition. Boundary values are often atypical (e.g., zero may behave differently from other non-negative numbers) and so are sometimes overlooked by developers. Program failures often occur when processing these atypical values.

Figure 8.6 Equivalence partitions

You identify partitions by using the program specification or user documentation and from experience where you predict the classes of input value that are likely to detect errors. For example, say a program specification states that the program accepts four to eight inputs which are five-digit integers greater than 10,000. You use this information to identify the input partitions and possible test input values. These are shown in Figure 8.6.

When you use the specification of a system to identify equivalence partitions, this is called black-box testing. You don't need any knowledge of how the system works. It is sometimes useful to supplement the black-box tests with "white-box testing," where you look at the code of the program to find other possible tests. For example, your code may include exceptions to handle incorrect inputs. You can use this knowledge to identify "exception partitions"—different ranges where the same exception handling should be applied.

Equivalence partitioning is an effective approach to testing because it helps account for errors that programmers often make when processing inputs at the edges of partitions. You can also use testing guidelines to help choose test cases. Guidelines encapsulate knowledge of what kinds of test cases are effective for discovering errors. For example, when you are testing programs with sequences, arrays, or lists, guidelines that could help reveal defects include:

1. Test software with sequences that have only a single value. Programmers naturally think of sequences as made up of several values, and sometimes they embed this assumption in their programs. Consequently, if presented with a single-value sequence, a program may not work properly.

2. Use different sequences of different sizes in different tests. This decreases the chances that a program with defects will accidentally produce a correct output because of some accidental characteristics of the input.

3. Derive tests so that the first, middle, and last elements of the sequence are accessed. This approach reveals problems at partition boundaries.

Path testing

Path testing is a testing strategy that aims to exercise every independent execution path through a component or program. If every independent path is executed, then all statements in the component must have been executed at least once. All conditional statements are tested for both true and false cases. In an object-oriented development process, path testing may be used to test the methods associated with objects.

http://software-engineering-book.com/web/path-testing/

Whittaker's book (Whittaker 2009) includes many examples of guidelines that can be used in test-case design. Some of the most general guidelines that he suggests are:

- Choose inputs that force the system to generate all error messages:
- Design inputs that cause input buffers to overflow.
- Repeat the same input or series of inputs numerous times.
- Force invalid outputs to be generated.
- Force computation results to be too large or too small.

As you gain experience with testing, you can develop your own guidelines about how to choose effective test cases. I give more examples of testing guidelines in the next section.

8.1.3 Component testing

Software components are often made up of several interacting objects. For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration. You access the functionality of these objects through component interfaces (see Chapter 7). Testing composite components should therefore focus on showing that the component interface or interfaces behave according to its specification. You can assume that unit tests on the individual objects within the component have been completed.

Figure 8.7 illustrates the idea of component interface testing. Assume that components A, B, and C have been integrated to create a larger component or subsystem. The test cases are not applied to the individual components but rather to the interface of the composite component created by combining these components. Interface errors in the composite component may not be detectable by testing the individual objects because these errors result from interactions between the objects in the component.

There are different types of interface between program components and, consequently, different types of interface error that can occur:

1. *Parameter interfaces* These are interfaces in which data or sometimes function references are passed from one component to another. Methods in an object have a parameter interface.

Figure 8.7 Interface testing

2. *Shared memory interfaces* These are interfaces in which a block of memory is shared between components. Data is placed in the memory by one subsystem and retrieved from there by other subsystems. This type of interface is used in embedded systems, where sensors create data that is retrieved and processed by other system components.

3. *Procedural interfaces* These are interfaces in which one component encapsulates a set of procedures that can be called by other components. Objects and reusable components have this form of interface.

4. *Message passing interfaces* These are interfaces in which one component requests a service from another component by passing a message to it. A return message includes the results of executing the service. Some object-oriented systems have this form of interface, as do client–server systems.

Interface errors are one of the most common forms of error in complex systems (Lutz 1993). These errors fall into three classes:

- *Interface misuse* A calling component calls some other component and makes an error in the use of its interface. This type of error is common in parameter interfaces, where parameters may be of the wrong type or be passed in the wrong order, or the wrong number of parameters may be passed.

- *Interface misunderstanding* A calling component misunderstands the specification of the interface of the called component and makes assumptions about its behavior. The called component does not behave as expected, which then causes unexpected behavior in the calling component. For example, a binary search method may be called with a parameter that is an unordered array. The search would then fail.

- *Timing errors* These occur in real-time systems that use a shared memory or a message-passing interface. The producer of data and the consumer of data may

operate at different speeds. Unless particular care is taken in the interface design, the consumer can access out-of-date information because the producer of the information has not updated the shared interface information.

Testing for interface defects is difficult because some interface faults may only manifest themselves under unusual conditions. For example, say an object implements a queue as a fixed-length data structure. A calling object may assume that the queue is implemented as an infinite data structure, and so it does not check for queue overflow when an item is entered.

This condition can only be detected during testing by designing a sequence of test cases that force the queue to overflow. The tests should check how calling objects handle that overflow. However, as this is a rare condition, testers may think that this isn't worth checking when writing the test set for the queue object.

A further problem may arise because of interactions between faults in different modules or objects. Faults in one object may only be detected when some other object behaves in an unexpected way. Say an object calls another object to receive some service and the calling object assumes that the response is correct. If the called service is faulty in some way, the returned value may be valid but incorrect. The problem is therefore not immediately detectable but only becomes obvious when some later computation, using the returned value, goes wrong.

Some general guidelines for interface testing are:

1. Examine the code to be tested and identify each call to an external component. Design a set of tests in which the values of the parameters to the external components are at the extreme ends of their ranges. These extreme values are most likely to reveal interface inconsistencies.

2. Where pointers are passed across an interface, always test the interface with null pointer parameters.

3. Where a component is called through a procedural interface, design tests that deliberately cause the component to fail. Differing failure assumptions are one of the most common specification misunderstandings.

4. Use stress testing in message passing systems. This means that you should design tests that generate many more messages than are likely to occur in practice. This is an effective way of revealing timing problems.

5. Where several components interact through shared memory, design tests that vary the order in which these components are activated. These tests may reveal implicit assumptions made by the programmer about the order in which the shared data is produced and consumed.

Sometimes it is better to use inspections and reviews rather than testing to look for interface errors. Inspections can concentrate on component interfaces and questions about the assumed interface behavior asked during the inspection process.

8.1.4 System testing

System testing during development involves integrating components to create a version of the system and then testing the integrated system. System testing checks that components are compatible, interact correctly, and transfer the right data at the right time across their interfaces. It obviously overlaps with component testing, but there are two important differences:

1. During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
2. Components developed by different team members or subteams may be integrated at this stage. System testing is a collective rather than an individual process. In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

All systems have emergent behavior. This means that some system functionality and characteristics only become obvious when you put the components together. This may be planned emergent behavior, which has to be tested. For example, you may integrate an authentication component with a component that updates the system database. You then have a system feature that restricts information updating to authorized users. Sometimes, however, the emergent behavior is unplanned and unwanted. You have to develop tests that check that the system is only doing what it is supposed to do.

System testing should focus on testing the interactions between the components and objects that make up a system. You may also test reusable components or systems to check that they work as expected when they are integrated with new components. This interaction testing should discover those component bugs that are only revealed when a component is used by other components in the system. Interaction testing also helps find misunderstandings, made by component developers, about other components in the system.

Because of its focus on interactions, use case-based testing is an effective approach to system testing. Several components or objects normally implement each use case in the system. Testing the use case forces these interactions to occur. If you have developed a sequence diagram to model the use case implementation, you can see the objects or components that are involved in the interaction.

In the wilderness weather station example, the system software reports summarized weather data to a remote computeras described in Figure 7.3. Figure 8.8 shows the sequence of operations in the weather station when it responds to a request to collect data for the mapping system. You can use this diagram to identify operations that will be tested and to help design the test cases to execute the tests. Therefore issuing a request for a report will result in the execution of the following thread of methods:

SatComms:request → WeatherStation:reportWeather → Commslink:Get(summary) → WeatherData:summarize

Figure 8.8 Collect weather data sequence chart

The sequence diagram helps you design the specific test cases that you need, as it shows what inputs are required and what outputs are created:

1. An input of a request for a report should have an associated acknowledgment. A report should ultimately be returned from the request. During testing, you should create summarized data that can be used to check that the report is correctly organized.

2. An input request for a report to **WeatherStation** results in a summarized report being generated. You can test this in isolation by creating raw data corresponding to the summary that you have prepared for the test of **SatComms** and checking that the **WeatherStation** object correctly produces this summary. This raw data is also used to test the **WeatherData** object.

Of course, I have simplified the sequence diagram in Figure 8.8 so that it does not show exceptions. A complete use case/scenario test must take these exceptions into account and ensure that they are correctly handled.

For most systems, it is difficult to know how much system testing is essential and when you should stop testing. Exhaustive testing, where every possible program execution sequence is tested, is impossible. Testing, therefore, has to be based on a subset of possible test cases. Ideally, software companies should have policies for choosing this subset. These policies might be based on general testing policies, such as a policy that all program statements should be executed at least once. Alternatively, they may be based on experience of system usage and focus on testing the features of the operational system. For example:

Incremental integration and testing

System testing involves integrating different components, then testing the integrated system that you have created. You should always use an incremental approach to integration and testing where you integrate a component, test the system, integrate another component, test again, and so on. If problems occur, they are probably due to interactions with the most recently integrated component.

Incremental integration and testing is fundamental to agile methods, where regression tests are run every time a new increment is integrated.

http://software-engineering-book.com/web/integration/

1. All system functions that are accessed through menus should be tested.
2. Combinations of functions (e.g., text formatting) that are accessed through the same menu must be tested.
3. Where user input is provided, all functions must be tested with both correct and incorrect input.

It is clear from experience with major software products such as word processors or spreadsheets that similar guidelines are normally used during product testing. When features of the software are used in isolation, they normally work. Problems arise, as Whittaker explains (Whittaker 2009), when combinations of less commonly used features have not been tested together. He gives the example of how, in a commonly used word processor, using footnotes with multicolumn layout causes incorrect layout of the text.

Automated system testing is usually more difficult than automated unit or component testing. Automated unit testing relies on predicting the outputs and then encoding these predictions in a program. The prediction is then compared with the result. However, the point of implementing a system may be to generate outputs that are large or cannot be easily predicted. You may be able to examine an output and check its credibility without necessarily being able to create it in advance.

8.2 Test-driven development

Test-driven development (TDD) is an approach to program development in which you interleave testing and code development (Beck 2002; Jeffries and Melnik 2007). You develop the code incrementally, along with a set of tests for that increment. You don't start working on the next increment until the code that you have developed passes all of its tests. Test-driven development was introduced as part of the XP agile development method. However, it has now gained mainstream acceptance and may be used in both agile and plan-based processes.

Figure 8.9 Test-driven development

The fundamental TDD process is shown in Figure 8.9. The steps in the process are as follows:

1. You start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
2. You write a test for this functionality and implement it as an automated test. This means that the test can be executed and will report whether or not it has passed or failed.
3. You then run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail. This is deliberate as it shows that the test adds something to the test set.
4. You then implement the functionality and re-run the test. This may involve refactoring existing code to improve it and add new code to what's already there.
5. Once all tests run successfully, you move on to implementing the next chunk of functionality.

An automated testing environment, such as the JUnit environment that supports Java program testing (Tahchiev et al. 2010) is essential for TDD. As the code is developed in very small increments, you have to be able to run every test each time that you add functionality or refactor the program. Therefore, the tests are embedded in a separate program that runs the tests and invokes the system that is being tested. Using this approach, you can run hundreds of separate tests in a few seconds.

Test-driven development helps programmers clarify their ideas of what a code segment is actually supposed to do. To write a test, you need to understand what is intended, as this understanding makes it easier to write the required code. Of course, if you have incomplete knowledge or understanding, then TDD won't help.

If you don't know enough to write the tests, you won't develop the required code. For example, if your computation involves division, you should check that you are not dividing the numbers by zero. If you forget to write a test for this, then the checking code will never be included in the program.

As well as better problem understanding, other benefits of test-driven development are:

1. *Code coverage* In principle, every code segment that you write should have at least one associated test. Therefore, you can be confident that all of the code in

the system has actually been executed. Code is tested as it is written, so defects are discovered early in the development process.

2. *Regression testing* A test suite is developed incrementally as a program is developed. You can always run regression tests to check that changes to the program have not introduced new bugs.

3. *Simplified debugging* When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified. You do not need to use debugging tools to locate the problem. Reports of the use of TDD suggest that it is hardly ever necessary to use an automated debugger in test-driven development (Martin 2007).

4. *System documentation* The tests themselves act as a form of documentation that describe what the code should be doing. Reading the tests can make it easier to understand the code.

One of the most important benefits of TDD is that it reduces the costs of regression testing. Regression testing involves running test sets that have successfully executed after changes have been made to a system. The regression test checks that these changes have not introduced new bugs into the system and that the new code interacts as expected with the existing code. Regression testing is expensive and sometimes impractical when a system is manually tested, as the costs in time and effort are very high. You have to try to choose the most relevant tests to re-run and it is easy to miss important tests.

Automated testing dramatically reduces the costs of regression testing. Existing tests may be re-run quickly and cheaply. After making a change to a system in test-first development, all existing tests must run successfully before any further functionality is added. As a programmer, you can be confident that the new functionality that you have added has not caused or revealed problems with existing code.

Test-driven development is of most value in new software development where the functionality is either implemented in new code or by using components from standard libraries. If you are reusing large code components or legacy systems, then you need to write tests for these systems as a whole. You cannot easily decompose them into separate testable elements. Incremental test-driven development is impractical. Test-driven development may also be ineffective with multithreaded systems. The different threads may be interleaved at different times in different test runs, and so may produce different results.

If you use TDD, you still need a system testing process to validate the system, that is, to check that it meets the requirements of all of the system stakeholders. System testing also tests performance, reliability, and checks that the system does not do things that it shouldn't do, such as produce unwanted outputs. Andrea (Andrea 2007) suggests how testing tools can be extended to integrate some aspects of system testing with TDD.

Test-driven development is now a widely used and mainstream approach to software testing. Most programmers who have adopted this approach are happy with it

and find it a more productive way to develop software. It is also claimed that use of TDD encourages better structuring of a program and improved code quality. However, experiments to verify this claim have been inconclusive.

8.3 Release testing

Release testing is the process of testing a particular release of a system that is intended for use outside of the development team. Normally, the system release is for customers and users. In a complex project, however, the release could be for other teams that are developing related systems. For software products, the release could be for product management who then prepare it for sale.

There are two important distinctions between release testing and system testing during the development process:

1. The system development, team should not be responsible for release testing.
2. Release testing is a process of validation checking to ensure that a system meets its requirements and is good enough for use by system customers. System testing by the development team should focus on discovering bugs in the system (defect testing).

The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use. If so, it can be released as a product or delivered to the customer. Release testing, therefore, has to show that the system delivers its specified functionality, performance, and dependability, and that it does not fail during normal use.

Release testing is usually a black-box testing process whereby tests are derived from the system specification. The system is treated as a black box whose behavior can only be determined by studying its inputs and the related outputs. Another name for this is functional testing, so-called because the tester is only concerned with functionality and not the implementation of the software.

8.3.1 Requirements-based testing

A general principle of good requirements engineering practice is that requirements should be testable. That is, the requirement should be written so that a test can be designed for that requirement. A tester can then check that the requirement has been satisfied. Requirements-based testing, therefore, is a systematic approach to test-case design where you consider each requirement and derive a set of tests for it. Requirements-based testing is validation rather than defect testing—you are trying to demonstrate that the system has properly implemented its requirements.

For example, consider the following Mentcare system requirements that are concerned with checking for drug allergies:

If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.

If a prescriber chooses to ignore an allergy warning, he or she shall provide a reason why this has been ignored.

To check if these requirements have been satisfied, you may need to develop several related tests:

1. Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.
2. Set up a patient record with a known allergy. Prescribe the medication that the patient is allergic to and check that the warning is issued by the system.
3. Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
4. Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
5. Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.

You can see from this list that testing a requirement does not mean just writing a single test. You normally have to write several tests to ensure that you have coverage of the requirement. You should also keep traceability records of your requirements-based testing, which link the tests to the specific requirements that you have tested.

8.3.2 Scenario testing

Scenario testing is an approach to release testing whereby you devise typical scenarios of use and use these scenarios to develop test cases for the system. A scenario is a story that describes one way in which the system might be used. Scenarios should be realistic, and real system users should be able to relate to them. If you have used scenarios or user stories as part of the requirements engineering process (described in Chapter 4), then you may be able to reuse them as testing scenarios.

In a short paper on scenario testing, Kaner (Kaner 2003) suggests that a scenario test should be a narrative story that is credible and fairly complex. It should motivate stakeholders; that is, they should relate to the scenario and believe that it is

George is a nurse who specializes in mental health care. One of his responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, George logs into the Mentcare system and uses it to print his schedule of home visits for that day, along with summary information about the patients to be visited. He requests that the records for these patients be downloaded to his laptop. He is prompted for his key phrase to encrypt the records on the laptop.

One of the patients whom he visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. George looks up Jim's record and is prompted for his key phrase to decrypt the record. He checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect, so he notes the problem in Jim's record and suggests that he visit the clinic to have his medication changed. Jim agrees, so George enters a prompt to call him when he gets back to the clinic to make an appointment with a physician. George ends the consultation, and the system re-encrypts Jim's record.

After finishing his consultations, George returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for George of those patients whom he has to contact for follow-up information and make clinic appointments.

Figure 8.10 A user story for the Mentcare system

important that the system passes the test. He also suggests that it should be easy to evaluate. If there are problems with the system, then the release testing team should recognize them.

As an example of a possible scenario from the Mentcare system, Figure 8.10 describes one way that the system may be used on a home visit. This scenario tests a number of features of the Mentcare system:

1. Authentication by logging on to the system.
2. Downloading and uploading of specified patient records to a laptop.
3. Home visit scheduling.
4. Encryption and decryption of patient records on a mobile device.
5. Record retrieval and modification.
6. Links with the drugs database that maintains side-effect information.
7. The system for call prompting.

If you are a release tester, you run through this scenario, playing the role of George and observing how the system behaves in response to different inputs. As George, you may make deliberate mistakes, such as inputting the wrong key phrase to decode records. This checks the response of the system to errors. You should carefully note any problems that arise, including performance problems. If a system is too slow, this will change the way that it is used. For example, if it takes too long to encrypt a record, then users who are short of time may skip this stage. If they then lose their laptop, an unauthorized person could then view the patient records.

When you use a scenario-based approach, you are normally testing several requirements within the same scenario. Therefore, as well as checking individual requirements, you are also checking that combinations of requirements do not cause problems.

8.3.3 Performance testing

Once a system has been completely integrated, it is possible to test for emergent properties, such as performance and reliability. Performance tests have to be designed to ensure that the system can process its intended load. This usually involves running a series of tests where you increase the load until the system performance becomes unacceptable.

As with other types of testing, performance testing is concerned both with demonstrating that the system meets its requirements and discovering problems and defects in the system. To test whether performance requirements are being achieved, you may have to construct an operational profile. An operational profile (see Chapter 11) is a set of tests that reflect the actual mix of work that will be handled by the system. Therefore, if 90% of the transactions in a system are of type A, 5% of type B, and the remainder of types C, D, and E, then you have to design the operational profile so that the vast majority of tests are of type A. Otherwise, you will not get an accurate test of the operational performance of the system.

This approach, of course, is not necessarily the best approach for defect testing. Experience has shown that an effective way to discover defects is to design tests around the limits of the system. In performance testing, this means stressing the system by making demands that are outside the design limits of the software. This is known as stress testing.

Say you are testing a transaction processing system that is designed to process up to 300 transactions per second. You start by testing this system with fewer than 300 transactions per second. You then gradually increase the load on the system beyond 300 transactions per second until it is well beyond the maximum design load of the system and the system fails.

Stress testing helps you do two things:

1. Test the failure behavior of the system. Circumstances may arise through an unexpected combination of events where the load placed on the system exceeds the maximum anticipated load. In these circumstances, system failure should not cause data corruption or unexpected loss of user services. Stress testing checks that overloading the system causes it to "fail-soft" rather than collapse under its load.

2. Reveal defects that only show up when the system is fully loaded. Although it can be argued that these defects are unlikely to cause system failures in normal use, there may be unusual combinations of circumstances that the stress testing replicates.

Stress testing is particularly relevant to distributed systems based on a network of processors. These systems often exhibit severe degradation when they are heavily loaded. The network becomes swamped with coordination data that the different processes must exchange. The processes become slower and slower as they wait for the required data from other processes. Stress testing helps you discover when the degradation begins so that you can add checks to the system to reject transactions beyond this point.

8.4 User testing

User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing. This may involve formally testing a system that has been commissioned from an external supplier. Alternatively, it may be an informal process where users experiment with a new software product to see if they like it and to check that it does what they need. User testing is essential, even when comprehensive system and release testing have been carried out. Influences from the user's working environment can have a major effect on the reliability, performance, usability, and robustness of a system.

It is practically impossible for a system developer to replicate the system's working environment, as tests in the developer's environment are inevitably artificial. For example, a system that is intended for use in a hospital is used in a clinical environment where other things are going on, such as patient emergencies and conversations with relatives. These all affect the use of a system, but developers cannot include them in their testing environment.

There are three different types of user testing:

1. *Alpha testing*, where a selected group of software users work closely with the development team to test early releases of the software.
2. *Beta testing*, where a release of the software is made available to a larger group of users to allow them to experiment and to raise problems that they discover with the system developers.
3. *Acceptance testing*, where customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.

In alpha testing, users and developers work together to test a system as it is being developed. This means that the users can identify problems and issues that are not readily apparent to the development testing team. Developers can only really work from the requirements, but these often do not reflect other factors that affect the practical use of the software. Users can therefore provide information about practice that helps with the design of more realistic tests.

Alpha testing is often used when developing software products or apps. Experienced users of these products may be willing to get involved in the alpha testing process because this gives them early information about new system features that they can exploit. It also reduces the risk that unanticipated changes to the software will have disruptive effects on their business. However, alpha testing may also be used when custom software is being developed. Agile development methods advocate user involvement in the development process, and that users should play a key role in designing tests for the system.

Beta testing takes place when an early, sometimes unfinished, release of a software system is made available to a larger group of customers and users for evaluation. Beta testers may be a selected group of customers who are early adopters of the system.

Figure 8.11 The acceptance testing process

Alternatively, the software may be made publicly available for use by anyone who is interested in experimenting with it.

Beta testing is mostly used for software products that are used in many different settings. This is important as, unlike custom product developers, there is no way for the product developer to limit the software's operating environment. It is impossible for product developers to know and replicate all the settings in which the software product will be used. Beta testing is therefore used to discover interaction problems between the software and features of its operational environment. Beta testing is also a form of marketing. Customers learn about their system and what it can do for them.

Acceptance testing is an inherent part of custom systems development. Customers test a system, using their own data, and decide if it should be accepted from the system developer. Acceptance implies that final payment should be made for the software.

Figure 8.11 shows that here are six stages in the acceptance testing process:

1. *Define acceptance criteria* This stage should ideally take place early in the process before the contract for the system is signed. The acceptance criteria should be part of the system contract and be approved by the customer and the developer. In practice, however, it can be difficult to define criteria so early in the process. Detailed requirements may not be available, and the requirements will almost certainly change during the development process.

2. *Plan acceptance testing* This stage involves deciding on the resources, time, and budget for acceptance testing and establishing a testing schedule. The acceptance test plan should also discuss the required coverage of the requirements and the order in which system features are tested. It should define risks to the testing process such as system crashes and inadequate performance, and discuss how these risks can be mitigated.

3. *Derive acceptance tests* Once acceptance criteria have been established, tests have to be designed to check whether or not a system is acceptable. Acceptance tests should aim to test both the functional and non-functional characteristics (e.g., performance) of the system. They should ideally provide complete coverage of the system requirements. In practice, it is difficult to establish completely objective acceptance criteria. There is often scope for argument about whether or not a test shows that a criterion has definitely been met.

4. *Run acceptance tests* The agreed acceptance tests are executed on the system. Ideally, this step should take place in the actual environment where the system will be used, but this may be disruptive and impractical. Therefore, a user testing

environment may have to be set up to run these tests. It is difficult to automate this process as part of the acceptance tests may involve testing the interactions between end-users and the system. Some training of end-users may be required.

5. *Negotiate test results* It is very unlikely that all of the defined acceptance tests will pass and that there will be no problems with the system. If this is the case, then acceptance testing is complete and the system can be handed over. More commonly, some problems will be discovered. In such cases, the developer and the customer have to negotiate to decide if the system is good enough to be used. They must also agree on how the developer will fix the identified problems.

6. *Reject/accept system* This stage involves a meeting between the developers and the customer to decide on whether or not the system should be accepted. If the system is not good enough for use, then further development is required to fix the identified problems. Once complete, the acceptance testing phase is repeated.

You might think that acceptance testing is a clear-cut contractual issue. If a system does not pass its acceptance tests, then it should not be accepted and payment should not be made. However, the reality is more complex. Customers want to use the software as soon as they can because of the benefits of its immediate deployment. They may have bought new hardware, trained staff, and changed their processes. They may be willing to accept the software, irrespective of problems, because the costs of not using the software are greater than the costs of working around the problems.

Therefore, the outcome of negotiations may be conditional acceptance of the system. The customer may accept the system so that deployment can begin. The system provider agrees to repair urgent problems and deliver a new version to the customer as quickly as possible.

In agile methods such as Extreme Programming, there may be no separate acceptance testing activity. The end-user is part of the development team (i.e., he or she is an alpha tester) and provides the system requirements in terms of user stories. He or she is also responsible for defining the tests, which decide whether or not the developed software supports the user stories. These tests are therefore equivalent to acceptance tests. The tests are automated, and development does not proceed until the story acceptance tests have successfully been executed.

When users are embedded in a software development team, they should ideally be "typical" users with general knowledge of how the system will be used. However, it can be difficult to find such users, and so the acceptance tests may actually not be a true reflection of how a system is used in practice. Furthermore, the requirement for automated testing limits the flexibility of testing interactive systems. For such systems, acceptance testing may require groups of end-users to use the system as if it was part of their everyday work. Therefore, while an "embedded user" is an attractive notion in principle, it does not necessarily lead to high-quality tests of the system.

The problem of user involvement in agile teams is one reason why many companies use a mix of agile and more traditional testing. The system may be developed using agile techniques, but, after completion of a major release, separate acceptance testing is used to decide if the system should be accepted.

KEY POINTS

- Testing can only show the presence of errors in a program. It cannot show that there are no remaining faults.
- Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers. In the user testing process, customers or system users provide test data and check that tests are successful.
- Development testing includes unit testing in which you test individual objects and methods; component testing in which you test related groups of objects; and system testing in which you test partial or complete systems.
- When testing software, you should try to "break" the software by using experience and guidelines to choose types of test cases that have been effective in discovering defects in other systems.
- Wherever possible, you should write automated tests. The tests are embedded in a program that can be run every time a change is made to a system.
- Test-first development is an approach to development whereby tests are written before the code to be tested. Small code changes are made, and the code is refactored until all tests execute successfully.
- Scenario testing is useful because it replicates the practical use of the system. It involves inventing a typical usage scenario and using this to derive test cases.
- Acceptance testing is a user testing process in which the aim is to decide if the software is good enough to be deployed and used in its planned operational environment.

FURTHER READING

"How to design practical test cases." A how-to article on test-case design by an author from a Japanese company that has a good reputation for delivering software with very few faults. (T. Yamaura, *IEEE Software*, 15(6), November 1998) http://dx.doi.org/10.1109/52.730835.

"Test-driven development." This special issue on test-driven development includes a good general overview of TDD as well as experience papers on how TDD has been used for different types of software. (*IEEE Software*, 24 (3) May/June 2007).

Exploratory Software Testing. This is a practical, rather than theoretical, book on software testing which develops the ideas in Whittaker's earlier book, *How to Break Software*. The author presents a set of experience-based guidelines on software testing. (J. A. Whittaker, 2009, Addison-Wesley).

How Google Tests Software. This is a book about testing large-scale cloud-based systems and poses a whole set of new challenges compared to custom software applications. While I don't think that the Google approach can be used directly, there are interesting lessons in this book for large-scale system testing. (J. Whittaker, J. Arbon, and J. Carollo, 2012, Addison-Wesley).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/implementation-and-evolution/

EXERCISES

8.1. Explain how the number of known defects remaining in a program at the time of delivery affects product support.

8.2. Testing is meant to show that a program does what it is intended to do. Why may testers not always know what a program is intended for?

8.3. Some people argue that developers should not be involved in testing their own code but that all testing should be the responsibility of a separate team. Give arguments for and against testing by the developers themselves.

8.4. You have been asked to test a method called catWhiteSpace in a "Paragraph" object that, within the paragraph, replaces sequences of blank characters with a single blank character. Identify testing partitions for this example and derive a set of tests for the catWhiteSpace method.

8.5. What is regression testing? Explain how the use of automated tests and a testing framework such as JUnit simplifies regression testing.

8.6. The Mentcare system is constructed by adapting an off-the-shelf information system. What do you think are the differences between testing such a system and testing software that is developed using an object-oriented language such as Java?

8.7. Write a scenario that could be used to help design tests for the wilderness weather station system.

8.8. What do you understand by the term *stress testing*? Suggest how you might stress-test the Mentcare system.

8.9. What are the benefits of involving users in release testing at an early stage in the testing process? Are there disadvantages in user involvement?

8.10. A common approach to system testing is to test the more important functionalities of a system first, followed by the less important functionalities until the testing budget is exhausted. Discuss the ethics involved in identifying what "more important" means.

REFERENCES

Andrea, J. 2007. "Envisioning the Next Generation of Functional Testing Tools." *IEEE Software* 24 (3): 58–65. doi:10.1109/MS.2007.73.

Beck, K. 2002. *Test Driven Development: By Example*. Boston: Addison-Wesley.

Bezier, B. 1990. *Software Testing Techniques, 2nd ed*. New York: Van Nostrand Reinhold.

Boehm, B. W. 1979. "Software Engineering; R & D Trends and Defense Needs." In *Research Directions in Software Technology*, edited by P. Wegner, 1–9. Cambridge, MA: MIT Press.

Cusamano, M., and R. W. Selby. 1998. *Microsoft Secrets*. New York: Simon & Schuster.

Dijkstra, E. W. 1972. "The Humble Programmer." *Comm. ACM* 15 (10): 859–866. doi:10.1145/355604.361591.

Fagan, M. E. 1976. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems J.* 15 (3): 182–211.

Jeffries, R., and G. Melnik. 2007. "TDD: The Art of Fearless Programming." *IEEE Software* 24: 24–30. doi:10.1109/MS.2007.75.

Kaner, C. 2003. "An Introduction to Scenario Testing." *Software Testing and Quality Engineering* (October 2003).

Lutz, R. R. 1993. "Analysing Software Requirements Errors in Safety-Critical Embedded Systems." In *RE' 93*, 126–133. San Diego CA: IEEE. doi:o.1109/ISRE.1993.324825.

Martin, R. C. 2007. "Professionalism and Test-Driven Development." *IEEE Software* 24 (3): 32–36. doi:10.1109/MS.2007.85.

Prowell, S. J., C. J. Trammell, R. C. Linger, and J. H. Poore. 1999. *Cleanroom Software Engineering: Technology and Process*. Reading, MA: Addison-Wesley.

Tahchiev, P., F. Leme, V. Massol, and G. Gregory. 2010. *JUnit in Action, 2nd ed*. Greenwich, CT: Manning Publications.

Whittaker, J. A. 2009. *Exploratory Software Testing*. Boston: Addison-Wesley.

9

Software evolution

Objectives

The objectives of this chapter are to explain why software evolution is such an important part of software engineering and to describe the challenges of maintaining a large base of software systems, developed over many years. When you have read this chapter, you will:

- understand that software systems have to adapt and evolve if they are to remain useful and that software change and evolution should be considered as an integral part of software engineering;
- understand what is meant by legacy systems and why these systems are important to businesses;
- understand how legacy systems can be assessed to decide whether they should be scrapped, maintained, reengineered, or replaced;
- have learned about different types of software maintenance and the factors that affect the costs of making changes to legacy software systems.

Contents

9.1 Evolution processes
9.2 Legacy systems
9.3 Software maintenance

Large software systems usually have a long lifetime. For example, military or infrastructure systems, such as air traffic control systems, may have a lifetime of 30 years or more. Business systems are often more than 10 years old. Enterprise software costs a lot of money, so a company has to use a software system for many years to get a return on its investment. Successful software products and apps may have been introduced many years ago with new versions released every few years. For example, the first version of Microsoft Word was introduced in 1983, so it has been around for more than 30 years.

During their lifetime, operational software systems have to change if they are to emain useful. Business changes and changes to user expectations generate new requirements for the software. Parts of the software may have to be modified to correct errors that are found in operation, to adapt it for changes to its hardware and software platform, and to improve its performance or other non-functional characteristics. Software products and apps have to evolve to cope with platform changes and new features introduced by their competitors. Software systems, therefore, adapt and evolve during their lifetime from initial deployment to final retirement.

Businesses have to change their software to ensure that they continue to get value from it. Their systems are critical business assets, and they have to invest in change to maintain the value of these assets. Consequently, most large companies spend more on maintaining existing systems than on new systems development. Historical data suggests that somewhere between 60% and 90% of software costs are evolution costs (Lientz and Swanson 1980; Erlikh 2000). Jones (Jones 2006) found that about 75% of development staff in the United States in 2006 were involved in software evolution and suggested that this percentage was unlikely to fall in the foreseeable future.

Software evolution is particularly expensive in enterprise systems when individual software systems are part of a broader "system of systems." In such cases, you cannot just consider the changes to one system; you also need to examine how these changes affect the broader system of systems. Changing one system may mean that other systems in its environment may also have to evolve to cope with that change.

Therefore, as well as understanding and analyzing the impact of a proposed change on the system itself, you also have to assess how this change may affect other systems in the operational environment. Hopkins and Jenkins (Hopkins and Jenkins 2008) have coined the term *brownfield software development* to describe situations in which software systems have to be developed and managed in an environment where they are dependent on other software systems.

The requirements of installed software systems change as the business and its environment change, so new releases of the systems that incorporate changes and updates are usually created at regular intervals. Software engineering is therefore a spiral process with requirements, design, implementation, and testing going on throughout the lifetime of the system (Figure 9.1). You start by creating release 1 of the system. Once delivered, changes are proposed, and the development of release 2 starts almost immediately. In fact, the need for evolution may become obvious even before the system is deployed, so later releases of the software may start development before the current version has even been released.

In the last 10 years, the time between iterations of the spiral has reduced dramatically. Before the widespread use of the Internet, new versions of a software system

Figure 9.1 A spiral model of development and evolution

may only have been released every 2 or 3 years. Now, because of competitive pressures and the need to respond quickly to user feedback, the gap between releases of some apps and web-based systems may be weeks rather than years.

This model of software evolution is applicable when the same company is responsible for the software throughout its lifetime. There is a seamless transition from development to evolution, and the same software development methods and processes are applied throughout the lifetime of the software. Software products and apps are developed using this approach.

The evolution of custom software, however, usually follows a different model. The system customer may pay a software company to develop the software and then take over responsibility for support and evolution using its own staff. Alternatively, the software customer might issue a separate contract to a different software company for system support and evolution.

In this situation, there are likely to be discontinuities in the evolution process. Requirements and design documents may not be passed from one company to another. Companies may merge or reorganize, inherit software from other companies, and then find that this has to be changed. When the transition from development to evolution is not seamless, the process of changing the software after delivery is called software maintenance. As I discuss later in this chapter, maintenance involves extra process activities, such as program understanding, in addition to the normal activities of software development.

Rajlich and Bennett (Rajlich and Bennett 2000) propose an alternative view of the software evolution life cycle for business systems. In this model, they distinguish between evolution and servicing. Evolution is the phase in which significant changes to the software architecture and functionality are made. During servicing, the only changes that are made are relatively small but essential changes. These phases overlap with each other, as shown in Figure 9.2.

According to Rajlich and Bennett, when software is first used successfully, many changes to the requirements by stakeholders are proposed and implemented. This is

Figure 9.2 Evolution and servicing

the evolution phase. However, as the software is modified, its structure tends to degrade, and system changes become more and more expensive. This often happens after a few years of use when other environmental changes, such as hardware and operating systems, are also required. At some stage in the life cycle, the software reaches a transition point where significant changes and the implementation of new requirements become less and less cost-effective. At this stage, the software moves from evolution to servicing.

During the servicing phase, the software is still useful, but only small tactical changes are made to it. During this stage, the company is usually considering how the software can be replaced. In the final stage, the software may still be used, but only essential changes are made. Users have to work around problems that they discover. Eventually, the software is retired and taken out of use. This often incurs further costs as data is transferred from an old system to a newer replacement system.

9.1 Evolution processes

As with all software processes, there is no such thing as a standard software change or evolution process. The most appropriate evolution process for a software system depends on the type of software being maintained, the software development processes used in an organization, and the skills of the people involved. For some types of system, such as mobile apps, evolution may be an informal process, where change requests mostly come from conversations between system users and developers. For other types of systems, such as embedded critical systems, software evolution may be formalized, with structured documentation produced at each stage in the process.

Formal or informal system change proposals are the driver for system evolution in all organizations. In a change proposal, an individual or group suggests changes and updates to an existing software system. These proposals may be based on existing requirements that have not been implemented in the released system, requests for new requirements, bug reports from system stakeholders, and new ideas for software improvement from the system development team. The processes of change identification and system evolution are cyclical and continue throughout the lifetime of a system (Figure 9.3).

Before a change proposal is accepted, there needs to be an analysis of the software to work out which components need to be changed. This analysis allows the cost and the impact of the change to be assessed. This is part of the general process of change management, which should also ensure that the correct versions of

Figure 9.3 Change identification and evolution processes

components are included in each system release. I discuss change and configuration management in Chapter 25.

Figure 9.4 shows some of the activities involved in software evolution. The process includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers. The cost and impact of these changes are assessed to see how much of the system is affected by the change and how much it might cost to implement the change.

If the proposed changes are accepted, a new release of the system is planned. During release planning, all proposed changes (fault repair, adaptation, and new functionality) are considered. A decision is then made on which changes to implement in the next version of the system. The changes are implemented and validated, and a new version of the system is released. The process then iterates with a new set of changes proposed for the next release.

In situations where development and evolution are integrated, change implementation is simply an iteration of the development process. Revisions to the system are designed, implemented, and tested. The only difference between initial development and evolution is that customer feedback after delivery has to be considered when planning new releases of an application.

Where different teams are involved, a critical difference between development and evolution is that the first stage of change implementation requires program understanding.

Figure 9.4 A general model of the software evolution process

Figure 9.5 Change implementation

During the program understanding phase, new developers have to understand how the program is structured, how it delivers functionality, and how the proposed change might affect the program. They need this understanding to make sure that the implemented change does not cause new problems when it is introduced into the existing system.

If requirements specification and design documents are available, these should be updated during the evolution process to reflect the changes that are required (Figure 9.5). New software requirements should be written, and these should be analyzed and validated. If the design has been documented using UML models, these models should be updated. The proposed changes may be prototyped as part of the change analysis process, where you assess the implications and costs of making the change.

However, change requests sometimes relate to problems in operational systems that have to be tackled urgently. These urgent changes can arise for three reasons:

1. If a serious system fault is detected that has to be repaired to allow normal operation to continue or to address a serious security vulnerability.
2. If changes to the systems operating environment have unexpected effects that disrupt normal operation.
3. If there are unanticipated changes to the business running the system, such as the emergence of new competitors or the introduction of new legislation that affects the system.

In these cases, the need to make the change quickly means that you may not be able to update all of the software documentation. Rather than modify the requirements and design, you make an emergency fix to the program to solve the immediate problem (Figure 9.6). The danger here is that the requirements, the software design, and the code can become inconsistent. While you may intend to document the change in the requirements and design, additional emergency fixes to the software may then be needed. These take priority over documentation. Eventually, the original change is forgotten, and the system documentation and code are never realigned. This problem of maintaining multiple representations of a system is one of the arguments for minimal documentation, which is fundamental to agile development processes.

Emergency system repairs have to be completed as quickly as possible. You choose a quick and workable solution rather than the best solution as far as system structure is concerned. This tends to accelerate the process of software ageing so that future changes become progressively more difficult and maintenance costs increase. Ideally, after emergency code repairs are made, the new code should be refactored

Figure 9.6 The emergency repair process

and improved to avoid program degradation. Of course, the code of the repair may be reused if possible. However, an alternative, better solution to the problem may be discovered when more time is available for analysis.

Agile methods and processes, discussed in Chapter 3, may be used for program evolution as well as program development. Because these methods are based on incremental development, making the transition from agile development to postdelivery evolution should be seamless.

However, problems may arise during the handover from a development team to a separate team responsible for system evolution. There are two potentially problematic situations:

1. Where the development team has used an agile approach but the evolution team prefers a plan-based approach. The evolution team may expect detailed documentation to support evolution, and this is rarely produced in agile processes. There may be no definitive statement of the system requirements that can be modified as changes are made to the system.

2. Where a plan-based approach has been used for development but the evolution team prefers to use agile methods. In this case, the evolution team may have to start from scratch developing automated tests. The code in the system may not have been refactored and simplified, as is expected in agile development. In this case, some program reengineering may be required to improve the code before it can be used in an agile development process.

Agile techniques such as test-driven development and automated regression testing are useful when system changes are made. System changes may be expressed as user stories, and customer involvement can help prioritize changes that are required in an operational system. The Scrum approach of focusing on a backlog of work to be done can help prioritize the most important system changes. In short, evolution simply involves continuing the agile development process.

Agile methods used in development may, however, have to be modified when they are used for program maintenance and evolution. It may be practically impossible to involve users in the development team as change proposals come from a wide range of stakeholders. Short development cycles may have to be interrupted to deal with emergency repairs, and the gap between releases may have to be lengthened to avoid disrupting operational processes.

9.2 Legacy systems

Large companies started computerizing their operations in the 1960s, so for the past 50 years or so, more and more software systems have been introduced. Many of these systems have been replaced (sometimes several times) as the business has changed and evolved. However, a lot of old systems are still in use and play a critical part in the running of the business. These older software systems are sometimes called legacy systems.

Legacy systems are older systems that rely on languages and technology that are no longer used for new systems development. Typically, they have been maintained over a long period, and their structure may have been degraded by the changes that have been made. Legacy software may be dependent on older hardware, such as mainframe computers and may have associated legacy processes and procedures. It may be impossible to change to more effective business processes because the legacy software cannot be modified to support new processes.

Legacy systems are not just software systems but are broader sociotechnical systems that include hardware, software, libraries, and other supporting software and business processes. Figure 9.7 shows the logical parts of a legacy system and their relationships.

1. *System hardware* Legacy systems may have been written for hardware that is no longer available, that is expensive to maintain, and that may not be compatible with current organizational IT purchasing policies.

2. *Support software* The legacy system may rely on a range of support software from the operating system and utilities provided by the hardware manufacturer through to the compilers used for system development. Again, these may be obsolete and no longer supported by their original providers.

3. *Application software* The application system that provides the business services is usually made up of a number of application programs that have been developed at different times. Some of these programs will also be part of other application software systems.

4. *Application data* These data are processed by the application system. In many legacy systems, an immense volume of data has accumulated over the lifetime of the system. This data may be inconsistent, may be duplicated in several files, and may be spread over a number of different databases.

5. *Business processes* These processes are used in the business to achieve some business objective. An example of a business process in an insurance company would be issuing an insurance policy; in a manufacturing company, a business process would be accepting an order for products and setting up the associated manufacturing process. Business processes may be designed around a legacy system and constrained by the functionality that it provides.

6. *Business policies and rules* These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

An alternative way of looking at these components of a legacy system is as a series of layers, as shown in Figure 9.8.

Each layer depends on the layer immediately below it and interfaces with that layer. If interfaces are maintained, then you should be able to make changes within a layer without affecting either of the adjacent layers. In practice, however, this simple encapsulation is an oversimplification, and changes to one layer of the system may

Figure 9.7 The elements of a legacy system

require consequent changes to layers that are both above and below the changed level. The reasons for this are as follows:

1. Changing one layer in the system may introduce new facilities, and higher layers in the system may then be changed to take advantage of these facilities. For example, a new database introduced at the support software layer may include facilities to access the data through a web browser, and business processes may be modified to take advantage of this facility.
2. Changing the software may slow the system down so that new hardware is needed to improve the system performance. The increase in performance from the new hardware may then mean that further software changes that were previously impractical become possible.
3. It is often impossible to maintain hardware interfaces, especially if new hardware is introduced. This is a particular problem in embedded systems where there is a tight coupling between software and hardware. Major changes to the application software may be required to make effective use of the new hardware.

It is difficult to know exactly how much legacy code is still in use, but, as an indicator, industry has estimated that there are more than 200 billion lines of COBOL code in current business systems. COBOL is a programming language designed for writing business systems, and it was the main business development language from the 1960s to the 1990s, particularly in the finance industry (Mitchell 2012). These programs still work effectively and efficiently, and the companies using them see no need to change them. A major problem that they face, however, is a shortage of COBOL programmers as the original developers of the system retire. Universities no longer teach COBOL, and younger software engineers are more interested in programming in modern languages.

Skill shortages are only one of the problems of maintaining business legacy systems. Other issues include security vulnerabilities because these systems were developed before the widespread use of the Internet and problems in interfacing with systems written in modern programming languages. The original software tool supplier may be out of business or may no longer maintain the support tools used to

Figure 9.8 Legacy system layers

develop the system. The system hardware may be obsolete and so increasingly expensive to maintain.

Why then do businesses not simply replace these systems with more modern equivalents? The simple answer to this question is that it is too expensive and too risky to do so. If a legacy system works effectively, the costs of replacement may exceed the savings that come from the reduced support costs of a new system. Scrapping legacy systems and replacing them with more modern software open up the possibility of things going wrong and the new system failing to meet the needs of the business. Managers try to minimize those risks and therefore do not want to face the uncertainties of new software systems.

I discovered some of the problems of legacy system replacement when I was involved in analyzing a legacy system replacement project in a large organization. This enterprise used more than 150 legacy systems to run its business. It decided to replace all of these systems with a single, centrally maintained ERP system. For a number of business and technology reasons, the new system development was a failure, and it did not deliver the improvements promised. After spending more than £10 million, only a part of the new system was operational, and it worked less effectively than the systems it replaced. Users continued to use the older systems but could not integrate these with the part of the new system that had been implemented, so additional manual processing was required.

There are several reasons why it is expensive and risky to replace legacy systems with new systems:

1. There is rarely a complete specification of the legacy system. The original specification may have been lost. If a specification exists, it is unlikely that it has been updated with all of the system changes that have been made. Therefore, there is no straightforward way of specifying a new system that is functionally identical to the system that is in use.

2. Business processes and the ways in which legacy systems operate are often inextricably intertwined. These processes are likely to have evolved to take advantage of the software's services and to work around the software's shortcomings. If the system is replaced, these processes have to change with potentially unpredictable costs and consequences.

3. Important business rules may be embedded in the software and may not be documented elsewhere. A business rule is a constraint that applies to some business function, and breaking that constraint can have unpredictable consequences for the business. For example, an insurance company may have embedded its rules for assessing the risk of a policy application in its software. If these rules are not maintained, the company may accept high-risk policies that could result in expensive future claims.

4. New software development is inherently risky, so that there may be unexpected problems with a new system. It may not be delivered on time and for the price expected.

Keeping legacy systems in use avoids the risks of replacement, but making changes to existing software inevitably becomes more expensive as systems get older. Legacy software systems that are more than a few years old are particularly expensive to change:

1. The program style and usage conventions are inconsistent because different people have been responsible for system changes. This problem adds to the difficulty of understanding the system code.

2. Part or all of the system may be implemented using obsolete programming languages. It may be difficult to find people who have knowledge of these languages. Expensive outsourcing of system maintenance may therefore be required.

3. System documentation is often inadequate and out of date. In some cases, the only documentation is the system source code.

4. Many years of maintenance usually degrades the system structure, making it increasingly difficult to understand. New programs may have been added and interfaced with other parts of the system in an ad hoc way.

5. The system may have been optimized for space utilization or execution speed so that it runs effectively on older slower hardware. This normally involves using specific machine and language optimizations, and these usually lead to software that is hard to understand. This causes problems for programmers who have learned modern software engineering techniques and who don't understand the programming tricks that have been used to optimize the software.

6. The data processed by the system may be maintained in different files that have incompatible structures. There may be data duplication, and the data itself may be out of date, inaccurate, and incomplete. Several databases from different suppliers may be used.

At same stage, the costs of managing and maintaining the legacy system become so high that it has to be replaced with a new system. In the next section, I discuss a systematic decision-making approach to making such a replacement decision.

9.2.1 Legacy system management

For new software systems developed using modern software engineering processes, such as agile development and software product lines, it is possible to plan how to integrate system development and evolution. More and more companies understand that the system development process is a whole life-cycle process. Separating software development and software evolution is unhelpful and leads to higher costs. However, as I have discussed, there is still a huge number of legacy systems that are critical business systems. These have to be extended and adapted to changing e-business practices.

Most organizations have a limited budget for maintaining and upgrading their portfolio of legacy systems. They have to decide how to get the best return on their investment. This involves making a realistic assessment of their legacy systems and then deciding on the most appropriate strategy for evolving these systems. There are four strategic options:

1. *Scrap the system completely* This option should be chosen when the system is not making an effective contribution to business processes. This usually occurs when business processes have changed since the system was installed and are no longer reliant on the legacy system.
2. *Leave the system unchanged and continue with regular maintenance* This option should be chosen when the system is still required but is fairly stable and the system users make relatively few change requests.
3. *Reengineer the system to improve its maintainability* This option should be chosen when the system quality has been degraded by change and where new change to the system is still being proposed. This process may include developing new interface components so that the original system can work with other, newer systems.
4. *Replace all or part of the system with a new system* This option should be chosen when factors, such as new hardware, mean that the old system cannot continue in operation, or where off-the-shelf systems would allow the new system to be developed at a reasonable cost. In many cases, an evolutionary replacement strategy can be adopted where major system components are replaced by off-the-shelf systems with other components reused wherever possible.

When you are assessing a legacy system, you have to look at it from both a business perspective and a technical perspective (Warren 1998). From a business perspective, you have to decide whether or not the business really needs the system. From a technical perspective, you have to assess the quality of the application software and the system's support software and hardware. You then use a combination of the business value and the system quality to inform your decision on what to do with the legacy system.

For example, assume that an organization has 10 legacy systems. You should assess the quality and the business value of each of these systems. You may then create a chart showing relative business value and system quality. An example of

Figure 9.9 An example of a legacy system assessment

this is shown in Figure 9.9. From this diagram, you can see that there are four clusters of systems:

1. *Low quality, low business value* Keeping these systems in operation will be expensive, and the rate of the return to the business will be fairly small. These systems should be scrapped.
2. *Low quality, high business value* These systems are making an important business contribution, so they cannot be scrapped. However, their low quality means that they are expensive to maintain. These systems should be reengineered to improve their quality. They may be replaced, if suitable off-the-shelf systems are available.
3. *High quality, low business value* These systems don't contribute much to the business but may not be very expensive to maintain. It is not worth replacing these systems, so normal system maintenance may be continued if expensive changes are not required and the system hardware remains in use. If expensive changes become necessary, the software should be scrapped.
4. *High quality, high business value* These systems have to be kept in operation. However, their high quality means that you don't have to invest in transformation or system replacement. Normal system maintenance should be continued.

The business value of a system is a measure of how much time and effort the system saves compared to manual processes or the use of other systems. To assess the business value of a system, you have to identify system stakeholders, such as the end-users of a system and their managers, and ask a series of questions about the system. There are four basic issues that you have to discuss:

1. *The use of the system* If a system is only used occasionally or by a small number of people, this may mean that it has a low business value. A legacy system may have been developed to meet a business need that has either changed or can now be met

more effectively in other ways. You have to be careful, however, about occasional but important use of systems. For example, a university system for student registration may only be used at the beginning of each academic year. Although it is used infrequently, it is an essential system with a high business value.

2. *The business processes that are supported* When a system is introduced, business processes are usually introduced to exploit the system's capabilities. If the system is inflexible, changing these business processes may be impossible. However, as the environment changes, the original business processes may become obsolete. Therefore, a system may have a low business value because it forces the use of inefficient business processes.

3. *System dependability* System dependability is not only a technical problem but also a business problem. If a system is not dependable and the problems directly affect business customers, or mean that people in the business are diverted from other tasks to solve these problems, the system has a low business value.

4. *The system outputs* The key issue here is the importance of the system outputs to the successful functioning of the business. If the business depends on these outputs, then the system has a high business value. Conversely, if these outputs can be cheaply generated in some other way, or if the system produces outputs that are rarely used, then the system has a low business value.

For example, assume that a company provides a travel ordering system that is used by staff responsible for arranging travel. They can place orders with an approved travel agent. Tickets are then delivered, and the company is invoiced for them. However, a business value assessment may reveal that this system is only used for a fairly small percentage of travel orders placed. People making travel arrangements find it cheaper and more convenient to deal directly with travel suppliers through their websites. This system may still be used, but there is no real point in keeping it—the same functionality is available from external systems.

Conversely, say a company has developed a system that keeps track of all previous customer orders and automatically generates reminders for customers to reorder goods. This results in a large number of repeat orders and keeps customers satisfied because they feel that their supplier is aware of their needs. The outputs from such a system are important to the business, so this system has a high business value.

To assess a software system from a technical perspective, you need to consider both the application system itself and the environment in which the system operates. The environment includes the hardware and all associated support software such as compilers, debuggers and development environments that are needed to maintain the system. The environment is important because many system changes, such as upgrades to the hardware or operating system, result from changes to the environment.

Factors that you should consider during the environment assessment are shown in Figure 9.10. Notice that these are not all technical characteristics of the environment. You also have to consider the reliability of the suppliers of the hardware and support software. If suppliers are no longer in business, their systems may not be supported, so you may have to replace these systems.

Factor	Questions
Supplier stability	Is the supplier still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, does someone else maintain the systems?
Failure rate	Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
Age	How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. It may still function correctly, but there could be significant economic and business benefits to moving to a more modern system.
Performance	Is the performance of the system adequate? Do performance problems have a significant effect on system users?
Support requirements	What local support is required by the hardware and software? If high costs are associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licences? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
Interoperability	Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system?

Figure 9.10 Factors used in environment assessment

In the process of environmental assessment, if possible, you should ideally collect data about the system and system changes. Examples of data that may be useful include the costs of maintaining the system hardware and support software, the number of hardware faults that occur over some time period and the frequency of patches and fixes applied to the system support software.

To assess the technical quality of an application system, you have to assess those factors (Figure 9.11) that are primarily related to the system dependability, the difficulties of maintaining the system, and the system documentation. You may also collect data that will help you judge the quality of the system such as:

1. *The number of system change requests* System changes usually corrupt the system structure and make further changes more difficult. The higher this accumulated value, the lower the quality of the system.

2. *The number of user interfaces* This is an important factor in forms-based systems where each form can be considered as a separate user interface. The more interfaces, the more likely it is that there will be inconsistencies and redundancies in these interfaces.

3. *The volume of data used by the system* As the volume of data (number of files, size of database, etc.) processed by the system increases, so too do the inconsistencies and errors in that data. When data has been collected over a long period of time, errors and inconsistencies are inevitable. Cleaning up old data is a very expensive and time-consuming process.

Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent, and current?
Data	Is there an explicit data model for the system? To what extent is data duplicated across files? Is the data used by the system up to date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there people available who have experience with the system?

Figure 9.11 Factors used in application assessment

Ideally, objective assessment should be used to inform decisions about what to do with a legacy system. However, in many cases, decisions are not really objective but are based on organizational or political considerations. For example, if two businesses merge, the most politically powerful partner will usually keep its systems and scrap the other company's systems. If senior management in an organization decides to move to a new hardware platform, then this may require applications to be replaced. If no budget is available for system transformation in a particular year, then system maintenance may be continued, even though this will result in higher long-term costs.

9.3 Software maintenance

Software maintenance is the general process of changing a system after it has been delivered. The term is usually applied to custom software, where separate development groups are involved before and after delivery. The changes made to the software may be simple changes to correct coding errors, more extensive changes to correct design errors, or significant enhancements to correct specification errors or to accommodate new requirements. Changes are implemented by modifying existing system components and, where necessary, by adding new components to the system.

Program evolution dynamics

Program evolution dynamics is the study of evolving software systems, pioneered by Manny Lehman and Les Belady in the 1970s. This led to so-called Lehman's Laws, which are said to apply to all large-scale software systems. The most important of these laws are:

1. A program must continually change if it is to remain useful.
2. As an evolving program changes, its structure is degraded.
3. Over a program's lifetime, the rate of change is roughly constant and independent of the resources available.
4. The incremental change in each release of a system is roughly constant.
5. New functionality must be added to systems to increase user satisfaction.

http://software-engineering-book.com/web/program-evolution-dynamics/

There are three different types of software maintenance:

1. *Fault repairs to fix bugs and vulnerabilities.* Coding errors are usually relatively cheap to correct; design errors are more expensive because they may involve rewriting several program components. Requirements errors are the most expensive to repair because extensive system redesign may be necessary.
2. *Environmental adaptation to adapt the software to new platforms and environments.* This type of maintenance is required when some aspect of a system's environment, such as the hardware, the platform operating system, or other support software, changes. Application systems may have to be modified to cope with these environmental changes.
3. *Functionality addition to add new features and to support new requirements.* This type of maintenance is necessary when system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance.

In practice, there is no clear-cut distinction between these types of maintenance. When you adapt a system to a new environment, you may add functionality to take advantage of new environmental features. Software faults are often exposed because users use the system in unanticipated ways. Changing the system to accommodate their way of working is the best way to fix these faults.

These types of maintenance are generally recognized, but different people sometimes give them different names. "Corrective maintenance" is universally used to refer to maintenance for fault repair. However, "adaptive maintenance" sometimes means adapting to a new environment and sometimes means adapting the software to new requirements. "Perfective maintenance" sometimes means perfecting the software by implementing new requirements; in other cases, it means maintaining the functionality of the system but improving its structure and its performance. Because of this naming uncertainty, I have avoided the use of these terms in this book.

Figure 9.12
Maintenance effort distribution

Figure 9.12 shows an approximate distribution of maintenance costs, based on data from the most recent survey available (Davidsen and Krogstie 2010). This study compared maintenance cost distribution with a number of earlier studies from 1980 to 2005. The authors found that the distribution of maintenance costs had changed very little over 30 years. Although we don't have more recent data, this suggests that this distribution is still largely correct. Repairing system faults is not the most expensive maintenance activity. Evolving the system to cope with new environments and new or changed requirements generally consumes most maintenance effort.

Experience has shown that it is usually more expensive to add new features to a system during maintenance than it is to implement the same features during initial development. The reasons for this are:

1. *A new team has to understand the program being maintained.* After a system has been delivered, it is normal for the development team to be broken up and for people to work on new projects. The new team or the individuals responsible for system maintenance do not understand the system or the background to system design decisions. They need to spend time understanding the existing system before they can implement changes to it.

2. *Separating maintenance and development means there is no incentive for the development team to write maintainable software.* The contract to maintain a system is usually separate from the system development contract. A different company, rather than the original software developer, may be responsible for software maintenance. In those circumstances, a development team gets no benefit from investing effort to make the software maintainable. If a development team can cut corners to save effort during development it is worthwhile for them to do so, even if this means that the software is more difficult to change in future.

3. *Program maintenance work is unpopular.* Maintenance has a poor image among software engineers. It is seen as a less skilled process than system development

Documentation

System documentation can help the maintenance process by providing maintainers with information about the structure and organization of the system and the features that it offers to system users. While proponents of agile approaches suggest that the code should be the principal documentation, higher level design models and information about dependencies and constraints can make it easier to understand and make changes to that code.

http://software-engineering-book.com/web/documentation/ (web chapter)

and is often allocated to the least experienced staff. Furthermore, old systems may be written in obsolete programming languages. The developers working on maintenance may not have much experience of these languages and must learn these languages to maintain the system.

4. *As programs age, their structure degrades and they become harder to change.* As changes are made to programs, their structure tends to degrade. Consequently, they become harder to understand and change. Some systems have been developed without modern software engineering techniques. They may never have been well structured and were perhaps optimized for efficiency rather than understandability. System documentation may be lost or inconsistent. Old systems may not have been subject to stringent configuration management, so developers have to spend time finding the right versions of system components to change.

The first three of these problems stem from the fact that many organizations still consider software development and maintenance to be separate activities. Maintenance is seen as a second-class activity, and there is no incentive to spend money during development to reduce the costs of system change. The only long-term solution to this problem is to think of systems as evolving throughout their lifetime through a continual development process. Maintenance should have as high a status as new software development.

The fourth issue, the problem of degraded system structure, is, in some ways, the easiest problem to address. Software reengineering techniques (described later in this chapter) may be applied to improve the system structure and understandability. Architectural transformations can adapt the system to new hardware. Refactoring can improve the quality of the system code and make it easier to change.

In principle, it is almost always cost-effective to invest effort in designing and implementing a system to reduce the costs of future changes. Adding new functionality after delivery is expensive because you have to spend time learning the system and analyzing the impact of the proposed changes. Work done during development to structure the software and to make it easier to understand and change will reduce evolution costs. Good software engineering techniques such as precise specification, test-first development, the use of object-oriented development, and configuration management all help reduce maintenance cost.

These principled arguments for lifetime cost savings by investing in making systems more maintainable are, unfortunately, impossible to substantiate with real

data. Collecting data is expensive, and the value of that data is difficult to judge; therefore, the vast majority of companies do not think it is worthwhile to gather and analyze software engineering data.

In reality, most businesses are reluctant to spend more on software development to reduce longer-term maintenance costs. There are two main reasons for their reluctance:

1. Companies set out quarterly or annual spending plans, and managers are incentivized to reduce short-term costs. Investing in maintainability leads to short-term cost increases, which are measurable. However, the long-term gains can't be measured at the same time, so companies are reluctant to spend money on something with an unknown future return.
2. Developers are not usually responsible for maintaining the system they have developed. Consequently, they don't see the point of doing additional work that might reduce maintenance costs, as they will not get any benefit from it.

The only way around this problem is to integrate development and maintenance so that the original development team remains responsible for software throughout its lifetime. This is possible for software products and for companies such as Amazon, which develop and maintain their own software (O'Hanlon 2006). However, for custom software developed by a software company for a client, this is unlikely to happen.

9.3.1 Maintenance prediction

Maintenance prediction is concerned with trying to assess the changes that may be required in a software system and with identifying those parts of the system that are likely to be the most expensive to change. If you understand this, you can design the software components that are most likely to change to make them more adaptable. You can also invest effort in improving those components to reduce their lifetime maintenance costs. By predicting changes, you can also assess the overall maintenance costs for a system in a given time period and so set a budget for maintaining the software. Figure 9.13 shows possible predictions and the questions that these predictions may answer.

Predicting the number of change requests for a system requires an understanding of the relationship between the system and its external environment. Some systems have a very complex relationship with their external environment, and changes to that environment inevitably result in changes to the system. To evaluate the relationships between a system and its environment, you should look at:

1. *The number and complexity of system interfaces* The larger the number of interfaces and the more complex these interfaces, the more likely it is that interface changes will be required as new requirements are proposed.

Figure 9.13
Maintenance prediction

2. *The number of inherently volatile system requirements* As I discussed in Chapter 4, requirements that reflect organizational policies and procedures are likely to be more volatile than requirements that are based on stable domain characteristics.

3. *The business processes in which the system is used* As business processes evolve, they generate system change requests. As a system is integrated with more and more business processes, there are increased demands for changes.

In early work on software maintenance, researchers looked at the relationships between program complexity and maintainability (Banker et al. 1993; Coleman et al. 1994; Kozlov et al. 2008). These studies found that the more complex a system or component, the more expensive it is to maintain. Complexity measurements are particularly useful in identifying program components that are likely to be expensive to maintain. Therefore, to reduce maintenance costs you should try to replace complex system components with simpler alternatives.

After a system has been put into service, you may be able to use process data to help predict maintainability. Examples of process metrics that can be used for assessing maintainability are:

1. *Number of requests for corrective maintenance* An increase in the number of bug and failure reports may indicate that more errors are being introduced into the program than are being repaired during the maintenance process. This may indicate a decline in maintainability.

2. *Average time required for impact analysis* This is related to the number of program components that are affected by the change request. If the time required for impact analysis increases, it implies that more and more components are affected and maintainability is decreasing.

3. *Average time taken to implement a change request* This is not the same as the time for impact analysis although it may correlate with it. This is the amount of time that you need to modify the system and its documentation, after you have assessed which components are affected. An increase in the time needed to implement a change may indicate a decline in maintainability.

4. *Number of outstanding change requests* An increase in this number over time may imply a decline in maintainability.

You use predicted information about change requests and predictions about system maintainability to predict maintenance costs. Most managers combine this information with intuition and experience to estimate costs. The COCOMO 2 model of cost estimation, discussed in Chapter 23, suggests that an estimate for software maintenance effort can be based on the effort to understand existing code and the effort to develop the new code.

9.3.2 Software reengineering

Software maintenance involves understanding the program that has to be changed and then implementing any required changes. However, many systems, especially older legacy systems, are difficult to understand and change. The programs may have been optimized for performance or space utilization at the expense of understandability, or, over time, the initial program structure may have been corrupted by a series of changes.

To make legacy software systems easier to maintain, you can reengineer these systems to improve their structure and understandability. Reengineering may involve redocumenting the system, refactoring the system architecture, translating programs to a modern programming language, or modifying and updating the structure and values of the system's data. The functionality of the software is not changed, and, normally, you should try to avoid making major changes to the system architecture.

Reengineering has two important advantages over replacement:

1. *Reduced risk* There is a high risk in redeveloping business-critical software. Errors may be made in the system specification or there may be development problems. Delays in introducing the new software may mean that business is lost and extra costs are incurred.

2. *Reduced cost* The cost of reengineering may be significantly less than the cost of developing new software. Ulrich (Ulrich 1990) quotes an example of a commercial system for which the reimplementation costs were estimated at $50 million. The system was successfully reengineered for $12 million. I suspect that, with modern software technology, the relative cost of reimplementation is probably less than Ulrich's figure but will still be more than the costs of reengineering.

Figure 9.14 The reengineering process

Figure 9.14 is a general model of the reengineering process. The input to the process is a legacy program, and the output is an improved and restructured version of the same program. The activities in this reengineering process are:

1. *Source code translation* Using a translation tool, you can convert the program from an old programming language to a more modern version of the same language or to a different language.
2. *Reverse engineering* The program is analyzed and information extracted from it. This helps to document its organization and functionality. Again, this process is usually completely automated.
3. *Program structure improvement* The control structure of the program is analyzed and modified to make it easier to read and understand. This can be partially automated, but some manual intervention is usually required.
4. *Program modularization* Related parts of the program are grouped together, and, where appropriate, redundancy is removed. In some cases, this stage may involve architectural refactoring (e.g., a system that uses several different data stores may be refactored to use a single repository). This is a manual process.
5. *Data reengineering* The data processed by the program is changed to reflect program changes. This may mean redefining database schemas and converting existing databases to the new structure. You should usually also clean up the data. This involves finding and correcting mistakes, removing duplicate records, and so on. This can be a very expensive and prolonged process.

Program reengineering may not necessarily require all of the steps in Figure 9.11. You don't need source code translation if you still use the application's programming language. If you can do all reengineering automatically, then recovering documentation through reverse engineering may be unnecessary. Data reengineering is required only if the data structures in the program change during system reengineering.

Figure 9.15 Reengineering approaches

To make the reengineered system interoperate with the new software, you may have to develop adaptor services, as discussed in Chapter 18. These hide the original interfaces of the software system and present new, better-structured interfaces that can be used by other components. This process of legacy system wrapping is an important technique for developing large-scale reusable services.

The costs of reengineering obviously depend on the extent of the work that is carried out. There is a spectrum of possible approaches to reengineering, as shown in Figure 9.15. Costs increase from left to right so that source code translation is the cheapest option, and reengineering, as part of architectural migration, is the most expensive.

The problem with software reengineering is that there are practical limits to how much you can improve a system by reengineering. It isn't possible, for example, to convert a system written using a functional approach to an object-oriented system. Major architectural changes or radical reorganizing of the system data management cannot be carried out automatically, so they are very expensive. Although reengineering can improve maintainability, the reengineered system will probably not be as maintainable as a new system developed using modern software engineering methods.

9.3.3 Refactoring

Refactoring is the process of making improvements to a program to slow down degradation through change. It means modifying a program to improve its structure, reduce its complexity, or make it easier to understand. Refactoring is sometimes considered to be limited to object-oriented development, but the principles can in fact be applied to any development approach. When you refactor a program, you should not add functionality but rather should concentrate on program improvement. You can therefore think of refactoring as "preventative maintenance" that reduces the problems of future change.

Refactoring is an inherent part of agile methods because these methods are based on change. Program quality is liable to degrade quickly, so agile developers frequently refactor their programs to avoid this degradation. The emphasis on regression testing in agile methods lowers the risk of introducing new errors through refactoring. Any errors that are introduced should be detectable, as previously successful tests should then fail. However, refactoring is not dependent on other "agile activities."

Although reengineering and refactoring are both intended to make software easier to understand and change, they are not the same thing. Reengineering takes place after a system has been maintained for some time, and maintenance costs are increasing. You use automated tools to process and reengineer a legacy system to create a new system that is more maintainable. Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

Fowler et al. (Fowler et al. 1999) suggest that there are stereotypical situations (Fowler calls them "bad smells") where the code of a program can be improved. Examples of bad smells that can be improved through refactoring include:

1. *Duplicate code* The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.
2. *Long methods* If a method is too long, it should be redesigned as a number of shorter methods.
3. *Switch (case) statements* These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.
4. *Data clumping* Data clumps occur when the same group of data items (fields in classes, parameters in methods) reoccurs in several places in a program. These can often be replaced with an object that encapsulates all of the data.
5. *Speculative generality* This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

Fowler, in both his book and website, also suggests some primitive refactoring transformations that can be used singly or together to deal with bad smells. Examples of these transformations include Extract method, where you remove duplication and create a new method; Consolidate conditional expression, where you replace a sequence of tests with a single test; and Pull up method, where you replace similar methods in subclasses with a single method in a superclass. Interactive development environments, such as Eclipse, usually include refactoring support in their editors. This makes it easier to find dependent parts of a program that have to be changed to implement the refactoring.

Refactoring, carried out during program development, is an effective way to reduce the long-term maintenance costs of a program. However, if you take over a program for maintenance whose structure has been significantly degraded, then it may be practically impossible to refactor the code alone. You may also have to think about design refactoring, which is likely to be a more expensive and difficult problem. Design refactoring involves identifying relevant design patterns (discussed in Chapter 7) and replacing existing code with code that implements these design patterns (Kerievsky 2004).

KEY POINTS

- Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.
- For custom systems, the costs of software maintenance usually exceed the software development costs.
- The process of software evolution is driven by requests for changes and includes change impact analysis, release planning, and change implementation.
- Legacy systems are older software systems, developed using obsolete software and hardware technologies, that remain useful for a business.
- It is often cheaper and less risky to maintain a legacy system than to develop a replacement system using modern technology.
- The business value of a legacy system and the quality of the application software and its environment should be assessed to determine whether a system should be replaced, transformed, or maintained.
- There are three types of software maintenance, namely, bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.
- Software reengineering is concerned with restructuring and redocumenting software to make it easier to understand and change.
- Refactoring, making small program changes that preserve functionality, can be thought of as preventative maintenance.

FURTHER READING

Working Effectively with Legacy Code. Solid practical advice on the problems and difficulties of dealing with legacy systems. (M. Feathers, 2004, John Wiley & Sons).

"The Economics of Software Maintenance in the 21st Century." This article is a general introduction to maintenance and a comprehensive discussion of maintenance costs. Jones discusses the factors that affect maintenance costs and suggests that almost 75% of the software workforce are involved in software maintenance activities. (C. Jones, 2006) http://www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf

"You Can't Be Agile in Maintenance?" In spite of the title, this blog post argues that agile techniques are appropriate for maintenance and discusses which techniques as suggested in XP can be effective. (J. Bird, 2011) http://swreflections.blogspot.co.uk/2011/10/you-cant-be-agile-in-maintenance.html

"Software Reengineering and Testing Considerations." This is an excellent summary white paper of maintenance issues from a major Indian software company. (Y. Kumar and Dipti, 2012) http://www.infosys.com/engineering-services/white-papers/Documents/software-re-engineering-processes.pdf

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/implementation-and-evolution/

EXERCISES

9.1. Explain how advances in technology can force a software subsystem to undergo change or run the risk of becoming useless.

9.2. From Figure 9.4, you can see that impact analysis is an important subprocess in the software evolution process. Using a diagram, suggest what activities might be involved in change impact analysis.

9.3. Explain why legacy systems should be thought of as sociotechnical systems rather than simply software systems that were developed using old technology.

9.4. Some software subsystems are seen as "low quality, high business value." Discuss how those subsystems can be re-engineered with minimal impact on the operations of the organization.

9.5. What are the strategic options for legacy system evolution? When would you normally replace all or part of a system rather than continue maintenance of the software?

9.6. Explain why problems with support software might mean that an organization has to replace its legacy systems.

9.7. As a software project manager in a company that specializes in the development of software for the offshore oil industry, you have been given the task of discovering the factors that affect the maintainability of the systems developed by your company. Suggest how you might set up a program to analyze the maintenance process and determine appropriate maintainability metrics for the company.

9.8. Briefly describe the three main types of software maintenance. Why is it sometimes difficult to distinguish between them?

9.9. Explain the differences between software reengineering and refactoring?

9.10. Do software engineers have a professional responsibility to develop code that can be easily maintained even if their employer does not explicitly request it?

REFERENCES

Banker, R. D., S. M. Datar, C. F. Kemerer, and D. Zweig. 1993. "Software Complexity and Maintenance Costs." *Comm. ACM* 36 (11): 81–94. doi:10.1145/163359.163375.

Coleman, D., D. Ash, B. Lowther, and P. Oman. 1994. "Using Metrics to Evaluate Software System Maintainability." *IEEE Computer* 27 (8): 44–49. doi:10.1109/2.303623.

Davidsen, M. G., and J. Krogstie. 2010. "A Longitudinal Study of Development and Maintenance." *Information and Software Technology* 52 (7): 707–719. doi:10.1016/j.infsof.2010.03.003.

Erlikh, L. 2000. "Leveraging Legacy System Dollars for E-Business." *IT Professional* 2 (3 (May/June 2000)): 17–23. doi:10.1109/6294.846201.

Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley.

Hopkins, R., and K. Jenkins. 2008. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston: IBM Press.

Jones, T. C. 2006. "The Economics of Software Maintenance in the 21st Century." www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf.

Kerievsky, J. 2004. *Refactoring to Patterns*. Boston: Addison-Wesley.

Kozlov, D., J. Koskinen, M. Sakkinen, and J. Markkula. 2008. "Assessing Maintainability Change over Multiple Software Releases." *J. of Software Maintenance and Evolution* 20 (1): 31–58. doi:10.1002/smr.361.

Lientz, B. P., and E. B. Swanson. 1980. *Software Maintenance Management*. Reading, MA: Addison-Wesley.

Mitchell, R. M. 2012. "COBOL on the Mainframe: Does It Have a Future?" *Computerworld US*. http://features.techworld.com/applications/3344704/cobol-on-the-mainframe-does-it-have-a-future/

O'Hanlon, C. 2006. "A Conversation with Werner Vogels." *ACM Queue* 4 (4): 14–22. doi:10.1145/1142055.1142065.

Rajlich, V. T., and K. H. Bennett. 2000. "A Staged Model for the Software Life Cycle." *IEEE Computer* 33 (7): 66–71. doi:10.1109/2.869374.

Ulrich, W. M. 1990. "The Evolutionary Growth of Software Reengineering and the Decade Ahead." *American Programmer* 3 (10): 14–20.

Warren, I. (ed.). 1998. *The Renaissance of Legacy Systems*. London: Springer.

PART 2 Dependability and Security

As software systems are now part of all aspects of our lives, I believe that the most significant challenge that we face in software engineering is ensuring that we can trust these systems. To trust a system, we must have confidence that it will be available when required and perform as expected. It must be secure so that our computers or data are not threatened by it and it has to recover quickly in the event of failure or cyberattack. This part of the book has therefore focuses on the important topics of software system dependability and security.

Chapter 10 introduces the basic concepts of dependability and security namely reliability, availability, safety, security and resilience. I explain why building secure, dependable systems is not simply a technical problem. I introduce redundancy and diversity as the fundamental mechanisms used to create dependable and secure systems. The individual dependability attributes are covered in more detail in the following chapters.

Chapter 11 focuses on reliability and availability and I explain how these attributes can be specified as probabilities of failure or downtime. I discuss a number of architectural patterns for fault-tolerant system architectures and development techniques that can be used to reduce the number of faults in a system. In the final section, I explain how the reliability of a system may be tested and measured.

More and more systems are safety-critical systems, where system failure can endanger people. Chapter 12 is concerned with safety engineering and techniques that may be used to develop these safety-critical systems. I explain why safety is a broader notion than reliability and discuss methods for deriving system safety requirements. I also explain why defined and documented processes for safety-critical systems engineering are important and describe software safety cases—structured documents that are used to justify why a system is safe.

Threats to the security of our systems are one of the major problems faced by today's societies and I devote two chapters to this topic. Chapter 13 is concerned with application security engineering—methods used to achieve security in individual software systems. I explain the relationships between security and other dependability attributes and cover security requirements engineering, secure systems design and security testing.

Chapter 14 is a new chapter that addresses the broader issue of resilience. A resilient system can continue to deliver its essential services even when individual parts of the system fail or are subject to a cyberattack. I explain the basics of cybersecurity and discuss how resilience is achieved by using redundancy and diversity and by empowering people as well as through technical mechanisms. Finally, I discuss systems and software design issues that can contribute to improving the resilience of a system.

10

Dependable systems

Objectives

The objective of this chapter is to introduce the topic of software dependability and what is involved in developing dependable software systems. When you have read this chapter, you will:

- understand why dependability and security are important attributes for all software systems;
- understand the five important dimensions of dependability, namely, availability, reliability, safety, security, and resilience;
- understand the notion of sociotechnical systems and why we have to consider these systems as a whole rather than just software systems;
- know why redundancy and diversity are the fundamental concepts used in achieving dependable systems and processes;
- be aware of the potential for using formal methods in dependable systems engineering.

Contents

10.1 Dependability properties
10.2 Sociotechnical systems
10.3 Redundancy and diversity
10.4 Dependable processes
10.5 Formal methods and dependability

As computer systems have become deeply embedded in our business and personal lives, the problems that result from system and software failure are increasing. A failure of server software in an e-commerce company could lead to a major loss of revenue and customers for that company. A software error in an embedded control system in a car could lead to expensive recalls of that model for repair and, in the worst case, could be a contributory factor in accidents. The infection of company PCs with malware requires expensive clean-up operations to sort out the problem and could lead to the loss of or damage to sensitive information.

Because software-intensive systems are so important to governments, companies, and individuals, we have to be able to trust these systems. The software should be available when it is needed, and it should operate correctly without undesirable side effects, such as unauthorized information disclosure. In short, we should be able to depend on our software systems.

The term *dependability* was proposed by Jean-Claude Laprie in 1995 to cover the related systems attributes of availability, reliability, safety, and security. His ideas were revised over the next few years and are discussed in a definitive paper published in 2004 (Avizienis et al. 2004). As I discuss in Section 10.1, these properties are inextricably linked, so having a single term to cover them all makes sense.

The dependability of systems is usually more important than their detailed functionality for the following reasons:

1. *System failures affect a large number of people* Many systems include functionality that is rarely used. If this functionality were left out of the system, only a small number of users would be affected. System failures that affect the availability of a system potentially affect all users of the system. Unavailable systems may mean that normal business is impossible.

2. *Users often reject systems that are unreliable, unsafe, or insecure* If users find that a system is unreliable or insecure, they will refuse to use it. Furthermore, they may also refuse to buy or use other products from the company that produced the unreliable system. They do not want a repetition of their bad experience with an undependable system.

3. *System failure costs may be enormous* For some applications, such as a reactor control system or an aircraft navigation system, the cost of system failure is orders of magnitude greater than the cost of the control system. Failures in systems that control critical infrastructure such as the power network have widespread economic consequences.

4. *Undependable systems may cause information loss* Data is very expensive to collect and maintain; it is usually worth much more than the computer system on which it is processed. The cost of recovering lost or corrupt data is usually very high.

However, a system can be useful without it being very dependable. I don't think that the word processor that I used to write this book is a very dependable system. It sometimes freezes and has to be restarted. Nevertheless, because it is very useful,

Critical systems

Some classes of system are "critical systems" where system failure may result in injury to people, damage to the environment, or extensive economic losses. Examples of critical systems include embedded systems in medical devices, such as an insulin pump (safety-critical), spacecraft navigation systems (mission-critical), and online money transfer systems (business critical).

Critical systems are very expensive to develop. Not only must they be developed so that failures are very rare, but they must also include recovery mechanisms to be used if and when failures occur.

http://software-engineering-book.com/web/critical-systems/

I am prepared to tolerate occasional failure. However, to reflect my lack of trust in the system, I save my work frequently and keep multiple backup copies of it. I compensate for the lack of system dependability by actions that limit the damage that could result from system failure.

Building dependable software is part of the more general process of dependable systems engineering. As I discuss in Section 10.2, software is always part of a broader system. It executes in an operational environment that includes the hardware on which the software executes, the human users of that software and the organizational or business processes where the software is used. When designing a dependable system, you therefore have to consider:

1. *Hardware failure* System hardware may fail because of mistakes in its design, because components fail as a result of manufacturing errors, because of environmental factors such as dampness or high temperatures, or because components have reached the end of their natural life.
2. *Software failure* System software may fail because of mistakes in its specification, design, or implementation.
3. *Operational failure* Human users may fail to use or operate the system as intended by its designers. As hardware and software have become more reliable, failures in operation are now, perhaps, the largest single cause of system failures.

These failures are often interrelated. A failed hardware component may mean system operators have to cope with an unexpected situation and additional workload. This puts them under stress, and people under stress often make mistakes. These mistakes can cause the software to fail, which means more work for operators, even more stress, and so on.

As a result, it is particularly important that designers of dependable, software-intensive systems take a holistic sociotechnical systems perspective rather than focus on a single aspect of the system such as its software or hardware. If hardware, software, and operational processes are designed separately, without taking into account the potential weaknesses of other parts of the system, then it is more likely that errors will occur at the interfaces between the different parts of the system.

10.1 Dependability properties

All of us are familiar with the problem of computer system failure. For no obvious reason, our computers sometimes crash or go wrong in some way. Programs running on these computers may not operate as expected and occasionally may corrupt the data that is managed by the system. We have learned to live with these failures, but few of us completely trust the personal computers that we normally use.

The dependability of a computer system is a property of the system that reflects its trustworthiness. Trustworthiness here essentially means the degree of confidence a user has that the system will operate as they expect and that the system will not "fail" in normal use. It is not meaningful to express dependability numerically. Rather, relative terms such as "not dependable," "very dependable," and "ultra-dependable" can reflect the degree of trust that we might have in a system.

There are five principal dimensions to dependability, as I have shown in Figure 10.1.

1. *Availability* Informally, the availability of a system is the probability that it will be up and running and able to deliver useful services to users at any given time.
2. *Reliability* Informally, the reliability of a system is the probability, over a given period of time, that the system will correctly deliver services as expected by the user.
3. *Safety* Informally, the safety of a system is a judgment of how likely it is that the system will cause damage to people or its environment.
4. *Security* Informally, the security of a system is a judgment of how likely it is that the system can resist accidental or deliberate intrusions.
5. *Resilience* Informally, the resilience of a system is a judgment of how well that system can maintain the continuity of its critical services in the presence of disruptive events, such as equipment failure and cyberattacks. Resilience is a more recent addition to the set of dependability properties that were originally suggested by Laprie.

The dependability properties shown in Figure 10.1 are complex properties that can be broken down into several simpler properties. For example, security includes "integrity" (ensuring that the systems program and data are not damaged) and "confidentiality" (ensuring that information can only be accessed by people who are authorized). Reliability includes "correctness" (ensuring the system services are as specified), "precision" (ensuring information is delivered at an appropriate level of detail), and "timeliness" (ensuring that information is delivered when it is required).

Of course, not all dependability properties are critical for all systems. For the insulin pump system, introduced in Chapter 1, the most important properties are reliability (it must deliver the correct dose of insulin) and safety (it must never deliver a dangerous dose of insulin). Security is not an issue as the pump does not store confidential information. It is not networked and so cannot be maliciously attacked. For

Figure 10.1 Principal dependability properties

the wilderness weather system, availability and reliability are the most important properties because the costs of repair may be very high. For the Mentcare patient information system, security and resilience are particularly important because of the sensitive private data that is maintained and the need for the system to be available for patient consultations.

Other system properties are closely related to these five dependability properties and influence a system's dependability:

1. *Repairability* System failures are inevitable, but the disruption caused by failure can be minimized if the system can be repaired quickly. It must be possible to diagnose the problem, access the component that has failed, and make changes to fix that component. Repairability in software is enhanced when the organization using the system has access to the source code and has the skills to make changes to it. Open-source software makes this easier, but the reuse of components can make it more difficult.

2. *Maintainability* As systems are used, new requirements emerge, and it is important to maintain the value of a system by changing it to include these new requirements. Maintainable software is software that can be adapted economically to cope with new requirements, and where there is a low probability that making changes will introduce new errors into the system.

3. *Error tolerance* This property can be considered as part of usability and reflects the extent to which the system has been designed, so that user input errors are avoided and tolerated. When user errors occur, the system should, as far as possible, detect these errors and either fix them automatically or request the user to re-input their data.

The notion of system dependability as an encompassing property was introduced because the dependability properties of availability, security, reliability, safety, and resilience are closely related. Safe system operation usually depends on the system being available and operating reliably. A system may become unreliable because an intruder has corrupted its data. Denial-of-service attacks on a system are intended to

compromise the system's availability. If a system is infected with a virus, you cannot then be confident in its reliability or safety because the virus may change its behavior.

To develop dependable software, you therefore need to ensure that:

1. You avoid the introduction of accidental errors into the system during software specification and development.
2. You design verification and validation processes that are effective in discovering residual errors that affect the dependability of the system.
3. You design the system to be fault tolerant so that it can continue working when things go wrong.
4. You design protection mechanisms that guard against external attacks that can compromise the availability or security of the system.
5. You configure the deployed system and its supporting software correctly for its operating environment.
6. You include system capabilities to recognize external cyberattacks and to resist these attacks.
7. You design systems so that they can quickly recover from system failures and cyberattacks without the loss of critical data.

The need for fault tolerance means that dependable systems have to include redundant code to help them monitor themselves, detect erroneous states, and recover from faults before failures occur. This affects the performance of systems, as additional checking is required each time the system executes. Therefore, designers usually have to trade off performance and dependability. You may need to leave checks out of the system because these slow the system down. However, the consequential risk here is that the system fails because a fault has not been detected.

Building dependable systems is expensive. Increasing the dependability of a system means that you incur extra costs for system design, implementation, and validation. Verification and validation costs are particularly high for systems that must be ultra-dependable such as safety-critical control systems. As well as validating that the system meets its requirements, the validation process may have to prove to an external regulator that the system is safe. For example, aircraft systems have to demonstrate to regulators, such as the Federal Aviation Authority, that the probability of a catastrophic system failure that affects aircraft safety is extremely low.

Figure 10.2 shows the relationship between costs and incremental improvements in dependability. If your software is not very dependable, you can get significant improvements fairly cheaply by using better software engineering. However, if you are already using good practice, the costs of improvement are much greater, and the benefits from that improvement are less.

There is also the problem of testing software to demonstrate that it is dependable. Solving this problem relies on running many tests and looking at the number of failures that occur. As your software becomes more dependable, you see fewer and

Figure 10.2 Cost/dependability curve

fewer failures. Consequently, more and more tests are needed to try and assess how many problems remain in the software. Testing is a very expensive process, so this can significantly increase the cost of high-dependability systems.

10.2 Sociotechnical systems

In a computer system, the software and the hardware are interdependent. Without hardware, a software system is an abstraction, which is simply a representation of some human knowledge and ideas. Without software, hardware is a set of inert electronic devices. However, if you put them together to form a system, you create a machine that can carry out complex computations and deliver the results of these computations to its environment.

This illustrates one of the fundamental characteristics of a system—it is more than the sum of its parts. Systems have properties that become apparent only when their components are integrated and operate together. Software systems are not isolated systems but are part of more extensive systems that have a human, social, or organizational purpose. Therefore software engineering is not an isolated activity but is an intrinsic part of systems engineering (Chapter 19).

For example, the wilderness weather system software controls the instruments in a weather station. It communicates with other software systems and is a part of wider national and international weather forecasting systems. As well as hardware and software, these systems include processes for forecasting the weather and people who operate the system and analyze its outputs. The system also includes the organizations that depend on the system to help them provide weather forecasts to individuals, government and industry.

Figure 10.3 The sociotechnical systems stack

These broader systems are called *sociotechnical systems*. They include nontechnical elements such as people, processes, and regulations, as well as technical components such as computers, software, and other equipment. System dependability is influenced by all of the elements in a sociotechnical system—hardware, software, people, and organizations.

Sociotechnical systems are so complex that it is impossible to understand them as a whole. Rather, you have to view them as layers, as shown in Figure 10.3. These layers make up the sociotechnical systems stack:

1. *The equipment layer* is composed of hardware devices, some of which may be computers.
2. *The operating system layer* interacts with the hardware and provides a set of common facilities for higher software layers in the system.
3. *The communications and data management layer* extends the operating system facilities and provides an interface that allows interaction with more extensive functionality, such as access to remote systems and access to a system database. This is sometimes called middleware, as it is in between the application and the operating system.
4. *The application layer* delivers the application-specific functionality that is required. There may be many different application programs in this layer.
5. *The business process layer* includes the organizational business processes, which make use of the software system.
6. *The organizational layer* includes higher-level strategic processes as well as business rules, policies, and norms that should be followed when using the system.
7. *The social layer* refers to the laws and regulations of society that govern the operation of the system.

Notice that there is no separate "software layer." Software of one kind or another is an important part of all of the layers in the sociotechnical system. Equipment is controlled by embedded software; the operating system and applications are software. Business processes, organizations, and society rely on the Internet (software) and other global software systems.

In principle, most interactions should be between neighboring layers in the stack, with each layer hiding the detail of the layer below from the layer above. In practice, however, there can be unexpected interactions between layers, which result in problems for the system as a whole. For example, say there is a change in the law governing access to personal information. This comes from the social layer. It leads to new organizational procedures and changes to the business processes. The application system itself may not be able to provide the required level of privacy, so changes may have to be implemented in the communications and data management layer.

Thinking holistically about systems, rather than simply considering software in isolation, is essential when considering software security and dependability. Software itself is intangible and, even when damaged, is easily and cheaply restored. However, when these software failures ripple through other parts of the system, they affect the software's physical and human environment. Here, the consequences of failure are more significant. Important data may be lost or corrupted. People may have to do extra work to contain or recover from the failure; for example, equipment may be damaged, data may be lost or corrupted, or confidentiality may be breached, with unknown consequences.

You must, therefore, take a system-level view when you are designing software that has to be dependable and secure. You have to take into account the consequences of software failures for other elements in the system. You also need to understand how these other system elements may be the cause of software failure and how they can help to protect against and recover from software failures.

It is important to ensure that, wherever possible, software failure does not lead to overall system failure. You must therefore examine how the software interacts with its immediate environment to ensure that:

1. Software failures are, as far as possible, contained within the enclosing layer of the system stack and do not seriously affect the operation of other layers in the system.
2. You understand how faults and failures in the other layers of the systems stack may affect the software. You may also consider how checks may be built into the software to help detect these failures, and how support can be provided for recovering from failure.

As software is inherently flexible, unexpected system problems are often left to software engineers to solve. Say a radar installation has been sited so that ghosting of the radar image occurs. It is impractical to move the radar to a site with less interference, so the systems engineers have to find another way of removing this

ghosting. Their solution may be to enhance the image-processing capabilities of the software to remove the ghost images. This may slow down the software so that its performance becomes unacceptable. The problem may then be characterized as a software failure, whereas, in fact, it is a failure in the design process for the system as a whole.

This sort of situation, in which software engineers are left with the problem of enhancing software capabilities without increasing hardware cost, is very common. Many so-called software failures are not a consequence of inherent software problems but rather are the result of trying to change the software to accommodate modified system engineering requirements. A good example was the failure of the Denver airport baggage system (Swartz 1996), where the controlling software was expected to deal with limitations of the equipment used.

10.2.1 Regulation and compliance

The general model of economic organization that is now almost universal in the world is that privately owned companies offer goods and services and make a profit on these. We have a competitive environment so that these companies may compete on cost, on quality, on delivery time, and so on. However, to ensure the safety of their citizens, most governments limit the freedom of privately owned companies so that they must follow certain standards to ensure that their products are safe and secure. A company therefore cannot offer products for sale more cheaply because they have reduced their costs by reducing the safety of their products.

Governments have created a set of rules and regulations in different areas that define standards for safety and security. They have also established regulators or regulatory bodies whose job is to ensure that companies offering products in an area comply with these rules. Regulators have wide powers. They can fine companies and even imprison directors if regulations are breached. They may have a licensing role (e.g., in the aviation and nuclear industries) where they must issue a license before a new system may be used. Therefore, aircraft manufacturers have to have a certificate of airworthiness from the regulator in each country where the aircraft is used.

To achieve certification, companies that are developing safety-critical systems have to produce an extensive safety case (discussed in Chapter 13) that shows that rules and regulations have been followed. The case must convince a regulator that the system can operate safely. Developing such a safety case is very costly. It can be as expensive to develop the documentation for certification as it is to develop the system itself.

Regulation and compliance (following the rules) applies to the sociotechnical system as a whole and not simply the software element of that system. For example, a regulator in the nuclear industry is concerned that in the event of overheating, a nuclear reactor will not release radioactivity into the environment. Arguments to convince the regulator that this is the case may be based on software protection systems, the operational process used to monitor the reactor core and the integrity of structures that contain any release of radioactivity.

Each of these elements has to have its own safety case. So, the protection system must have a safety case that demonstrates that the software will operate correctly and shut down the reactor as intended. The overall case must also show that if the software protection system fails, there are alternative safety mechanisms, which do not rely on software, that are invoked.

10.3 Redundancy and diversity

Component failures in any system are inevitable. People make mistakes, undiscovered bugs in software cause undesirable behavior, and hardware burns out. We use a range of strategies to reduce the number of human failures such as replacing hardware components before the end of their predicted lifetime and checking software using static analysis tools. However, we cannot be sure that these will eliminate component failures. We should therefore design systems so that individual component failures do not lead to overall system failure.

Strategies to achieve and enhance dependability rely on both redundancy and diversity. Redundancy means that spare capacity is included in a system that can be used if part of that system fails. Diversity means that redundant components of the system are of different types, thus increasing the chances that they will not fail in exactly the same way.

We use redundancy and diversity to enhance dependability in our everyday lives. Commonly, to secure our homes we use more than one lock (redundancy), and, usually, the locks used are of different types (diversity). This means that if intruders find a way to defeat one of the locks, they have to find a different way of defeating the other locks before they can gain entry. As a matter of routine, we should all back up our computers and so maintain redundant copies of our data. To avoid problems with disk failure, backups should be kept on a separate, diverse, external device.

Software systems that are designed for dependability may include redundant components that provide the same functionality as other system components. These are switched into the system if the primary component fails. If these redundant components are diverse, that is, not the same as other components, a common fault in replicated components will not result in a system failure. Another form of redundancy is the inclusion of checking code, which is not strictly necessary for the system to function. This code can detect some kinds of problems, such as data corruption, before they cause failures. It can invoke recovery mechanisms to correct problems to ensure that the system continues to operate.

In systems for which availability is a critical requirement, redundant servers are normally used. These automatically come into operation if a designated server fails. Sometimes, to ensure that attacks on the system cannot exploit a common vulnerability, these servers may be of different types and may run different operating systems. Using different operating systems is an example of software diversity and

The Ariane 5 explosion

In 1996, the European Space Agency's Ariane 5 rocket exploded 37 seconds after lift-off on its maiden flight. The fault was caused by a software systems failure. There was a backup system but it was not diverse, and so the software in the backup computer failed in exactly the same way. The rocket and its satellite payload were destroyed.

http://software-engineering-book.com/web/ariane/

redundancy, where similar functionality is provided in different ways. (I discuss software diversity in more detail in Chapter 12.)

Diversity and redundancy may also be also used in the design of dependable software development processes. Dependable development processes avoid the introduction of faults into a system. In a dependable process, activities such as software validation do not rely on a single tool or technique. This improves software dependability because it reduces the chances of process failure, where human errors made during the software development process lead to software errors.

For example, validation activities may include program testing, manual program inspections, and static analysis as fault-finding techniques. Any one of these techniques might find faults that are missed by the other methods. Furthermore, different team members may be responsible for the same process activity (e.g., a program inspection). People tackle tasks in different ways depending on their personality, experience, and education, so this kind of redundancy provides a diverse perspective on the system.

However, as I discuss in Chapter 11, using software redundancy and diversity can itself introduce bugs into software. Diversity and redundancy make systems more complex and usually harder to understand. Not only is there more code to write and check, but additional functionality must also be added to the system to detect component failure and to switch control to alternative components. This additional complexity means that it is more likely that programmers will make errors and less likely that people checking the system will find these errors.

Some engineers therefore think that, as software cannot wear out, it is best to avoid software redundancy and diversity. Their view is that the best approach is to design the software to be as simple as possible, with extremely rigorous software verification and validation procedures (Parnas, van Schouwen, and Shu 1990). More can be spent on verification and validation because of the savings that result from not having to develop redundant software components.

Both approaches are used in commercial, safety-critical software systems. For example, the Airbus 340 flight control hardware and software is both diverse and redundant. The flight control software on the Boeing 777 runs on redundant hardware, but each computer runs the same software, which has been very extensively validated. The Boeing 777 flight control system designers have focused on simplicity rather than redundancy. Both of these aircraft are very reliable, so both the diverse and the simple approach to dependability can clearly be successful.

Dependable operational processes

This chapter discusses dependable development processes, but system operational processes are equally important contributors for system dependability. In designing these operational processes, you have to take into account human factors and always bear in mind that people are liable to make mistakes when using a system. A dependable process should be designed to avoid human errors, and, when mistakes are made, the software should detect the mistakes and allow them to be corrected.

http://software-engineering-book.com/web/human-error/

10.4 Dependable processes

Dependable software processes are software processes that are designed to produce dependable software. The rationale for investing in dependable processes is that a good software process is likely to lead to delivered software that contains fewer errors and is therefore less likely to fail in execution. A company using a dependable process can be sure that the process has been properly enacted and documented and that appropriate development techniques have been used for critical systems development. Figure 10.4 shows some of the attributes of dependable software processes.

The evidence that a dependable process has been used is often important in convincing a regulator that the most effective software engineering practice has been applied in developing the software. System developers will normally present a model of the process to a regulator, along with evidence that the process has been followed. The regulator also has to be convinced that the process is used consistently by all of the process participants and that it can be used in different development projects. This means that the process must be explicitly defined and repeatable:

1. An explicitly defined process is one that has a defined process model that is used to drive the software production process. Data must be collected during the process that proves that the development team has followed the process as defined in the process model.

2. A repeatable process is one that does not rely on individual interpretation and judgment. Rather, the process can be repeated across projects and with different team members, irrespective of who is involved in the development. This is particularly important for critical systems, which often have a long development cycle during which there are often significant changes in the development team.

Dependable processes make use of redundancy and diversity to achieve reliability. They often include different activities that have the same aim. For example, program inspections and testing aim to discover errors in a program. The approaches can be used together so that they are likely to find more errors than would be found using one technique on its own.

Process characteristic	Description
Auditable	The process should be understandable by people apart from process participants, who can check that process standards are being followed and make suggestions for process improvement.
Diverse	The process should include redundant and diverse verification and validation activities.
Documentable	The process should have a defined process model that sets out the activities in the process and the documentation that is to be produced during these activities.
Robust	The process should be able to recover from failures of individual process activities.
Standardized	A comprehensive set of software development standards covering software production and documentation should be available.

Figure 10.4 Attributes of dependable processes

The activities that are used in dependable processes obviously depend on the type of software that is being developed. In general, however, these activities should be geared toward avoiding the introduction of errors into a system, detecting and removing errors, and maintaining information about the process itself.

Examples of activities that might be included in a dependable process include:

1. Requirements reviews to check that the requirements are, as far as possible, complete and consistent.
2. Requirements management to ensure that changes to the requirements are controlled and that the impact of proposed requirements changes is understood by all developers affected by the change.
3. Formal specification, where a mathematical model of the software is created and analyzed. (I discussed the benefits of formal specification in Section 10.5.) Perhaps its most important benefit is that it forces a very detailed analysis of the system requirements. This analysis itself is likely to discover requirements problems that may have been missed in requirements reviews.
4. System modeling, where the software design is explicitly documented as a set of graphical models and the links between the requirements and these models are explicitly documented. If a model-driven engineering approach is used (see Chapter 5), code may be generated automatically from these models.
5. Design and program inspections, where the different descriptions of the system are inspected and checked by different people. A checklist of common design and programming errors may be used to focus the inspection process.
6. Static analysis, where automated checks are carried out on the source code of the program. These look for anomalies that could indicate programming errors or omissions. (I cover static analysis in Chapter 12.)
7. Test planning and management, where a comprehensive set of system tests is designed. The testing process has to be carefully managed to demonstrate that these tests provide coverage of the system requirements and have been correctly applied in the testing process.

As well as process activities that focus on system development and testing, there must also be well-defined quality management and change management processes. While the specific activities in a dependable process may vary from one company to another, the need for effective quality and change management is universal.

Quality management processes (covered in Chapter 24) establish a set of process and product standards. They also include activities that capture process information to demonstrate that these standards have been followed. For example, there may be a standard defined for carrying out program inspections. The inspection team leader is responsible for documenting the process to show that the inspection standard has been followed.

Change management, discussed in Chapter 25, is concerned with managing changes to a system, ensuring that accepted changes are actually implemented, and confirming that planned releases of the software include the planned changes. One common problem with software is that the wrong components are included in a system build. This can lead to a situation where an executing system includes components that have not been checked during the development process. Configuration management procedures must be defined as part of the change management process to ensure that this does not happen.

As agile methods have become increasingly used, researchers and practitioners have thought carefully about how to use agile approaches in dependable software development (Trimble 2012). Most companies that develop critical software systems have based their development on plan-based processes and have been reluctant to make radical changes to their development process. However, they recognize the value of agile approaches and are exploring how their dependable development processes can be more agile.

As dependable software often requires certification, both process and product documentation have to be produced. Up-front requirements analysis is also essential to discover possible requirements and requirements conflicts that may compromise the safety and security of the system. Formal change analysis is essential to assess the effect of changes on the safety and integrity of the system. These requirements conflict with the general approach in agile development of co-development of the requirements and the system and minimizing documentation.

Although most agile development uses an informal, undocumented process, this is not a fundamental requirement of agility. An agile process may be defined that incorporates techniques such as iterative development, test-first development and user involvement in the development team. As long as the team follows that process and documents their actions, agile techniques can be used. To support this notion, various proposals of modified agile methods have been made that incorporate the requirements of dependable systems engineering (Douglass 2013). These combine the most appropriate techniques from agile and plan-based development.

10.5 Formal methods and dependability

For more than 30 years, researchers have advocated the use of formal methods of software development. Formal methods are mathematical approaches to software development where you define a formal model of the software. You may then formally analyze this model to search for errors and inconsistencies, prove that a program

is consistent with this model, or you may apply a series of correctness-preserving transformations to the model to generate a program. Abrial (Abrial 2009) claims that the use of formal methods can lead to "faultless systems," although he is careful to limit what he means in this claim.

In an excellent survey, Woodcock et al. (Woodcock et al. 2009) discuss industrial applications where formal methods have been successfully applied. These include train control systems (Badeau and Amelot 2005), cash card systems (Hall and Chapman 2002), and flight control systems (Miller et al. 2005). Formal methods are the basis of tools used in static verification, such as the driver verification system used by Microsoft (Ball et al. 2006).

Using a mathematically formal approach to software development was proposed at an early stage in the development of computer science. The idea was that a formal specification and a program could be developed independently. A mathematical proof could then be developed to show that the program and its specification were consistent. Initially, proofs were developed manually but this was a long and expensive process. It quickly became clear that manual proofs could only be developed for very small systems. Program proving is now supported by large-scale automated theorem proving software, which has meant that larger systems can be proved. However, developing the proof obligations for theorem provers is a difficult and specialized task, so formal verification is not widely used.

An alternative approach, which avoids a separate proof activity, is refinement-based development. Here, a formal specification of a system is refined through a series of correctness-preserving transformations to generate the software. Because these are trusted transformations, you can be confident that the generated program is consistent with its formal specification. This was the approach used in the software development for the Paris Metro system (Badeau and Amelot 2005). It used a language called B (Abrial 2010), which was designed to support specification refinement.

Formal methods based on model-checking (Jhala and Majumdar 2009) have been used in a number of systems (Bochot et al. 2009; Calinescu and Kwiatkowska 2009). These systems rely on constructing or generating a formal state model of a system and using a model-checker to check that properties of the model, such as safety properties, always hold. The model-checking program exhaustively analyzes the specification and either reports that the system property is satisfied by the model or presents an example that shows it is not satisfied. If a model can be automatically or systematically generated from a program, this means that bugs in the program can be uncovered. (I cover model checking in safety-critical systems in Chapter 12.)

Formal methods for software engineering are effective for discovering or avoiding two classes of error in software representations:

1. *Specification and design errors and omissions.* The process of developing and analyzing a formal model of the software may reveal errors and omissions in the software requirements. If the model is generated automatically or systematically from source code, analysis using model checking can discover undesirable states that may occur, such as deadlock in a concurrent system.

Formal specification techniques

Formal system specifications may be expressed using two fundamental approaches, either as models of the system interfaces (algebraic specifications) or as models of the system state. An extra web chapter on this topic shows examples of both of these approaches. The chapter includes a formal specification of part of the insulin pump system.

http://software-engineering-book.com/web/formal-methods/ (web chapter)

2. *Inconsistencies between a specification and a program.* If a refinement method is used, mistakes made by developers that make the software inconsistent with the specification are avoided. Program proving discovers inconsistencies between a program and its specification.

The starting point for all formal methods is a mathematical system model, which acts as a system specification. To create this model, you translate the system's user requirements, which are expressed in natural language, diagrams, and tables, into a mathematical language that has formally defined semantics. The formal specification is an unambiguous description of what the system should do.

Formal specifications are the most precise way of specifying systems, and so reduce the scope for misunderstanding. Many supporters of formal methods believe that creating formal specification, even without refinement or program proof, is worthwhile. Constructing a formal specification forces a detailed analysis of the requirements and this is an effective way of discovering requirements problems. In a natural language specification, errors can be concealed by the imprecision of the language. This is not the case if the system is formally specified.

The advantages of developing a formal specification and using it in a formal development process are:

1. As you develop a formal specification in detail, you develop a deep and detailed understanding of the system requirements. Requirements problems that are discovered early are usually much cheaper to correct than if they are found at later stages in the development process.

2. As the specification is expressed in a language with formally defined semantics, you can analyze it automatically to discover inconsistencies and incompleteness.

3. If you use a method such as the B method, you can transform the formal specification into a program through a sequence of correctness-preserving transformations. The resulting program is therefore guaranteed to meet its specification.

4. Program testing costs may be reduced because you have verified the program against its specification. For example, in the development of the software for the Paris Metro systems, the use of refinement meant that there was no need for software component testing and only system testing was required.

Woodcock's survey (Woodcock et al. 2009) found that users of formal methods reported fewer errors in the delivered software. Neither the costs nor the time needed for software development were higher than in comparable development projects. There were significant benefits in using formal approaches in safety critical systems that required regulator certification. The documentation produced was an important part of the safety case (see Chapter 12) for the system.

In spite of these advantages, formal methods have had limited impact on practical software development, even for critical systems. Woodcock reports on 62 projects over 25 years that used formal methods. Even if we allow for projects that used these techniques but did not report their use, this is a tiny fraction of the total number of critical systems developed in that time. Industry has been reluctant to adopt formal methods for a number of reasons:

1. Problem owners and domain experts cannot understand a formal specification, so they cannot check that it accurately represents their requirements. Software engineers, who understand the formal specification, may not understand the application domain, so they too cannot be sure that the formal specification is an accurate reflection of the system requirements.
2. It is fairly easy to quantify the costs of creating a formal specification, but more difficult to estimate the possible cost savings that will result from its use. As a result, managers are unwilling to take the risk of adopting formal methods. They are unconvinced by reports of success as, by and large, these came from atypical projects where the developers were keen advocates of a formal approach.
3. Most software engineers have not been trained to use formal specification languages. Hence, they are reluctant to propose their use in development processes.
4. It is difficult to scale current formal methods up to very large systems. When formal methods are used, it is mostly for specifying critical kernel software rather than complete systems.
5. Tool support for formal methods is limited, and the available tools are often open source and difficult to use. The market is too small for commercial tool providers.
6. Formal methods are not compatible with agile development where programs are developed incrementally. This is not a major issue, however, as most critical systems are still developed using a plan-based approach.

Parnas, an early advocate of formal development, has criticized current formal methods and claims that these have started from a fundamentally wrong premise (Parnas 2010). He believes that these methods will not gain acceptance until they are radically simplified, which will require a different type of mathematics as a basis. My own view is that even this will not mean that formal methods are routinely adopted for critical systems engineering unless it can be clearly demonstrated that their adoption and use is cost-effective, compared to other software engineering methods.

KEY POINTS

- System dependability is important because failure of critical computer systems can lead to large economic losses, serious information loss, physical damage or threats to human life.
- The dependability of a computer system is a system property that reflects the user's degree of trust in the system. The most important dimensions of dependability are availability, reliability, safety, security, and resilience.
- Sociotechnical systems include computer hardware, software, and people, and are situated within an organization. They are designed to support organizational or business goals and objectives.
- The use of a dependable, repeatable process is essential if faults in a system are to be minimized. The process should include verification and validation activities at all stages, from requirements definition through to system implementation.
- The use of redundancy and diversity in hardware, software processes, and software systems is essential to the development of dependable systems.
- Formal methods, where a formal model of a system is used as a basis for development, help reduce the number of specification and implementation errors in a system. However, formal methods have had a limited take-up in industry because of concerns about the cost-effectiveness of this approach.

FURTHER READING

"Basic Concepts and Taxonomy of Dependable and Secure Computing." This work presents a thorough discussion of dependability concepts written by some of the pioneers in the field who were responsible for developing these ideas. (A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr., *IEEE Transactions on Dependable and Secure Computing*, 1 (1), 2004) http://dx.doi.org/10.1109/TDSC.2004.2

Formal Methods: Practice and Experience. An excellent survey of the use of formal methods in industry, along with a description of some projects that have used formal methods. The authors present a realistic summary of the barriers to the use of these methods. (J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. *Computing Surveys,* 41 (1) January 2009) http://dx.doi.org/10.1145/1592434.1592436

The LSCITS Socio-technical Systems Handbook. This handbook introduces sociotechnical systems in an accessible way and provides access to more detailed papers on sociotechnical topics. (2012) http://archive.cs.st-andrews.ac.uk/STSE-Handbook/

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/critical-systems/

EXERCISES

10.1. Suggest six reasons why software dependability is important in most sociotechnical systems.

10.2. Explain with an example why resilience to cyber attacks is a very important characteristic of system dependability.

10.3. Using an example, explain why it is important when developing dependable systems to consider these as sociotechnical systems and not simply as technical software and hardware systems.

10.4. Give two examples of government functions that are supported by complex sociotechnical systems and explain why, in the foreseeable future, these functions cannot be completely automated.

10.5. Explain the difference between redundancy and diversity.

10.6. Explain why it is reasonable to assume that the use of dependable processes will lead to the creation of dependable software.

10.7. Give two examples of diverse, redundant activities that might be incorporated into dependable processes.

10.8. Give two reasons why different versions of a system based on software diversity may fail in a similar way.

10.9. You are an engineer in charge of the development of a small, safety-critical train control system, which must be demonstrably safe and secure. You suggest that formal methods should be used in the development of this system, but your manager is skeptical of this approach. Write a report highlighting the benefits of formal methods and presenting a case for their use in this project.

10.10. It has been suggested that the need for regulation inhibits innovation and that regulators force the use of older methods of systems development that have been used on other systems. Discuss whether or not you think this is true and the desirability of regulators imposing their views on what methods should be used.

REFERENCES

Abrial, J. R. 2009. "Faultless Systems: Yes We Can." *IEEE Computer* 42 (9): 30–36. doi:10.1109/MC.2009.283.

———. 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.

Avizienis, A., J. C. Laprie, B. Randell, and C. Landwehr. 2004. "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Trans. on Dependable and Secure Computing* 1 (1): 11–33. doi:10.1109/TDSC.2004.2.

Badeau, F., and A. Amelot. 2005. "Using B as a High Level Programming Language in an Industrial Project: Roissy VAL." In *Proc. ZB 2005: Formal Specification and Development in Z and B*. Guildford, UK: Springer. doi:10.1007/11415787_20.

Ball, T., E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. 2006. "Thorough Static Analysis of Device Drivers." In *Proc. EuroSys 2006*. Leuven, Belgium. doi:10.1145/1218063.1217943.

Bochot, T., P. Virelizier, H. Waeselynck, and V. Wiels. 2009. "Model Checking Flight Control Systems: The Airbus Experience." In *Proc. 31st International Conf. on Software Engineering, Companion Volume*, 18–27. Leipzig: IEEE Computer Society Press. doi:10.1109/ICSE-COMPANION.2009.5070960.

Calinescu, R. C., and M. Z. Kwiatkowska. 2009. "Using Quantitative Analysis to Implement Autonomic IT Systems." In *Proc. 31st International Conf. on Software Engineering, Companion Volume*, 100–10. Leipzig: IEEE Computer Society Press. doi:10.1109/ICSE.2009.5070512.

Douglass, B. 2013. "Agile Analysis Practices for Safety-Critical Software Development." http://www.ibm.com/developerworks/rational/library/agile-analysis-practices-safety-critical-development/.

Hall, A., and R. Chapman. 2002. "Correctness by Construction: Developing a Commercially Secure System." *IEEE Software* 19 (1): 18–25.doi:10.1109/52.976937.

Jhala, R., and R. Majumdar. 2009. "Software Model Checking." *Computing Surveys* 41 (4), Article 21. doi:1145/1592434.1592438.

Miller, S. P., E. A. Anderson, L. G. Wagner, M. W. Whalen, and M. P. E. Heimdahl. 2005. "Formal Verification of Flight Critical Software." In *Proc. AIAA Guidance, Navigation and Control Conference*. San Francisco. doi:10.2514/6.2005-6431.

Parnas, D. 2010. "Really Rethinking Formal Methods." *IEEE Computer* 43 (1): 28–34. doi:10.1109/MC.2010.22.

Parnas, D., J. van Schouwen, and P. K. Shu. 1990. "Evaluation of Safety-Critical Software." *Comm. ACM* 33 (6): 636–651. doi:10.1145/78973.78974.

Swartz, A. J. 1996. "Airport 95: Automated Baggage System?" *ACM Software Engineering Notes* 21 (2): 79–83. doi:10.1145/227531.227544.

Trimble, J. 2012. "Agile Development Methods for Space Operations." In *SpaceOps 2012*. Stockholm. doi:10.2514/6.2012-1264554.

Woodcock, J., P. G. Larsen, J. Bicarregui, and J. Fitzgerald. 2009. "Formal Methods: Practice and Experience." *Computing Surveys* 41 (4): 1–36. doi:10.1145/1592434.1592436.

11

Reliability engineering

Objectives

The objective of this chapter is to explain how software reliability may be specified, implemented, and measured. When you have read this chapter, you will:

- understand the distinction between software reliability and software availability;
- have been introduced to metrics for reliability specification and how these are used to specify measurable reliability requirements;
- understand how different architectural styles may be used to implement reliable, fault-tolerant systems architectures;
- know about good programming practice for reliable software engineering;
- understand how the reliability of a software system may be measured using statistical testing.

Contents

11.1 Availability and reliability
11.2 Reliability requirements
11.3 Fault-tolerant architectures
11.4 Programming for reliability
11.5 Reliability measurement

Our dependence on software systems for almost all aspects of our business and personal lives means that we expect that software to be available when we need it. This may be early in the morning or late at night, at weekends or during holidays—the software must run all day, every day of the year. We expect that software will operate without crashes and failures and will preserve our data and personal information. We need to be able to trust the software that we use, which means that the software must be reliable.

The use of software engineering techniques, better programming languages, and effective quality management has led to significant improvements in software reliability over the past 20 years. Nevertheless, system failures still occur that affect the system's availability or lead to incorrect results being produced. In situations where software has a particularly critical role—perhaps in an aircraft or as part of the national critical infrastructure—special reliability engineering techniques may be used to achieve the high levels of reliability and availability that are required.

Unfortunately, it is easy to get confused when talking about system reliability, with different people meaning different things when they talk about system faults and failures. Brian Randell, a pioneer researcher in software reliability, defined a fault–error–failure model (Randell 2000) based on the notion that human errors cause faults; faults lead to errors, and errors lead to system failures. He defined these terms precisely:

1. *Human error or mistake* Human behavior that results in the introduction of faults into a system. For example, in the wilderness weather system, a programmer might decide that the way to compute the time for the next transmission is to add 1 hour to the current time. This works except when the transmission time is between 23.00 and midnight (midnight is 00.00 in the 24-hour clock).

2. *System fault* A characteristic of a software system that can lead to a system error. The fault in the above example is the inclusion of code to add 1 to a variable called Transmission_time, without a check to see if the value of Transmission_time is greater than or equal to 23.00.

3. *System error* An erroneous system state during execution that can lead to system behavior that is unexpected by system users. In this example, the value of the variable Transmission_time is set incorrectly to 24.XX rather than 00.XX when the faulty code is executed.

4. *System failure* An event that occurs at some point in time when the system does not deliver a service as expected by its users. In this case, no weather data is transmitted because the time is invalid.

System faults do not necessarily result in system errors, and system errors do not necessarily result in system failures:

1. Not all code in a program is executed. The code that includes a fault (e.g., the failure to initialize a variable) may never be executed because of the way that the software is used.

2. Errors are transient. A state variable may have an incorrect value caused by the execution of faulty code. However, before this is accessed and causes a system failure, some other system input may be processed that resets the state to a valid value. The wrong value has no practical effect.
3. The system may include fault detection and protection mechanisms. These ensure that the erroneous behavior is discovered and corrected before the system services are affected.

Another reason why the faults in a system may not lead to system failures is that users adapt their behavior to avoid using inputs that they know cause program failures. Experienced users "work around" software features that they have found to be unreliable. For example, I avoid some features, such as automatic numbering, in the word processing system that I use because my experience is that it often goes wrong. Repairing faults in such unused features makes no practical difference to the system reliability.

The distinction between faults, errors, and failures leads to three complementary approaches that are used to improve the reliability of a system:

1. *Fault avoidance* The software design and implementation process should use approaches to software development that help avoid design and programming errors and so minimize the number of faults introduced into the system. Fewer faults means less chance of runtime failures. Fault-avoidance techniques include the use of strongly typed programming language to allow extensive compiler checking and minimizing the use of error-prone programming language constructs, such as pointers.
2. *Fault detection and correction* Verification and validation processes are designed to discover and remove faults in a program, before it is deployed for operational use. Critical systems require extensive verification and validation to discover as many faults as possible before deployment and to convince the system stakeholders and regulators that the system is dependable. Systematic testing and debugging and static analysis are examples of fault-detection techniques.
3. *Fault tolerance* The system is designed so that faults or unexpected system behavior during execution are detected at runtime and are managed in such a way that system failure does not occur. Simple approaches to fault tolerance based on built-in runtime checking may be included in all systems. More specialized fault-tolerance techniques, such as the use of fault-tolerant system architectures, discussed in Section 11.3, may be used when a very high level of system availability and reliability is required.

Unfortunately, applying fault-avoidance, fault-detection, and fault-tolerance techniques is not always cost-effective. The cost of finding and removing the remaining faults in a software system rises exponentially as program faults are discovered and removed (Figure 11.1). As the software becomes more reliable, you need to spend more and more time and effort to find fewer and fewer faults. At some stage, even for critical systems, the costs of this additional effort become unjustifiable.

Figure 11.1 The increasing costs of residual fault removal

As a result, software companies accept that their software will always contain some residual faults. The level of faults depends on the type of system. Software products have a relatively high level of faults, whereas critical systems usually have a much lower fault density.

The rationale for accepting faults is that, if and when the system fails, it is cheaper to pay for the consequences of failure than it would be to discover and remove the faults before system delivery. However, the decision to release faulty software is not simply an economic one. The social and political acceptability of system failure must also be taken into account.

11.1 Availability and reliability

In Chapter 10, I introduced the concepts of system reliability and system availability. If we think of systems as delivering some kind of service (to deliver cash, control brakes, or connect phone calls, for example), then the availability of that service is whether or not that service is up and running and its reliability is whether or not that service delivers correct results. Availability and reliability can both be expressed as probabilities. If the availability is 0.999, this means that, over some time period, the system is available for 99.9% of that time. If, on average, 2 inputs in every 1000 result in failures, then the reliability, expressed as a rate of occurrence of failure, is 0.002.

More precise definitions of availability and reliability are:

1. *Reliability* The probability of failure-free operation over a specified time, in a given environment, for a specific purpose.

2. *Availability* The probability that a system, at a point in time, will be operational and able to deliver the requested services.

Figure 11.2 A system as an input/output mapping

System reliability is not an absolute value—it depends on where and how that system is used. For example, let's say that you measure the reliability of an application in an office environment where most users are uninterested in the operation of the software. They follow the instructions for its use and do not try to experiment with the system. If you then measure the reliability of the same system in a university environment, then the reliability may be quite different. Here, students may explore the boundaries of the system and use it in unexpected ways. This may result in system failures that did not occur in the more constrained office environment. Therefore, the perceptions of the system's reliability in each of these environments are different.

The above definition of reliability is based on the idea of failure-free operation, where failures are external events that affect the users of a system. But what constitutes "failure"? A technical definition of failure is behavior that does not conform to the system's specification. However, there are two problems with this definition:

1. Software specifications are often incomplete or incorrect, and it is left to software engineers to interpret how the system should behave. As they are not domain experts, they may not implement the behavior that users expect. The software may behave as specified, but, for users, it is still failing.

2. No one except system developers reads software specification documents. Users may therefore anticipate that the software should behave in one way when the specification says something completely different.

Failure is therefore not something that can be objectively defined. Rather, it is a judgment made by users of a system. This is one reason why users do not all have the same impression of a system's reliability.

To understand why reliability is different in different environments, we need to think about a system as an input/output mapping. Figure 11.2 shows a software system that

Figure 11.3 Software usage patterns

links a set of inputs with a set of outputs. Given an input or input sequence, the program responds by producing a corresponding output. For example, given an input of a URL, a web browser produces an output that is the display of the requested web page.

Most inputs do not lead to system failure. However, some inputs or input combinations, shown in the shaded ellipse I_e in Figure 11.2, cause system failures or erroneous outputs to be generated. The program's reliability depends on the number of system inputs that are members of the set of inputs that lead to an erroneous output—in other words, the set of inputs that cause faulty code to be executed and system errors to occur. If inputs in the set I_e are executed by frequently used parts of the system, then failures will be frequent. However, if the inputs in I_e are executed by code that is rarely used, then users will hardly ever see failures.

Faults that affect the reliability of the system for one user may never show up under someone else's mode of working. In Figure 11.3, the set of erroneous inputs corresponds to the ellipse labeled I_e in Figure 11.2. The set of inputs produced by User 2 intersects with this erroneous input set. User 2 will therefore experience some system failures. User 1 and User 3, however, never use inputs from the erroneous set. For them, the software will always appear to be reliable.

The availability of a system does not just depend on the number of system failures, but also on the time needed to repair the faults that have caused the failure. Therefore, if system A fails once a year and system B fails once a month, then A is apparently more reliable then B. However, assume that system A takes 6 hours to restart after a failure, whereas system B takes 5 minutes to restart. The availability of system B over the year (60 minutes of down time) is much better than that of system A (360 minutes of downtime).

Furthermore, the disruption caused by unavailable systems is not reflected in the simple availability metric that specifies the percentage of time that the system is available. The time when the system fails is also important. If a system is unavailable for an hour each day between 3 am and 4 am, this may not affect many users. However, if the same system is unavailable for 10 minutes during the working day, system unavailability has a much greater effect on users.

Reliability and availability are closely related, but sometimes one is more important than the other. If users expect continuous service from a system, then the system

has a high-availability requirement. It must be available whenever a demand is made. However, if a system can recover quickly from failures without loss of user data, then these failures may not significantly affect system users.

A telephone exchange switch that routes phone calls is an example of a system where availability is more important than reliability. Users expect to be able to make a call when they pick up a phone or activate a phone app, so the system has high-availability requirements. If a system fault occurs while a connection is being set up, this is often quickly recoverable. Exchange or base station switches can reset the system and retry the connection attempt. This can be done quickly, and phone users may not even notice that a failure has occurred. Furthermore, even if a call is interrupted, the consequences are usually not serious. Users simply reconnect if this happens.

11.2 Reliability requirements

In September 1993, a plane landed at Warsaw Airport in Poland during a thunderstorm. For 9 seconds after landing, the brakes on the computer-controlled braking system did not work. The braking system had not recognized that the plane had landed and assumed that the aircraft was still airborne. A safety feature on the aircraft had stopped the deployment of the reverse thrust system, which slows down the aircraft, because reverse thrust is catastrophic if the plane is in the air. The plane ran off the end of the runway, hit an earth bank, and caught fire.

The inquiry into the accident showed that the braking system software had operated according to its specification. There were no errors in the control system. However, the software specification was incomplete and had not taken into account a rare situation, which arose in this case. The software worked, but the system failed.

This incident shows that system dependability does not just depend on good engineering. It also requires attention to detail when the system requirements are derived and the specification of software requirements that are geared to ensuring the dependability of a system. Those dependability requirements are of two types:

1. *Functional requirements*, which define checking and recovery facilities that should be included in the system and features that provide protection against system failures and external attacks.
2. *Non-functional requirements*, which define the required reliability and availability of the system.

As I discussed in Chapter 10, the overall reliability of a system depends on the hardware reliability, the software reliability, and the reliability of the system operators. The system software has to take this requirement into account. As well as including requirements that compensate for software failure, there may also be related reliability requirements to help detect and recover from hardware failures and operator errors.

Availability	Explanation
0.9	The system is available for 90% of the time. This means that, in a 24-hour period (1440 minutes), the system will be unavailable for 144 minutes.
0.99	In a 24-hour period, the system is unavailable for 14.4 minutes.
0.999	The system is unavailable for 84 seconds in a 24-hour period.
0.9999	The system is unavailable for 8.4 seconds in a 24-hour period—roughly, one minute per week.

Figure 11.4 Availability specification

11.2.1 Reliability metrics

Reliability can be specified as a probability that a system failure will occur when a system is in use within a specified operating environment. If you are willing to accept, for example, that 1 in any 1000 transactions may fail, then you can specify the failure probability as 0.001. This doesn't mean that there will be exactly 1 failure in every 1000 transactions. It means that if you observe *N* thousand transactions, the number of failures that you observe should be about *N*.

Three metrics may be used to specify reliability and availability:

1. *Probability of failure on demand* (*POFOD*) If you use this metric, you define the probability that a demand for service from a system will result in a system failure. So, POFOD = 0.001 means that there is a 1/1000 chance that a failure will occur when a demand is made.

2. *Rate of occurrence of failures* (*ROCOF*) This metric sets out the probable number of system failures that are likely to be observed relative to a certain time period (e.g., an hour), or to the number of system executions. In the example above, the ROCOF is 1/1000. The reciprocal of ROCOF is the *mean time to failure (MTTF)*, which is sometimes used as a reliability metric. MTTF is the average number of time units between observed system failures. A ROCOF of two failures per hour implies that the mean time to failure is 30 minutes.

3. *Availability (AVAIL)* AVAIL is the probability that a system will be operational when a demand is made for service. Therefore, an availability of 0.9999 means that, on average, the system will be available for 99.99% of the operating time. Figure 11.4 shows what different levels of availability mean in practice.

POFOD should be used in situations where a failure on demand can lead to a serious system failure. This applies irrespective of the frequency of the demands. For example, a protection system that monitors a chemical reactor and shuts down the reaction if it is overheating should have its reliability specified using POFOD. Generally, demands on a protection system are infrequent as the system is a last line of defense, after all other recovery strategies have failed. Therefore a POFOD of 0.001 (1 failure in 1000 demands)

might seem to be risky. However, if there are only two or three demands on the system in its entire lifetime, then the system is unlikely to ever fail.

ROCOF should be used when demands on systems are made regularly rather than intermittently. For example, in a system that handles a large number of transactions, you may specify a ROCOF of 10 failures per day. This means that you are willing to accept that an average of 10 transactions per day will not complete successfully and will have to be canceled and resubmitted. Alternatively, you may specify ROCOF as the number of failures per 1000 transactions.

If the absolute time between failures is important, you may specify the reliability as the mean time to failures (MTTF). For example, if you are specifying the required reliability for a system with long transactions (such as a computer-aided design system), you should use this metric. The MTTF should be much longer than the average time that a user works on his or her models without saving the user's results. This means that users are unlikely to lose work through a system failure in any one session.

11.2.2 Non-functional reliability requirements

Non-functional reliability requirements are specifications of the required reliability and availability of a system using one of the reliability metrics (POFOD, ROCOF, or AVAIL) described in the previous section. Quantitative reliability and availability specification has been used for many years in safety-critical systems but is uncommon for business critical systems. However, as more and more companies demand 24/7 service from their systems, it makes sense for them to be precise about their reliability and availability expectations.

Quantitative reliability specification is useful in a number of ways:

1. The process of deciding the required level of the reliability helps to clarify what stakeholders really need. It helps stakeholders understand that there are different types of system failure, and it makes clear to them that high levels of reliability are expensive to achieve.
2. It provides a basis for assessing when to stop testing a system. You stop when the system has reached its required reliability level.
3. It is a means of assessing different design strategies intended to improve the reliability of a system. You can make a judgment about how each strategy might lead to the required levels of reliability.
4. If a regulator has to approve a system before it goes into service (e.g., all systems that are critical to flight safety on an aircraft are regulated), then evidence that a required reliability target has been met is important for system certification.

To avoid incurring excessive and unnecessary costs, it is important that you specify the reliability that you really need rather than simply choose a very high level of reliability for the whole system. You may have different requirements for different

Overspecification of reliability

Overspecification of reliability means defining a level of required reliability that is higher than really necessary for the practical operation of the software. Overspecification of reliability increases development costs disproportionately. The reason for this is that the costs of reducing faults and verifying reliability increase exponentially as reliability increases

http://software-engineering-book.com/web/over-specifying-reliability/

parts of the system if some parts are more critical than others. You should follow these three guidelines when specifying reliability requirements:

1. Specify the availability and reliability requirements for different types of failure. There should be a lower probability of high-cost failures than failures that don't have serious consequences.
2. Specify the availability and reliability requirements for different types of system service. Critical system services should have the highest reliability but you may be willing to tolerate more failures in less critical services. You may decide that it is only cost-effective to use quantitative reliability specification for the most critical system services.
3. Think about whether high reliability is really required. For example, you may use error-detection mechanisms to check the outputs of a system and have error-correction processes in place to correct errors. There may then be no need for a high level of reliability in the system that generates the outputs as errors can be detected and corrected.

To illustrate these guidelines, think about the reliability and availability requirements for a bank ATM system that dispenses cash and provides other services to customers. Banks have two concerns with such systems:

1. To ensure that they carry out customer services as requested and that they properly record customer transactions in the account database.
2. To ensure that these systems are available for use when required.

Banks have many years of experience with identifying and correcting incorrect account transactions. They use accounting methods to detect when things have gone wrong. Most transactions that fail can simply be canceled, resulting in no loss to the bank and minor customer inconvenience. Banks that run ATM networks therefore accept that ATM failures may mean that a small number of transactions are incorrect, but they think it more cost-effective to fix these errors later rather than incur high costs in avoiding faulty transactions. Therefore, the absolute reliability required of an ATM may be relatively low. Several failures per day may be acceptable.

For a bank (and for the bank's customers), the availability of the ATM network is more important than whether or not individual ATM transactions fail. Lack of availability means increased demand on counter services, customer dissatisfaction, engineering costs to repair the network, and so on. Therefore, for transaction-based systems such as banking and e-commerce systems, the focus of reliability specification is usually on specifying the availability of the system.

To specify the availability of an ATM network, you should identify the system services and specify the required availability for each of these services, notably:

- the customer account database service; and
- the individual services provided by an ATM such as "withdraw cash" and "provide account information."

The database service is the most critical as failure of this service means that all of the ATMs in the network are out of action. Therefore, you should specify this service to have a high level of availability. In this case, an acceptable figure for database availability (ignoring issues such as scheduled maintenance and upgrades) would probably be around 0.9999, between 7 am and 11 pm. This means a downtime of less than 1 minute per week.

For an individual ATM, the overall availability depends on mechanical reliability and the fact that it can run out of cash. Software issues are probably less significant than these factors. Therefore, a lower level of software availability for the ATM software is acceptable. The overall availability of the ATM software might therefore be specified as 0.999, which means that a machine might be unavailable for between 1 and 2 minutes each day. This allows for the ATM software to be restarted in the event of a problem.

The reliability of control systems is usually specified in terms of the probability that the system will fail when a demand is made (POFOD). Consider the reliability requirements for the control software in the insulin pump, introduced in Chapter 1. This system delivers insulin a number of times per day and monitors the user's blood glucose several times per hour.

There are two possible types of failure in the insulin pump:

1. *Transient software failures*, which can be repaired by user actions such as resetting or recalibrating the machine. For these types of failure, a relatively low value of POFOD (say 0.002) may be acceptable. This means that one failure may occur in every 500 demands made on the machine. This is approximately once every 3.5 days, because the blood sugar is checked about 5 times per hour.
2. *Permanent software failures*, which require the software to be reinstalled by the manufacturer. The probability of this type of failure should be much lower. Roughly once a year is the minimum figure, so POFOD should be no more than 0.00002.

RR1: A predefined range for all operator inputs shall be defined, and the system shall check that all operator inputs fall within this predefined range. (Checking)
RR2: Copies of the patient database shall be maintained on two separate servers that are not housed in the same building. (Recovery, redundancy)
RR3: *N*-version programming shall be used to implement the braking control system. (Redundancy)
RR4: The system must be implemented in a safe subset of Ada and checked using static analysis. (Process)

Figure 11.5 Examples of functional reliability requirements

Failure to deliver insulin does not have immediate safety implications, so commercial factors rather than safety factors govern the level of reliability required. Service costs are high because users need fast repair and replacement. It is in the manufacturer's interest to limit the number of permanent failures that require repair.

11.2.3 Functional reliability specification

To achieve a high level of reliability and availability in a software-intensive system, you use a combination of fault-avoidance, fault-detection, and fault-tolerance techniques. This means that functional reliability requirements have to be generated which specify how the system should provide fault avoidance, detection, and tolerance.

These functional reliability requirements should specify the faults to be detected and the actions to be taken to ensure that these faults do not lead to system failures. Functional reliability specification, therefore, involves analyzing the non-functional requirements (if these have been specified), assessing the risks to reliability and specifying system functionality to address these risks.

There are four types of functional reliability requirements:

1. *Checking requirements* These requirements identify checks on inputs to the system to ensure that incorrect or out-of-range inputs are detected before they are processed by the system.
2. *Recovery requirements* These requirements are geared to helping the system recover after a failure has occurred. These requirements are usually concerned with maintaining copies of the system and its data and specifying how to restore system services after failure.
3. *Redundancy requirements* These specify redundant features of the system that ensure that a single component failure does not lead to a complete loss of service. I discuss this in more detail in the next chapter.
4. *Process requirements* These are fault-avoidance requirements, which ensure that good practice is used in the development process. The practices specified should reduce the number of faults in a system.

Some examples of these types of reliability requirement are shown in Figure 11.5.

There are no simple rules for deriving functional reliability requirements. Organizations that develop critical systems usually have organizational knowledge about possible reliability requirements and how these requirements reflect the actual reliability of a system. These organizations may specialize in specific types of systems, such as railway control systems, so the reliability requirements can be reused across a range of systems.

11.3 Fault-tolerant architectures

Fault tolerance is a runtime approach to dependability in which systems include mechanisms to continue in operation, even after a software or hardware fault has occurred and the system state is erroneous. Fault-tolerance mechanisms detect and correct this erroneous state so that the occurrence of a fault does not lead to a system failure. Fault tolerance is required in systems that are safety or security critical and where the system cannot move to a safe state when an error is detected.

To provide fault tolerance, the system architecture has to be designed to include redundant and diverse hardware and software. Examples of systems that may need fault-tolerant architectures are aircraft systems that must be available throughout the duration of the flight, telecommunication systems, and critical command and control systems.

The simplest realization of a dependable architecture is in replicated servers, where two or more servers carry out the same task. Requests for processing are channeled through a server management component that routes each request to a particular server. This component also keeps track of server responses. In the event of server failure, which can be detected by a lack of response, the faulty server is switched out of the system. Unprocessed requests are resubmitted to other servers for processing.

This replicated server approach is widely used for transaction processing systems where it is easy to maintain copies of transactions to be processed. Transaction processing systems are designed so that data is only updated once a transaction has finished correctly. Delays in processing do not affect the integrity of the system. It can be an efficient way of using hardware if the backup server is one that is normally used for low-priority tasks. If a problem occurs with a primary server, its unprocessed transactions are transferred to the backup server, which gives that work the highest priority.

Replicated servers provide redundancy but not usually diversity. The server hardware is usually identical, and the servers run the same version of the software. Therefore, they can cope with hardware failures and software failures that are localized to a single machine. They cannot cope with software design problems that cause all versions of the software to fail at the same time. To handle software design failures, a system has to use diverse software and hardware.

Torres-Pomales surveys a range of software fault-tolerance techniques (Torres-Pomales 2000), and Pullum (Pullum 2001) describes different types of fault-tolerant architecture. In the following sections, I describe three architectural patterns that have been used in fault-tolerant systems.

Figure 11.6 Protection system architecture

11.3.1 Protection systems

A protection system is a specialized system that is associated with some other system. This is usually a control system for some process, such as a chemical manufacturing process, or an equipment control system, such as the system on a driverless train. An example of a protection system might be a system on a train that detects if the train has gone through a red signal. If there is no indication that the train control system is slowing down the train, then the protection system automatically applies the train brakes to bring it to a halt. Protection systems independently monitor their environment. If sensors indicate a problem that the controlled system is not dealing with, then the protection system is activated to shut down the process or equipment.

Figure 11.6 illustrates the relationship between a protection system and a controlled system. The protection system monitors both the controlled equipment and the environment. If a problem is detected, it issues commands to the actuators to shut down the system or invoke other protection mechanisms such as opening a pressure-release valve. Notice that there are two sets of sensors. One set is used for normal system monitoring and the other specifically for the protection system. In the event of sensor failure, backups are in place that will allow the protection system to continue in operation. The system may also have redundant actuators.

A protection system only includes the critical functionality that is required to move the system from a potentially unsafe state to a safe state (which could be system shutdown). It is an instance of a more general fault-tolerant architecture in which a principal system is supported by a smaller and simpler backup system that only includes essential functionality. For example, the control software for the U.S. Space Shuttle had a backup system with “get you home” functionality. That is, the backup system could land the vehicle if the principal control system failed but had no other control functions.

Figure 11.7 Self-monitoring architecture

The advantage of this architectural style is that protection system software can be much simpler than the software that is controlling the protected process. The only function of the protection system is to monitor operation and to ensure that the system is brought to a safe state in the event of an emergency. Therefore, it is possible to invest more effort in fault avoidance and fault detection. You can check that the software specification is correct and consistent and that the software is correct with respect to its specification. The aim is to ensure that the reliability of the protection system is such that it has a very low probability of failure on demand (say, 0.001). Given that demands on the protection system should be rare, a probability of failure on demand of 1/1000 means that protection system failures should be very rare.

11.3.2 Self-monitoring architectures

A self-monitoring architecture (Figure 11.7) is a system architecture in which the system is designed to monitor its own operation and to take some action if a problem is detected. Computations are carried out on separate channels, and the outputs of these computations are compared. If the outputs are identical and are available at the same time, then the system is judged to be operating correctly. If the outputs are different, then a failure is assumed. When this occurs, the system raises a failure exception on the status output line. This signals that control should be transferred to some other system.

To be effective in detecting both hardware and software faults, self-monitoring systems have to be designed so that:

1. The hardware used in each channel is diverse. In practice, this might mean that each channel uses a different processor type to carry out the required computations, or the chipset making up the system may be sourced from different manufacturers. This reduces the probability of common processor design faults affecting the computation.

2. The software used in each channel is diverse. Otherwise, the same software error could arise at the same time on each channel.

On its own, this architecture may be used in situations where it is important for computations to be correct, but where availability is not essential. If the answers

Figure 11.8 The Airbus flight control system architecture

from each channel differ, the system shuts down. For many medical treatment and diagnostic systems, reliability is more important than availability because an incorrect system response could lead to the patient receiving incorrect treatment. However, if the system shuts down in the event of an error, this is an inconvenience but the patient will not usually be harmed.

In situations that require high availability, you have to use several self-checking systems in parallel. You need a switching unit that detects faults and selects a result from one of the systems, where both channels are producing a consistent response. This approach is used in the flight control system for the Airbus 340 series of aircraft, which uses five self-checking computers. Figure 11.8 is a simplified diagram of the Airbus flight control system that shows the organization of the self-monitoring systems.

In the Airbus flight control system, each of the flight control computers carries out the computations in parallel, using the same inputs. The outputs are connected to hardware filters that detect if the status indicates a fault and, if so, that the output from that computer is switched off. The output is then taken from an alternative system. Therefore, it is possible for four computers to fail and for the aircraft operation to continue. In more than 15 years of operation, there have been no reports of situations where control of the aircraft has been lost due to total flight control system failure.

Figure 11.9 Triple modular redundancy

The designers of the Airbus system have tried to achieve diversity in a number of different ways:

1. The primary flight control computers use a different processor from the secondary flight control systems.
2. The chipset that is used in each channel in the primary and secondary systems is supplied by a different manufacturer.
3. The software in the secondary flight control systems provides critical functionality only—it is less complex than the primary software.
4. The software for each channel in both the primary and the secondary systems is developed using different programming languages and by different teams.
5. Different programming languages are used in the secondary and primary systems.

As I discuss in Section 11.3.4, these do not guarantee diversity but they reduce the probability of common failures in different channels.

11.3.3 *N*-version programming

Self-monitoring architectures are examples of systems in which multiversion programming is used to provide software redundancy and diversity. This notion of multiversion programming has been derived from hardware systems where the notion of triple modular redundancy (TMR) has been used for many years to build systems that are tolerant of hardware failures (Figure 11.9).

In a TMR system, the hardware unit is replicated three (or sometimes more) times. The output from each unit is passed to an output comparator that is usually implemented as a voting system. This system compares all of its inputs, and, if two or more are the same, then that value is output. If one of the units fails and does not produce the same output as the other units, its output is ignored. A fault manager may try to repair the faulty unit automatically, but if this is impossible, the system is automatically reconfigured to take the unit out of service. The system then continues to function with two working units.

This approach to fault tolerance relies on most hardware failures being the result of component failure rather than design faults. The components are therefore likely

Figure 11.10 *N*-version programming

to fail independently. It assumes that, when fully operational, all hardware units perform to specification. There is therefore a low probability of simultaneous component failure in all hardware units.

Of course, the components could all have a common design fault and thus all produce the same (wrong) answer. Using hardware units that have a common specification but that are designed and built by different manufacturers reduces the chances of such a common mode failure. It is assumed that the probability of different teams making the same design or manufacturing error is small.

A similar approach can be used for fault-tolerant software where *N* diverse versions of a software system execute in parallel (Avizienis 1995). This approach to software fault tolerance, illustrated in Figure 11.10, has been used in railway signaling systems, aircraft systems, and reactor protection systems.

Using a common specification, the same software system is implemented by a number of teams. These versions are executed on separate computers. Their outputs are compared using a voting system, and inconsistent outputs or outputs that are not produced in time are rejected. At least three versions of the system should be available so that two versions should be consistent in the event of a single failure.

N-version programming may be less expensive than self-checking architectures in systems for which a high level of availability is required. However, it still requires several different teams to develop different versions of the software. This leads to very high software development costs. As a result, this approach is only used in systems where it is impractical to provide a protection system that can guard against safety-critical failures.

11.3.4 Software diversity

All of the above fault-tolerant architectures rely on software diversity to achieve fault tolerance. This is based on the assumption that diverse implementations of the same specification (or a part of the specification, for protection systems) are independent. They should not include common errors and so will not fail in the same way, at the same time. The software should therefore be written by different teams who should not communicate during the development process. This requirement reduces the chances of common misunderstandings or misinterpretations of the specification.

The company that is procuring the system may include explicit diversity policies that are intended to maximize the differences between the system versions. For example:

1. By including requirements that different design methods should be used. For example, one team may be required to produce an object-oriented design, and another team may produce a function-oriented design.
2. By stipulating that the programs should be implemented using different programming languages. For example, in a three-version system, Ada, C++, and Java could be used to write the software versions.
3. By requiring the use of different tools and development environments for the system.
4. By requiring different algorithms to be used in some parts of the implementation. However, this limits the freedom of the design team and may be difficult to reconcile with system performance requirements.

Ideally, the diverse versions of the system should have no dependencies and so should fail in completely different ways. If this is the case, then the overall reliability of a diverse system is obtained by multiplying the reliabilities of each channel. So, if each channel has a probability of failure on demand of 0.001, then the overall POFOD of a three-channel system (with all channels independent) is a million times greater than the reliability of a single channel system.

In practice, however, achieving complete channel independence is impossible. It has been shown experimentally that independent software design teams often make the same mistakes or misunderstand the same parts of the specification (Brilliant, Knight, and Leveson 1990; Leveson 1995). There are several reasons for this misunderstanding:

1. Members of different teams are often from the same cultural background and may have been educated using the same approach and textbooks. This means that they may find the same things difficult to understand and have common difficulties in communicating with domain experts. It is quite possible that they will, independently, make the same mistakes and design the same algorithms to solve a problem.
2. If the requirements are incorrect or they are based on misunderstandings about the environment of the system, then these mistakes will be reflected in each implementation of the system.
3. In a critical system, the detailed system specification that is derived from the system's requirements should provide an unambiguous definition of the system's behavior. However, if the specification is ambiguous, then different teams may misinterpret the specification in the same way.

One way to reduce the possibility of common specification errors is to develop detailed specifications for the system independently and to define the specifications in different languages. One development team might work from a formal specification,

another from a state-based system model, and a third from a natural language specification. This approach helps avoid some errors of specification interpretation, but does not get around the problem of requirements errors. It also introduces the possibility of errors in the translation of the requirements, leading to inconsistent specifications.

In an analysis of the experiments, Hatton (Hatton 1997) concluded that a three-channel system was somewhere between 5 and 9 times more reliable than a single-channel system. He concluded that improvements in reliability that could be obtained by devoting more resources to a single version could not match this and so *N*-version approaches were more likely to lead to more reliable systems than single-version approaches.

What is unclear, however, is whether the improvements in reliability from a multiversion system are worth the extra development costs. For many systems, the extra costs may not be justifiable, as a well-engineered single-version system may be good enough. It is only in safety- and mission-critical systems, where the costs of failure are very high, that multiversion software may be required. Even in such situations (e.g., a spacecraft system), it may be enough to provide a simple backup with limited functionality until the principal system can be repaired and restarted.

11.4 Programming for reliability

I have deliberately focused in this book on programming-language independent aspects of software engineering. It is almost impossible to discuss programming without getting into the details of a specific programming language. However, when considering reliability engineering, there are a set of accepted good programming practices that are fairly universal and that help reduce faults in delivered systems.

A list of eight good practice guidelines is shown in Figure 11.11. They can be applied regardless of the particular programming language used for systems development, although the way they are used depends on the specific languages and notations that are used for system development. Following these guidelines also reduces the chances of introducing security-related vulnerabilities into programs.

Guideline 1: Control the visibility of information in a program

A security principle that is adopted by military organizations is the "need to know" principle. Only those individuals who need to know a particular piece of information in order to carry out their duties are given that information. Information that is not directly relevant to their work is withheld.

When programming, you should adopt an analogous principle to control access to the variables and data structures that you use. Program components should only be allowed access to data that they need for their implementation. Other program data should be inaccessible and hidden from them. If you hide information, it cannot be corrupted by program components that are not supposed to use it. If the interface remains the same, the data representation may be changed without affecting other components in the system.

Dependable programming guidelines
1. Limit the visibility of information in a program.
2. Check all inputs for validity.
3. Provide a handler for all exceptions.
4. Minimize the use of error-prone constructs.
5. Provide restart capabilities.
6. Check array bounds.
7. Include timeouts when calling external components.
8. Name all constants that represent real-world values.

Figure 11.11 Good practice guidelines for dependable programming

You can achieve this by implementing data structures in your program as abstract data types. An abstract data type is one in which the internal structure and representation of a variable of that type are hidden. The structure and attributes of the type are not externally visible, and all access to the data is through operations.

For example, you might have an abstract data type that represents a queue of requests for service. Operations should include get and put, which add and remove items from the queue, and an operation that returns the number of items in the queue. You might initially implement the queue as an array but subsequently decide to change the implementation to a linked list. This can be achieved without any changes to code using the queue, because the queue representation is never directly accessed.

In some object-oriented languages, you can implement abstract data types using interface definitions, where you declare the interface to an object without reference to its implementation. For example, you can define an interface Queue, which supports methods to place objects onto the queue, remove them from the queue, and query the size of the queue. In the object class that implements this interface, the attributes and methods should be private to that class.

Guideline 2: Check all inputs for validity

All programs take inputs from their environment and process them. The specification makes assumptions about these inputs that reflect their real-world use. For example, it may be assumed that a bank account number is always an eight-digit positive integer. In many cases, however, the system specification does not define what actions should be taken if the input is incorrect. Inevitably, users will make mistakes and will sometimes enter the wrong data. As I discuss in Chapter 13, malicious attacks on a system may rely on deliberately entering invalid information. Even when inputs come from sensors or other systems, these systems can go wrong and provide incorrect values.

You should therefore always check the validity of inputs as soon as they are read from the program's operating environment. The checks involved obviously depend on the inputs themselves, but possible checks that may be used are:

1. *Range checks* You may expect inputs to be within a particular range. For example, an input that represents a probability should be within the range 0.0 to 1.0; an input that represents the temperature of a liquid water should be between 0 degrees Celsius and 100 degrees Celsius, and so on.

2. *Size checks* You may expect inputs to be a given number of characters, for example, 8 characters to represent a bank account. In other cases, the size may not be fixed, but there may be a realistic upper limit. For example, it is unlikely that a person's name will have more than 40 characters.

3. *Representation checks* You may expect an input to be of a particular type, which is represented in a standard way. For example, people's names do not include numeric characters, email addresses are made up of two parts, separated by a @ sign, and so on.

4. *Reasonableness checks* Where an input is one of a series and you know something about the relationships between the members of the series, then you can check that an input value is reasonable. For example, if the input value represents the readings of a household electricity meter, then you would expect the amount of electricity used to be approximately the same as in the corresponding period in the previous year. Of course, there will be variations. but order of magnitude differences suggest that something has gone wrong.

The actions that you take if an input validation check fails depend on the type of system being implemented. In some cases, you report the problem to the user and request that the value is re-input. Where a value comes from a sensor, you might use the most recent valid value. In embedded real-time systems, you might have to estimate the value based on previous data, so that the system can continue in operation.

Guideline 3: Provide a handler for all exceptions

During program execution, errors or unexpected events inevitably occur. These may arise because of a program fault, or they may be a result of unpredictable external circumstances. An error or an unexpected event that occurs during the execution of a program is called an exception. Examples of exceptions might be a system power failure, an attempt to access nonexistent data, or numeric overflow or underflow.

Exceptions may be caused by hardware or software conditions. When an exception occurs, it must be managed by the system. This can be done within the program itself, or it may involve transferring control to a system exception-handling mechanism. Typically, the system's exception management mechanism reports the error and shuts down execution. Therefore, to ensure that program exceptions do not cause system failure, you should define an exception handler for all possible exceptions that may arise; you should also make sure that all exceptions are detected and explicitly handled.

Languages such as Java, C++, and Python have built-in exception-handling constructs. When an exceptional situation occurs, the exception is signaled and the language runtime system transfers control to an exception handler. This is a code section that states exception names and appropriate actions to handle each exception (Figure 11.12). The exception handler is outside the normal flow of control, and this normal control flow does not resume after the exception has been handled.

Figure 11.12 Exception handling

An exception handler usually does one of three things:

1. Signals to a higher-level component that an exception has occurred and provides information to that component about the type of exception. You use this approach when one component calls another and the calling component needs to know if the called component has executed successfully. If not, it is up to the calling component to take action to recover from the problem.
2. Carries out some alternative processing to that which was originally intended. Therefore, the exception handler takes some actions to recover from the problem. Processing may then continue as normal. Alternatively, the exception handler may indicate that an exception has occurred so that a calling component is aware of and can deal with the exception.
3. Passes control to the programming language runtime support system that handles the exception. This is often the default when faults occur in a program, for example, when a numeric value overflows. The usual action of the runtime system is to halt processing. You should only use this approach when it is possible to move the system to a safe and quiescent state, before handing over control to the runtime system.

Handling exceptions within a program makes it possible to detect and recover from some input errors and unexpected external events. As such, it provides a degree of fault tolerance. The program detects faults and can take action to recover from them. As most input errors and unexpected external events are usually transient, it is often possible to continue normal operation after the exception has been processed.

Guideline 4: Minimize the use of error-prone constructs

Faults in programs, and therefore many program failures, are usually a consequence of human error. Programmers make mistakes because they lose track of the numerous relationships between the state variables. They write program statements that result in unexpected behavior and system state changes. People will always make

Error-prone constructs

Some programming language features are more likely than others to lead to the introduction of program bugs. Program reliability is likely to be improved if you avoid using these constructs. Wherever possible, you should minimize the use of go to statements, floating-point numbers, pointers, dynamic memory allocation, parallelism, recursion, interrupts, aliasing, unbounded arrays, and default input processing.

http://software-engineering-book.com/web/error-prone-constructs/

mistakes, but in the late 1960s it became clear that some approaches to programming were more likely to introduce errors into a program than others.

For example, you should try to avoid using floating-point numbers because the precision of floating point numbers is limited by their hardware representation. Comparisons of very large or very small numbers are unreliable. Another construct that is potentially error-prone is dynamic storage allocation where you explicitly manage storage in the program. It is very easy to forget to release storage when it's no longer needed, and this can lead to hard to detect runtime errors.

Some standards for safety-critical systems development completely prohibit the use of error-prone constructs. However, such an extreme position is not normally practical. All of these constructs and techniques are useful, though they must be used with care. Wherever possible, their potentially dangerous effects should be controlled by using them within abstract data types or objects. These act as natural "firewalls" limiting the damage caused if errors occur.

Guideline 5: Provide restart capabilities

Many organizational information systems are based on short transactions where processing user inputs takes a relatively short time. These systems are designed so that changes to the system's database are only finalized after all other processing has been successfully completed. If something goes wrong during processing, the database is not updated and so does not become inconsistent. Virtually all e-commerce systems, where you only commit to your purchase on the final screen, work in this way.

User interactions with e-commerce systems usually last a few minutes and involve minimal processing. Database transactions are short and are usually completed in less than a second. However, other types of system such as CAD systems and word processing systems involve long transactions. In a long transaction system, the time between starting to use the system and finishing work may be several minutes or hours. If the system fails during a long transaction, then all of the work may be lost. Similarly, in computationally intensive systems such as some e-science systems, minutes or hours of processing may be required to complete the computation. All of this time is lost in the event of a system failure.

In all of these types of systems, you should provide a restart capability that is based on keeping copies of data collected or generated during processing. The restart facility should allow the system to restart using these copies, rather than having to

start all over from the beginning. These copies are sometimes called checkpoints. For example:

1. In an e-commerce system, you can keep copies of forms filled in by a user and allow them to access and submit these forms without having to fill them in again.
2. In a long transaction or computationally intensive system, you can automatically save data every few minutes and, in the event of a system failure, restart with the most recently saved data. You should also allow for user error and provide a way for users to go back to the most recent checkpoint and start again from there.

If an exception occurs and it is impossible to continue normal operation, you can handle the exception using backward error recovery. This means that you reset the state of the system to the saved state in the checkpoint and restart operation from that point.

Guideline 6: Check array bounds

All programming languages allow the specification of arrays—sequential data structures that are accessed using a numeric index. These arrays are usually laid out in contiguous areas within the working memory of a program. Arrays are specified to be of a particular size, which reflects how they are used. For example, if you wish to represent the ages of up to 10,000 people, then you might declare an array with 10,000 locations to hold the age data.

Some programming languages, such as Java, always check that when a value is entered into an array, the index is within that array. So, if an array A is indexed from 0 to 10,000, an attempt to enter values into elements A [-5] or A [12345] will lead to an exception being raised. However, programming languages such as C and C++ do not automatically include array bound checks and simply calculate an offset from the beginning of the array. Therefore, A [12345] would access the word that was 12345 locations from the beginning of the array, irrespective of whether or not this was part of the array.

These languages do not include automatic array bound checking because this introduces an overhead every time the array is accessed and so it increases program execution time. However, the lack of bound checking leads to security vulnerabilities, such as buffer overflow, which I discuss in Chapter 13. More generally, it introduces a system vulnerability that can lead to system failure. If you are using a language such as C or C++ that does not include array bound checking, you should always include checks that the array index is within bounds.

Guideline 7: Include timeouts when calling external components

In distributed systems, components of the system execute on different computers, and calls are made across the network from component to component. To receive some service, component A may call component B. A waits for B to respond before continuing execution. However, if component B fails to respond for some reason, then component A cannot continue. It simply waits indefinitely for a response. A person

who is waiting for a response from the system sees a silent system failure, with no response from the system. They have no alternative but to kill the waiting process and restart the system.

To avoid this prospect, you should always include timeouts when calling external components. A timeout is an automatic assumption that a called component has failed and will not produce a response. You define a time period during which you expect to receive a response from a called component. If you have not received a response in that time, you assume failure and take back control from the called component. You can then attempt to recover from the failure or tell the system users what has happened and allow them to decide what to do.

Guideline 8: Name all constants that represent real-world values

All nontrivial programs include a number of constant values that represent the values of real-world entities. These values are not modified as the program executes. Sometimes, these are absolute constants and never change (e.g., the speed of light), but more often they are values that change relatively slowly over time. For example, a program to calculate personal tax will include constants that are the current tax rates. These change from year to year, and so the program must be updated with the new constant values.

You should always include a section in your program in which you name all real-world constant values that are used. When using the constants, you should refer to them by name rather than by their value. This has two advantages as far as dependability is concerned:

1. You are less likely to make mistakes and use the wrong value. It is easy to mistype a number, and the system will often be unable to detect a mistake. For example, say a tax rate is 34%. A simple transposition error might lead to this being mistyped as 43%. However, if you mistype a name (such as Standard-tax-rate), this error can be detected by the compiler as an undeclared variable.
2. When a value changes, you do not have to look through the whole program to discover where you have used that value. All you need do is to change the value associated with the constant declaration. The new value is then automatically included everywhere that it is needed.

11.5 Reliability measurement

To assess the reliability of a system, you have to collect data about its operation. The data required may include:

1. The number of system failures given a number of requests for system services. This is used to measure the POFOD and applies irrespective of the time over which the demands are made.

Figure 11.13 Statistical testing for reliability measurement

2. The time or the number of transactions between system failures plus the total elapsed time or total number of transactions. This is used to measure ROCOF and MTTF.

3. The repair or restart time after a system failure that leads to loss of service. This is used in the measurement of availability. Availability does not just depend on the time between failures but also on the time required to get the system back into operation.

The time units that may be used in these metrics are calendar time or a discrete unit such as number of transactions. You should use calendar time for systems that are in continuous operation. Monitoring systems, such as process control systems, fall into this category. Therefore, the ROCOF might be the number of failures per day. Systems that process transactions such as bank ATMs or airline reservation systems have variable loads placed on them depending on the time of day. In these cases, the unit of "time" used could be the number of transactions; that is, the ROCOF would be number of failed transactions per *N* thousand transactions.

Reliability testing is a statistical testing process that aims to measure the reliability of a system. Reliability metrics such as POFOD, the probability of failure on demand, and ROCOF, the rate of occurrence of failure, may be used to quantitatively specify the required software reliability. You can check on the reliability testing process if the system has achieved that required reliability level.

The process of measuring the reliability of a system is sometimes called statistical testing (Figure 11.13). The statistical testing process is explicitly geared to reliability measurement rather than fault finding. Prowell et al. (Prowell et al. 1999) give a good description of statistical testing in their book on Cleanroom software engineering.

There are four stages in the statistical testing process:

1. You start by studying existing systems of the same type to understand how these are used in practice. This is important as you are trying to measure the reliability as experienced by system users. Your aim is to define an operational profile. An operational profile identifies classes of system inputs and the probability that these inputs will occur in normal use.

2. You then construct a set of test data that reflect the operational profile. This means that you create test data with the same probability distribution as the test data for the systems that you have studied. Normally, you use a test data generator to support this process.

3. You test the system using these data and count the number and type of failures that occur. The times of these failures are also logged. As I discussed in Chapter 10, the time units chosen should be appropriate for the reliability metric used.

4. After you have observed a statistically significant number of failures, you can compute the software reliability and work out the appropriate reliability metric value.

This conceptually attractive approach to reliability measurement is not easy to apply in practice. The principal difficulties that arise are due to:

1. *Operational profile uncertainty* The operational profiles based on experience with other systems may not be an accurate reflection of the real use of the system.
2. *High costs of test data generation* It can be very expensive to generate the large volume of data required in an operational profile unless the process can be totally automated.
3. *Statistical uncertainty when high reliability is specified* You have to generate a statistically significant number of failures to allow accurate reliability measurements. When the software is already reliable, relatively few failures occur and it is difficult to generate new failures.
4. *Recognizing failure* It is not always obvious whether or not a system failure has occurred. If you have a formal specification, you may be able to identify deviations from that specification, but, if the specification is in natural language, there may be ambiguities that mean observers could disagree on whether the system has failed.

By far the best way to generate the large dataset required for reliability measurement is to use a test data generator, which can be set up to automatically generate inputs matching the operational profile. However, it is not usually possible to automate the production of all test data for interactive systems because the inputs are often a response to system outputs. Datasets for these systems have to be generated manually, with correspondingly higher costs. Even where complete automation is possible, writing commands for the test data generator may take a significant amount of time.

Statistical testing may be used in conjunction with fault injection to gather data about how effective the process of defect testing has been. Fault injection (Voas and McGraw 1997) is the deliberate injection of errors into a program. When the program is executed, these lead to program faults and associated failures. You then analyze the failure to discover if the root cause is one of the errors that you have added to the program. If you find that X% of the injected faults lead to failures, then proponents of fault injection argue that this suggests that the defect testing process will also have discovered X% of the actual faults in the program.

This approach assumes that the distribution and type of injected faults reflect the actual faults in the system. It is reasonable to think that this might be true for faults due to programming errors, but it is less likely to be true for faults resulting from requirements or design problems. Fault injection is ineffective in predicting the number of faults that stem from anything but programming errors.

Reliability growth modeling

A reliability growth model is a model of how the system reliability changes over time during the testing process. As system failures are discovered, the underlying faults causing these failures are repaired so that the reliability of the system should improve during system testing and debugging. To predict reliability, the conceptual reliability growth model must then be translated into a mathematical model.

http://software-engineering-book.com/web/reliability-growth-modeling/

11.5.1 Operational profiles

The operational profile of a software system reflects how it will be used in practice. It consists of a specification of classes of input and the probability of their occurrence. When a new software system replaces an existing automated system, it is reasonably easy to assess the probable pattern of usage of the new software. It should correspond to the existing usage, with some allowance made for the new functionality that is (presumably) included in the new software. For example, an operational profile can be specified for telephone switching systems because telecommunication companies know the call patterns that these systems have to handle.

Typically, the operational profile is such that the inputs that have the highest probability of being generated fall into a small number of classes, as shown on the left of Figure 11.14. There are many classes where inputs are highly improbable but not impossible. These are shown on the right of Figure 11.14. The ellipsis (. . .) means that there are many more of these uncommon inputs than are shown.

Musa (Musa 1998) discusses the development of operational profiles in telecommunication systems. As there is a long history of collecting usage data in that domain, the process of operational profile development is relatively straightforward. It simply reflects the historical usage data. For a system that required about 15 person-years of development effort, an operational profile was developed in about 1 person-month. In other cases, operational profile generation took longer (2–3 person-years), but the cost was spread over a number of system releases.

When a software system is new and innovative, however, it is difficult to anticipate how it will be used. Consequently, it is practically impossible to create an accurate operational profile. Many different users with different expectations, backgrounds, and experience may use the new system. There is no historical usage database. These users may make use of systems in ways that the system developers did not anticipate.

Developing an accurate operational profile is certainly possible for some types of system, such as telecommunication systems, that have a standardized pattern of use. However, for other types of system, developing an accurate operational profile may be difficult or impossible:

1. A system may have many different users who each have their own ways of using the system. As I explained earlier in this chapter, different users have

Figure 11.14
Distribution of inputs in an operational profile

different impressions of reliability because they use a system in different ways. It is difficult to match all of these patterns of use in a single operational profile.

2. Users change the ways that they use a system over time. As users learn about a new system and become more confident with it, they start to use it in more sophisticated ways. Therefore, an operational profile that matches the initial usage pattern of a system may not be valid after users become familiar with the system.

For these reasons, it is often impossible to develop a trustworthy operational profile. If you use an out-of-date or incorrect operational profile, you cannot be confident about the accuracy of any reliability measurements that you make.

KEY POINTS

- Software reliability can be achieved by avoiding the introduction of faults, by detecting and removing faults before system deployment, and by including fault-tolerance facilities that allow the system to remain operational after a fault has caused a system failure.
- Reliability requirements can be defined quantitatively in the system requirements specification. Reliability metrics include probability of failure on demand (POFOD), rate of occurrence of failure (ROCOF), and availability (AVAIL).
- Functional reliability requirements are requirements for system functionality, such as checking and redundancy requirements, which help the system meet its non-functional reliability requirements.

- Dependable system architectures are system architectures that are designed for fault tolerance. A number of architectural styles support fault tolerance, including protection systems, self-monitoring architectures, and *N*-version programming.
- Software diversity is difficult to achieve because it is practically impossible to ensure that each version of the software is truly independent.
- Dependable programming relies on including redundancy in a program as checks on the validity of inputs and the values of program variables.
- Statistical testing is used to estimate software reliability. It relies on testing the system with test data that matches an operational profile, which reflects the distribution of inputs to the software when it is in use.

FURTHER READING

Software Fault Tolerance Techniques and Implementation. A comprehensive discussion of techniques to achieve software fault tolerance and fault-tolerant architectures. The book also covers general issues of software dependability. Reliability engineering is a mature area, and the techniques discussed here are still current. (L. L. Pullum, Artech House, 2001).

"Software Reliability Engineering: A Roadmap." This survey paper by a leading researcher in software reliability summarizes the state of the art in software reliability engineering and discusses research challenges in this area. (M. R. Lyu, *Proc. Future of Software Engineering*, IEEE Computer Society, 2007) http://dx.doi.org/10.1109/FOSE.2007.24

"Mars Code." This paper discusses the approach to reliability engineering used in the development of software for the Mars Curiosity Rover. This relied on the use of good programming practice, redundancy, and model checking (covered in Chapter 12). (G. J. Holzmann, *Comm. ACM*., 57 (2), 2014) http://dx.doi.org/10.1145/2560217.2560218

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/reliability-and-safety/

More information on the Airbus flight control system:

http://software-engineering-book.com/case-studies/airbus-340/

EXERCISES

11.1. Explain why it is practically impossible to validate reliability specifications when these are expressed in terms of a very small number of failures over the total lifetime of a system.

11.2. Suggest appropriate reliability metrics for the classes of software system below. Give reasons for your choice of metric. Predict the usage of these systems and suggest appropriate values for the reliability metrics.

- a system that monitors patients in a hospital intensive care unit
- a word processor
- an automated vending machine control system
- a system to control braking in a car
- a system to control a refrigeration unit
- a management report generator

11.3. Imagine that a network operations center monitors and controls the national telecommunications network of a country. This includes controlling and monitoring the operational status of switching and transmission equipment and keeping track of nationwide equipment inventories. The center needs to have redundant systems. Explain three reliability metrics you would use to specify the needs of such systems.

11.4. What is the common characteristic of all architectural styles that are geared to supporting software fault tolerance?

11.5. Suggest circumstances where it is appropriate to use a fault-tolerant architecture when implementing a software-based control system and explain why this approach is required.

11.6. You are responsible for the design of a communications switch that has to provide 24/7 availability but that is not safety-critical. Giving reasons for your answer, suggest an architectural style that might be used for this system.

11.7. It has been suggested that the control software for a radiation therapy machine, used to treat patients with cancer, should be implemented using *N*-version programming. Comment on whether or not you think this is a good suggestion.

11.8. Explain why all the versions in a system designed around software diversity may fail in a similar way.

11.9. Explain how programming language support of exception handling can contribute to the reliability of software systems.

11.10. Software failures can cause considerable inconvenience to users of the software. Is it ethical for companies to release software that they know includes faults that could lead to software failures? Should they be liable for compensating users for losses that are caused by the failure of their software? Should they be required by law to offer software warranties in the same way that consumer goods manufacturers must guarantee their products?

REFERENCES

Avizienis, A. A. 1995. "A Methodology of N-Version Programming." In *Software Fault Tolerance*, edited by M. R. Lyu, 23–46. Chichester, UK: John Wiley & Sons.

Brilliant, S. S., J. C. Knight, and N. G. Leveson. 1990. "Analysis of Faults in an N-Version Software Experiment." *IEEE Trans. On Software Engineering* 16 (2): 238–247. doi:10.1109/32.44387.

Hatton, L. 1997. "N-Version Design Versus One Good Version." *IEEE Software* 14 (6): 71–76. doi:10.1109/52.636672.

Leveson, N. G. 1995. *Safeware: System Safety and Computers*. Reading, MA: Addison-Wesley.

Musa, J. D. 1998. *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*. New York: McGraw-Hill.

Prowell, S. J., C. J. Trammell, R. C. Linger, and J. H. Poore. 1999. *Cleanroom Software Engineering: Technology and Process*. Reading, MA: Addison-Wesley.

Pullum, L. 2001. *Software Fault Tolerance Techniques and Implementation*. Norwood, MA: Artech House.

Randell, B. 2000. "Facing Up To Faults." *Computer J.* 45 (2): 95–106. doi:10.1093/comjnl/43.2.95.

Torres-Pomales, W. 2000. "Software Fault Tolerance: A Tutorial." NASA. http://ntrs.nasa.gov/archive/nasa/casi. . ./20000120144_2000175863.pdf

Voas, J., and G. McGraw. 1997. *Software Fault Injection: Innoculating Programs Against Errors*. New York: John Wiley & Sons.

12

Safety engineering

Objectives

The objective of this chapter is to explain techniques that are used to ensure safety when developing critical systems. When you have read this chapter, you will:

- understand what is meant by a safety-critical system and why safety has to be considered separately from reliability in critical systems engineering;
- understand how an analysis of hazards can be used to derive safety requirements;
- know about processes and tools that are used for software safety assurance;
- understand the notion of a safety case that is used to justify the safety of a system to regulators, and how formal arguments may be used in safety cases.

Contents

12.1 Safety-critical systems
12.2 Safety requirements
12.3 Safety engineering processes
12.4 Safety cases

In Section 11.2, I briefly described an air accident at Warsaw Airport where an Airbus crashed on landing. Two people were killed and 54 were injured. The subsequent inquiry showed that a major contributory cause of the accident was a failure of the control software that reduced the efficiency of the aircraft's braking system. This is one of the, thankfully rare, examples of where the behavior of a software system has led to death or injury. It illustrates that software is now a central component in many systems that are critical to preserving and maintaining life. These are safety-critical software systems, and a range of specialized methods and techniques have been developed for safety-critical software engineering.

As I discussed in Chapter 10, safety is one of the principal dependability properties. A system can be considered to be safe if it operates without catastrophic failure, that is, failure that causes or may cause death or injury to people. Systems whose failure may lead to environmental damage may also be safety-critical as environmental damage (such as a chemical leak) can lead to subsequent human injury or death.

Software in safety-critical systems has a dual role to play in achieving safety:

1. The system may be software-controlled so that the decisions made by the software and subsequent actions are safety-critical. Therefore, the software behavior is directly related to the overall safety of the system.

2. Software is extensively used for checking and monitoring other safety-critical components in a system. For example, all aircraft engine components are monitored by software looking for early indications of component failure. This software is safety-critical because, if it fails, other components may fail and cause an accident.

Safety in software systems is achieved by developing an understanding of the situations that might lead to safety-related failures. The software is engineered so that such failures do not occur. You might therefore think that if a safety-critical system is reliable and behaves as specified, it will therefore be safe. Unfortunately, it isn't quite as simple as that. System reliability is necessary for safety achievement, but it isn't enough. Reliable systems can be unsafe and vice versa. The Warsaw Airport accident was an example of such a situation, which I'll discuss in more detail in Section 12.2.

Software systems that are reliable may not be safe for four reasons:

1. We can never be 100% certain that a software system is fault-free and fault-tolerant. Undetected faults can be dormant for a long time, and software failures can occur after many years of reliable operation.

2. The specification may be incomplete in that it does not describe the required behavior of the system in some critical situations. A high percentage of system malfunctions are the result of specification rather than design errors. In a study of errors in embedded systems, Lutz (Lutz 1993) concludes that "difficulties with requirements are the key root cause of the safety-related software errors, which have persisted until integration and system testing.[†]"

[†]Lutz, R R. 1993. "Analysing Software Requirements Errors in Safety-Critical Embedded Systems." In RE'93, 126–133. San Diego CA: IEEE. doi:0.1109/ISRE.1993.324825.

More recent work by Veras et al. (Veras et al. 2010) in space systems confirms that requirements errors are still a major problem for embedded systems.

3. Hardware malfunctions may cause sensors and actuators to behave in an unpredictable way. When components are close to physical failure, they may behave erratically and generate signals that are outside the ranges that can be handled by the software. The software may then either fail or wrongly interpret these signals.

4. The system operators may generate inputs that are not individually incorrect but that, in some situations, can lead to a system malfunction. An anecdotal example of this occurred when an aircraft undercarriage collapsed while the aircraft was on the ground. Apparently, a technician pressed a button that instructed the utility management software to raise the undercarriage. The software carried out the mechanic's instruction perfectly. However, the system should have disallowed the command unless the plane was in the air.

Therefore, safety has to be considered as well as reliability when developing safety-critical systems. The reliability engineering techniques that I introduced in Chapter 11 are obviously applicable for safety-critical systems engineering. I therefore do not discuss system architectures and dependable programming here but instead focus on techniques for improving and assuring system safety.

12.1 Safety-critical systems

Safety-critical systems are systems in which it is essential that system operation is always safe. That is, the system should never damage people or the system's environment, irrespective of whether or not the system conforms to its specification. Examples of safety-critical systems include control and monitoring systems in aircraft, process control systems in chemical and pharmaceutical plants, and automobile control systems.

Safety-critical software falls into two classes:

1. *Primary safety-critical software* This is software that is embedded as a controller in a system. Malfunctioning of such software can cause a hardware malfunction, which results in human injury or environmental damage. The insulin pump software that I introduced in Chapter 1 is an example of a primary safety-critical system. System failure may lead to user injury.

   The insulin pump system is a simple system, but software control is also used in very complex safety-critical systems. Software rather than hardware control is essential because of the need to manage large numbers of sensors and actuators, which have complex control laws. For example, advanced, aerodynamically unstable, military aircraft require continual software-controlled adjustment of their flight surfaces to ensure that they do not crash.

2. *Secondary safety-critical software* This is software that can indirectly result in an injury. An example of such software is a computer-aided engineering design system

whose malfunctioning might result in a design fault in the object being designed. This fault may cause injury to people if the designed system malfunctions. Another example of a secondary safety-critical system is the Mentcare system for mental health patient management. Failure of this system, whereby an unstable patient may not be treated properly, could lead to that patient injuring himself or others.

Some control systems, such as those controlling critical national infrastructure (electricity supply, telecommunications, sewage treatment, etc.), are secondary safety-critical systems. Failure of these systems is unlikely to have immediate human consequences. However, a prolonged outage of the controlled systems could lead to injury and death. For example, failure of a sewage treatment system could lead to a higher level of infectious disease as raw sewage is released into the environment.

I explained in Chapter 11 how software and system availability and reliability are achieved through fault avoidance, fault detection and removal, and fault tolerance. Safety-critical systems development uses these approaches and augments them with hazard-driven techniques that consider the potential system accidents that may occur:

1. *Hazard avoidance* The system is designed so that hazards are avoided. For example, a paper-cutting system that requires an operator to use two hands to press separate buttons simultaneously avoids the hazard of the operator's hands being in the blade's pathway.
2. *Hazard detection and removal* The system is designed so that hazards are detected and removed before they result in an accident. For example, a chemical plant system may detect excessive pressure and open a relief valve to reduce pressure before an explosion occurs.
3. *Damage limitation* The system may include protection features that minimize the damage that may result from an accident. For example, an aircraft engine normally includes automatic fire extinguishers. If there is an engine fire, it can often be controlled before it poses a threat to the aircraft.

A hazard is a system state that could lead to an accident. Using the above example of the paper-cutting system, a hazard arises when the operator's hand is in a position where the cutting blade could injure it. Hazards are not accidents—we often get ourselves into hazardous situations and get out of them without any problems. However, accidents are always preceded by hazards, so reducing hazards reduces accidents.

A hazard is one example of the specialized vocabulary that is used in safety-critical systems engineering. I explain other terminology used in safety-critical systems in Figure 12.1.

We are now actually pretty good at building systems that can cope with one thing going wrong. We can design mechanisms into the system that can detect and recover from single problems. However, when several things go wrong at the same time, accidents are more likely. As systems become more and more complex, we don't understand the relationships between the different parts of the system. Consequently, we cannot predict the consequences of a combination of unexpected system events or failures.

In an analysis of serious accidents, Perrow (Perrow 1984) suggested that almost all of the accidents were due to a combination of failures in different parts of a system.

Term	Definition
Accident (or mishap)	An unplanned event or sequence of events that results in human death or injury, damage to property or to the environment. An overdose of insulin is an example of an accident.
Damage	A measure of the loss resulting from a mishap. Damage can range from many people being killed as a result of an accident to minor injury or property damage. Damage resulting from an overdose of insulin could lead to serious injury or the death of the user of the insulin pump.
Hazard	A condition with the potential for causing or contributing to an accident. A failure of the sensor that measures blood glucose is an example of a hazard.
Hazard probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from "probable" (say 1/100 chance of a hazard occurring) to "implausible" (no conceivable situations are likely in which the hazard could occur). The probability of a sensor failure in the insulin pump that overestimates the user's blood sugar level is low.
Hazard severity	An assessment of the worst possible damage that could result from a particular hazard. Hazard severity can range from catastrophic, where many people are killed, to minor, where only minor damage results. When an individual death is a possibility, a reasonable assessment of hazard severity is "very high."
Risk	A measure of the probability that the system will cause an accident. The risk is assessed by considering the hazard probability, the hazard severity, and the probability that the hazard will lead to an accident. The risk of an insulin overdose is medium to low.

Figure 12.1 Safety terminology

Unanticipated combinations of subsystem failures led to interactions that resulted in overall system failure. For example, failure of an air conditioning system may lead to overheating. Once hardware gets hot, its behavior becomes unpredictable, so overheating may lead to the system hardware generating incorrect signals. These wrong signals may then cause the software to react incorrectly.

Perrow made the point that, in complex systems, it is impossible to anticipate all possible combinations of failures. He therefore coined the phrase "normal accidents," with the implication that accidents have to be considered as inevitable when we build complex safety-critical systems.

To reduce complexity, we could use simple hardware controllers rather than software control. However, software-controlled systems can monitor a wider range of conditions than simpler electromechanical systems. They can be adapted relatively easily. They use computer hardware, which has high inherent reliability and which is physically small and lightweight.

Software-controlled systems can provide sophisticated safety interlocks. They can support control strategies that reduce the amount of time people need to spend in hazardous environments. Although software control may introduce more ways in which a system can go wrong, it also allows better monitoring and protection. Therefore, software control can contribute to improvements in system safety.

It is important to maintain a sense of proportion about safety-critical systems. Critical software systems operate without problems most of the time. Relatively few people worldwide have been killed or injured because of faulty software. Perrow is right in say-

Risk-based requirements specification

Risk-based specification is an approach that has been widely used by safety and security-critical systems developers. It focuses on those events that could cause the most damage or that are likely to occur frequently. Events that have only minor consequences or that are extremely rare may be ignored. The risk-based specification process involves understanding the risks faced by the system, discovering their root causes, and generating requirements to manage these risks.

http://software-engineering-book.com/web/risk-based-specification/

ing that accidents will always be a possibility. It is impossible to make a system 100% safe, and society has to decide whether or not the consequences of an occasional accident are worth the benefits that come from the use of advanced technologies.

12.2 Safety requirements

In the introduction to this chapter, I described an air accident at Warsaw Airport where the braking system on an Airbus failed. The inquiry into this accident showed that the braking system software had operated according to its specification. There were no errors in the program. However, the software specification was incomplete and had not taken into account a rare situation, which arose in this case. The software worked, but the system failed.

This episode illustrates that system safety does not just depend on good engineering. It requires attention to detail when the system requirements are derived and the inclusion of special software requirements that are geared to ensuring the safety of a system. Safety requirements are functional requirements, which define checking and recovery facilities that should be included in the system and features that provide protection against system failures and external attacks.

The starting point for generating functional safety requirements is usually domain knowledge, safety standards, and regulations. These lead to high-level requirements that are perhaps best described as "shall not" requirements. By contrast with normal functional requirements that define what the system shall do, "shall not" requirements define system behavior that is unacceptable. Examples of "shall not" requirements are:

> "The system shall not allow reverse thrust mode to be selected when the aircraft is in flight."
>
> "The system shall not allow the simultaneous activation of more than three alarm signals."
>
> "The navigation system shall not allow users to set the required destination when the car is moving."

These "shall not" requirements cannot be implemented directly but have to be decomposed into more specific software functional requirements. Alternatively, they may be implemented through system design decisions such as a decision to use particular types of equipment in the system.

Figure 12.2 Hazard-driven requirements specification

Safety requirements are primarily protection requirements and are not concerned with normal system operation. They may specify that the system should be shut down so that safety is maintained. In deriving safety requirements, you therefore need to find an acceptable balance between safety and functionality and avoid overprotection. There is no point in building a very safe system if it does not operate in a cost-effective way.

Risk-based requirements specification is a general approach used in critical systems engineering where risks faced by the system are identified and requirements to avoid or mitigate these risks are identified. It may be used for all types of dependability requirements. For safety-critical systems, it translates into a process driven by identified hazards. As I discussed in the previous section, a hazard is something that could (but need not) result in death or injury to a person.

There are four activities in a hazard-driven safety specification process:

1. *Hazard identification* The hazard identification process identifies hazards that may threaten the system. These hazards may be recorded in a hazard register. This is a formal document that records the safety analyses and assessments and that may be submitted to a regulator as part of a safety case.

2. *Hazard assessment* The hazard assessment process decides which hazards are the most dangerous and/or the most likely to occur. These should be prioritized when deriving safety requirements.

3. *Hazard analysis* This is a process of root-cause analysis that identifies the events that can lead to the occurrence of a hazard.

4. *Risk reduction* This process is based on the outcome of hazard analysis and leads to identification of safety requirements. These requirements may be concerned with ensuring that a hazard does not arise or lead to an accident or that if an accident does occur, the associated damage is minimized.

Figure 12.2 illustrates this hazard-driven safety requirements specification process.

12.2.1 Hazard identification

In safety-critical systems, hazard identification starts by identifying different classes of hazards, such as physical, electrical, biological, radiation, and service failure hazards. Each of these classes can then be analyzed to discover specific hazards that could occur. Possible combinations of hazards that are potentially dangerous must also be identified.

Experienced engineers, working with domain experts and professional safety advisers, identify hazards from previous experience and from an analysis of the application domain. Group working techniques such as brainstorming may be used, where a group meets to exchange ideas. For the insulin pump system, people who may be involved include doctors, medical physicists and engineers, and software designers.

The insulin pump system that I introduced in Chapter 1 is a safety-critical system, because failure can cause injury or even death to the system user. Accidents that may occur when using this machine include the user suffering from long-term consequences of poor blood sugar control (eye, heart, and kidney problems), cognitive dysfunction as a result of low blood sugar levels, or the occurrence of some other medical conditions, such as an allergic reaction.

Some of the hazards that may arise in the insulin pump system are:

- insulin overdose computation (service failure);
- insulin underdose computation (service failure);
- failure of the hardware monitoring system (service failure);
- power failure due to exhausted battery (electrical);
- electrical interference with other medical equipment such as a heart pacemaker (electrical);
- poor sensor and actuator contact caused by incorrect fitting (physical);
- parts of machine breaking off in patient's body (physical);
- infection caused by introduction of machine (biological); and
- allergic reaction to the materials or insulin used in the machine (biological).

Software-related hazards are normally concerned with failure to deliver a system service or with the failure of monitoring and protection systems. Monitoring and protection systems may be included in a device to detect conditions, such as a low battery level, which could lead to device failure.

A hazard register may be used to record the identified hazards with an explanation of why the hazard has been included. The hazard register is an important legal document that records all safety-related decisions about each hazard. It can be used to show that the requirements engineers have paid due care and attention in considering all foreseeable hazards and that these hazards have been analyzed. In the event of an accident, the hazard register may be used in a subsequent inquiry or legal proceedings to show that the system developers have not been negligent in their system safety analysis.

12.2.2 Hazard assessment

The hazard assessment process focuses on understanding the factors that lead to the occurrence of a hazard and the consequences if an accident or incident associated with that hazard should occur. You need to carry out this analysis to understand

Figure 12.3 The risk triangle

whether a hazard is a serious threat to the system or environment. The analysis also provides a basis for deciding on how to manage the risk associated with the hazard.

For each hazard, the outcome of the analysis and classification process is a statement of acceptability. This is expressed in terms of risk, where the risk takes into account the likelihood of an accident and its consequences. There are three risk categories that are used in hazard assessment:

1. *Intolerable risks* in safety-critical systems are those that threaten human life. The system must be designed so that such hazards either cannot arise or, that if they do, features in the system will ensure that they are detected before they cause an accident. In the case of the insulin pump, an intolerable risk is that an overdose of insulin should be delivered.

2. *As low as reasonably practical (ALARP) risks* are those that have less serious consequences or that are serious but have a very low probability of occurrence. The system should be designed so that the probability of an accident arising because of a hazard is minimized, subject to other considerations such as cost and delivery. An ALARP risk for an insulin pump might be the failure of the hardware monitoring system. The consequences of this failure are, at worst, a short-term insulin underdose. This situation would not lead to a serious accident.

3. *Acceptable risks* are those where the associated accidents normally result in minor damage. System designers should take all possible steps to reduce "acceptable" risks, as long as these measures do not significantly increase costs, delivery time, or other non-functional system attributes. An acceptable risk in the case of the insulin pump might be the risk of an allergic reaction arising in the user. This reaction usually causes only minor skin irritation. It would not be worth using special, more expensive materials in the device to reduce this risk.

Figure 12.3 shows these three regions. The width of the triangle reflects the costs of ensuring that risks do not result in incidents or accidents. The highest

Identified hazard	Hazard probability	Accident severity	Estimated risk	Acceptability
1. Insulin overdose computation	Medium	High	High	Intolerable
2. Insulin underdose computation	Medium	Low	Low	Acceptable
3. Failure of hardware monitoring system	Medium	Medium	Low	ALARP
4. Power failure	High	Low	Low	Acceptable
5. Machine incorrectly fitted	High	High	High	Intolerable
6. Machine breaks in patient	Low	High	Medium	ALARP
7. Machine causes infection	Medium	Medium	Medium	ALARP
8. Electrical interference	Low	High	Medium	ALARP
9. Allergic reaction	Low	Low	Low	Acceptable

Figure 12.4 Risk classification for the insulin pump

costs are incurred by risks at the top of the diagram, the lowest costs by risks at the apex of the triangle.

The boundaries between the regions in Figure 12.3 are not fixed but depend on how acceptable risks are in the societies where the system will be deployed. This varies from country to country—some societies are more risk averse and litigious than others. Over time, however, all societies have become more risk-averse, so the boundaries have moved downward. For rare events, the financial costs of accepting risks and paying for any resulting accidents may be less than the costs of accident prevention. However, public opinion may demand that money be spent to reduce the likelihood of a system accident irrespective of cost.

For example, it may be cheaper for a company to clean up pollution on the rare occasion it occurs, rather than to install systems for pollution prevention. However, because the public and the media will not tolerate such accidents, clearing up the damage rather than preventing the accident is no longer acceptable. Events in other systems may also lead to a reclassification of risk. For example, risks that were thought to be improbable (and hence in the ALARP region) may be reclassified as intolerable because of external events, such as terrorist attacks, or natural phenomena, such as tsunamis.

Figure 12.4 shows a risk classification for the hazards identified in the previous section for the insulin delivery system. I have separated the hazards that relate to the incorrect computation of insulin into an insulin overdose and an insulin underdose. An insulin overdose is potentially more serious than an insulin underdose in the short term. Insulin overdose can result in cognitive dysfunction, coma, and ultimately death. Insulin underdoses lead to high levels of blood sugar. In the short term, these high levels cause tiredness but are not very serious; in the longer term, however, they can lead to serious heart, kidney, and eye problems.

Hazards 4–9 in Figure 12.4 are not software related, but software nevertheless has a role to play in hazard detection. The hardware monitoring software should monitor the system state and warn of potential problems. The warning will often allow the hazard to

be detected before it causes an accident. Examples of hazards that might be detected are power failure, which is detected by monitoring the battery, and incorrect fitting of the machine, which may be detected by monitoring signals from the blood sugar sensor.

The monitoring software in the system is, of course, safety-related. Failure to detect a hazard could result in an accident. If the monitoring system fails but the hardware is working correctly, then this is not a serious failure. However, if the monitoring system fails and hardware failure cannot then be detected, then this could have more serious consequences.

Hazard assessment involves estimating the hazard probability and risk severity. This is difficult as hazards and accidents are uncommon. Consequently, the engineers involved may not have direct experience of previous incidents or accidents. In estimating probabilities and accident severity, it makes sense to use relative terms such as *probable*, *unlikely*, *rare*, *high*, *medium*, and *low*. Quantifying these terms is practically impossible because not enough statistical data is available for most types of accident.

12.2.3 Hazard analysis

Hazard analysis is the process of discovering the root causes of hazards in a safety-critical system. Your aim is to find out what events or combination of events could cause a system failure that results in a hazard. To do this, you can use either a top-down or a bottom-up approach. Deductive, top-down techniques, which are easier to use, start with the hazard and work from that to the possible system failure. Inductive, bottom-up techniques start with a proposed system failure and identify what hazards might result from that failure.

Various techniques have been proposed as possible approaches to hazard decomposition or analysis (Storey 1996). One of the most commonly used techniques is fault tree analysis, a top-down technique that was developed for the analysis of both hardware and software hazards (Leveson, Cha, and Shimeall 1991). This technique is fairly easy to understand without specialist domain knowledge.

To do a fault tree analysis, you start with the hazards that have been identified. For each hazard, you then work backwards to discover the possible causes of that hazard. You put the hazard at the root of the tree and identify the system states that can lead to that hazard. For each of these states, you then identify further system states that can lead to them. You continue this decomposition until you reach the root cause(s) of the risk. Hazards that can only arise from a combination of root causes are usually less likely to lead to an accident than hazards with a single root cause.

Figure 12.5 is a fault tree for the software-related hazards in the insulin delivery system that could lead to an incorrect dose of insulin being delivered. In this case, I have merged insulin underdose and insulin overdose into a single hazard, namely, "incorrect insulin dose administered." This reduces the number of fault trees that are required. Of course, when you specify how the software should react to this hazard, you have to distinguish between an insulin underdose and an insulin overdose. As I have said, they are not equally serious—in the short term, an overdose is the more serious hazard.

From Figure 12.5, you can see that:

1. Three conditions could lead to the administration of an incorrect dose of insulin. (1) The level of blood sugar may have been incorrectly measured, so the insulin requirement has been computed with an incorrect input. (2) The delivery system

Figure 12.5 An example of a fault tree

may not respond correctly to commands specifying the amount of insulin to be injected. Alternatively, (3) the dose may be correctly computed, but it is delivered too early or too late.

2. The left branch of the fault tree, concerned with incorrect measurement of the blood sugar level, identifies how this might happen. This could occur either because the sensor that provides an input to calculate the sugar level has failed or because the calculation of the blood sugar level has been carried out incorrectly. The sugar level is calculated from some measured parameter, such as the conductivity of the skin. Incorrect computation can result from either an incorrect algorithm or an arithmetic error that results from the use of floating-point numbers.

3. The central branch of the tree is concerned with timing problems and concludes that these can only result from system timer failure.

4. The right branch of the tree, concerned with delivery system failure, examines possible causes of this failure. These could result from an incorrect computation of the insulin requirement or from a failure to send the correct signals to the pump that delivers the insulin. Again, an incorrect computation can result from algorithm failure or arithmetic errors.

Fault trees are also used to identify potential hardware problems. Hardware fault trees may provide insights into requirements for software to detect and, perhaps, correct these problems. For example, insulin doses are not administered frequently—no more than five or six times per hour and sometimes less often than that. Therefore, processor capacity is available to run diagnostic and self-checking programs. Hardware errors such as sensor, pump, or timer errors can be discovered and warnings issued before they have a serious effect on the patient.

12.2.4 Risk reduction

Once potential risks and their root causes have been identified, you are then able to derive safety requirements that manage the risks and ensure that incidents or accidents do not occur. You can use three possible strategies:

1. *Hazard avoidance*, where a system is designed so that the hazard cannot occur.
2. *Hazard detection and removal*, where a system is designed so that hazards are detected and neutralized before they result in an accident.
3. *Damage limitation*, where a system is designed so that the consequences of an accident are minimized.

Normally, designers of critical systems use a combination of these approaches. In a safety-critical system, intolerable hazards may be handled by minimizing their probability and adding a protection system (see Chapter 11) that provides a safety backup. For example, in a chemical plant control system, the system will attempt to detect and avoid excess pressure in the reactor. However, there may also be an independent protection system that monitors the pressure and opens a relief valve if high pressure is detected.

In the insulin delivery system, a safe state is a shutdown state where no insulin is injected. Over a short period, this is not a threat to the diabetic's health. For the software failures that could lead to an incorrect dose of insulin, the following "solutions" might be developed:

1. *Arithmetic error* This error may occur when an arithmetic computation causes a representation failure. The specification should identify all possible arithmetic errors that may occur and state that an exception handler must be included for each possible error. The specification should set out the action to be taken for each of these errors. The default safe action is to shut down the delivery system and activate a warning alarm.
2. *Algorithmic error* This is a more difficult situation as there is no clear program exception that must be handled. This type of error could be detected by comparing

SR1: The system shall not deliver a single dose of insulin that is greater than a specified maximum dose for a system user.

SR2: The system shall not deliver a daily cumulative dose of insulin that is greater than a specified maximum daily dose for a system user.

SR3: The system shall include a hardware diagnostic facility that shall be executed at least four times per hour.

SR4: The system shall include an exception handler for all of the exceptions that are identified in Table 3.

SR5: The audible alarm shall be sounded when any hardware or software anomaly is discovered and a diagnostic message as defined in Table 4 shall be displayed.

SR6: In the event of an alarm, insulin delivery shall be suspended until the user has reset the system and cleared the alarm.

Note: Tables 3 and 4 relate to tables that are included in the requirements document; they are not shown here.

Figure 12.6 Examples of safety requirements

the required insulin dose computed with the previously delivered dose. If it is much higher, this may mean that the amount has been computed incorrectly. The system may also keep track of the dose sequence. After a number of above-average doses have been delivered, a warning may be issued and further dosage limited.

Some of the resulting safety requirements for the insulin pump software are shown in Figure 12.6. The requirements in Figure 12.6 are user requirements. Naturally, they would be expressed in more detail in a more detailed system requirements specification.

12.3 Safety engineering processes

The software processes used to develop safety-critical software are based on the processes used in software reliability engineering. In general, a great deal of care is taken in developing a complete, and often very detailed, system specification. The design and implementation of the system usual follow a plan-based, waterfall model, with reviews and checks at each stage in the process. Fault avoidance and fault detection are the drivers of the process. For some types of system, such as aircraft systems, fault-tolerant architectures, as I discussed in Chapter 11, may be used.

Reliability is a prerequisite for safety-critical systems. Because of the very high costs and potentially tragic consequences of system failure, additional verification activities may be used in safety-critical systems development. These activities may include developing formal models of a system, analyzing them to discover errors and inconsistencies, and using static analysis software tools that parse the software source code to discover potential faults.

Safe systems have to be reliable, but, as I have discussed, reliability is not enough. Requirements and verification errors and omissions may mean that reliable systems are unsafe. Therefore, safety-critical systems development processes should include

safety reviews, where engineers and system stakeholders examine the work done and explicitly look for potential issues that could affect the safety of the system.

Some types of safety-critical systems are regulated, as I explained in Chapter 10. National and international regulators require detailed evidence that the system is safe. This evidence might include:

1. The specification of the system that has been developed and records of the checks made on that specification.
2. Evidence of the verification and validation processes that have been carried out and the results of the system verification and validation.
3. Evidence that the organizations developing the system have defined and dependable software processes that include safety assurance reviews. There must also be records showing that these processes have been properly enacted.

Not all safety-critical systems are regulated. For example, there is no regulator for automobiles, although cars now have many embedded computer systems. The safety of car-based systems is the responsibility of the car manufacturer. However, because of the possibility of legal action in the event of an accident, developers of unregulated systems have to maintain the same detailed safety information. If a case is brought against them, they have to be able to show that they have not been negligent in the development of the car's software.

The need for this extensive process and product documentation is another reason why agile processes cannot be used, without significant change, for safety-critical systems development. Agile processes focus on the software itself and (rightly) argue that a great deal of process documentation is never actually used after it has been produced. However, where you have to keep records for legal or regulatory reasons, you must maintain documentation about both the processes used and the system itself.

Safety-critical systems, like other types of system that have high dependability requirements, need to be based on dependable processes (see Chapter 10). A dependable process will normally include activities such as requirements management, change management and configuration control, system modeling, reviews and inspections, test planning, and test coverage analysis. When a system is safety-critical, there may be additional safety assurance and verification and analyses processes.

12.3.1 Safety assurance processes

Safety assurance is a set of activities that check that a system will operate safely. Specific safety assurance activities should be included at all stages in the software development process. These activities record the safety analyses that have been carried out and the person or persons responsible for these analyses. Safety assurance activities have to be thoroughly documented. This documentation may be part of the evidence that is used to convince a regulator or system owner that a system will operate safely.

Examples of safety assurance activities are:

1. *Hazard analysis and monitoring*, where hazards are traced from preliminary hazard analysis through to testing and system validation.
2. *Safety reviews*, which are used throughout the development process.
3. *Safety certification*, where the safety of critical components is formally certified. This involves a group external to the system development team examining the available evidence and deciding whether or not a system or component should be considered to be safe before it is made available for use.

To support these safety assurance processes, project safety engineers should be appointed who have explicit responsibility for the safety aspects of a system. These individuals will be accountable if a safety-related system failure occurs. They must be able to demonstrate that the safety assurance activities have been properly carried out.

Safety engineers work with quality managers to ensure that a detailed configuration management system is used to track all safety-related documentation and keep it in step with the associated technical documentation. There is little point in having stringent validation procedures if a failure of configuration management means that the wrong system is delivered to the customer. Quality and configuration management are covered in Chapters 24 and 25.

Hazard analysis is an essential part of safety-critical systems development. It involves identifying hazards, their probability of occurrence, and the probability of a hazard leading to an accident. If there is program code that checks for and handles each hazard, then you can argue that these hazards will not result in accidents. Where external certification is required before a system is used (e.g., in an aircraft), it is usually a condition of certification that this traceability can be demonstrated.

The central safety document that should be produced is the hazard register. This document provides evidence of how identified hazards have been taken into account during software development. This hazard register is used at each stage of the software development process to document how that development stage has taken the hazards into account.

A simplified example of a hazard register entry for the insulin delivery system is shown in Figure 12.7. This register documents the process of hazard analysis and shows design requirements that have been generated during this process. These design requirements are intended to ensure that the control system can never deliver an insulin overdose to a user of the insulin pump.

Individuals who have safety responsibilities should be explicitly identified in the hazard register. Personal identification is important for two reasons:

1. When people are identified, they can be held accountable for their actions. They are likely to take more care because any problems can be traced back to their work.
2. In the event of an accident, there may be legal proceedings or an inquiry. It is important to be able to identify those responsible for safety assurance so that they can defend their actions as part of the legal process.

Hazard Register. **Page 4: Printed 20.02.2012**

System: Insulin Pump System *File:* InsulinPump/Safety/HazardLog
Safety Engineer: James Brown *Log version:* 1/3

Identified Hazard	Insulin overdose delivered to patient				
Identified by	Jane Williams				
Criticality class	1				
Identified risk	High				
Fault tree identified	YES	*Date*	24.01.11	*Location*	Hazard register, Page 5
Fault tree creators	Jane Williams and Bill Smith				
Fault tree checked	YES	*Date*	28.01.11	*Checker*	James Brown

System safety design requirements

1. The system shall include self-testing software that will test the sensor system, the clock, and the insulin delivery system.
2. The self-checking software shall be executed once per minute.
3. In the event of the self-checking software discovering a fault in any of the system components, an audible warning shall be issued and the pump display shall indicate the name of the component where the fault has been discovered. The delivery of insulin shall be suspended.
4. The system shall incorporate an override system that allows the system user to modify the computed dose of insulin that is to be delivered by the system.
5. The amount of override shall be no greater than a pre-set value (maxOverride), which is set when the system is configured by medical staff.

Figure 12.7
A simplified hazard register entry

Safety reviews are reviews of the software specification, design, and source code whose aim is to discover potentially hazardous conditions. These are not automated processes but involve people carefully checking for errors that have been made and for assumptions or omissions that may affect the safety of a system. For example, in the aircraft accident that I introduced earlier, a safety review might have questioned the assumption that an aircraft is on the ground when there is weight on both wheels and the wheels are rotating.

Safety reviews should be driven by the hazard register. For each of the identified hazards, a review team examines the system and judges whether or not it would cope with that hazard in a safe way. Any doubts raised are flagged in the review team's report and have to be addressed by the system development team. I discuss reviews of different types in more detail in Chapter 24, which covers software quality assurance.

Software safety certification is used when external components are incorporated into a safety-critical system. When all parts of a system have been locally developed, complete information about the development processes used can be maintained. However, it is not cost-effective to develop components that are readily available from other vendors. The problem for safety-critical systems development is that these external components may have been developed to different standards than locally developed components. Their safety is unknown.

Consequently, it may be a requirement that all external components must be certified before they can be integrated with a system. The safety certification team, which is separate from the development team, carries out extensive verification and validation of

Licensing of software engineers

In some areas of engineering, safety engineers must be licensed engineers. Inexperienced, poorly qualified engineers are not allowed to take responsibility for safety. In 30 states of the United States, there is some form of licensing for software engineers involved in safety-related systems development. These states require that engineering involved in safety-critical software development should be licensed engineers, with a defined minimum level of qualifications and experience. This is a controversial issue, and licensing is not required in many other countries.

http://software-engineering-book.com/safety-licensing/

the components. If appropriate, they liaise with the component developers to check that the developers have used dependable processes to create these components and to examine the component source code. Once the safety certification team is satisfied that a component meets its specification and does not have "hidden" functionality, they may issue a certificate allowing that component to be used in safety-critical systems.

12.3.2 Formal verification

Formal methods of software development, as I discussed in Chapter 10, rely on a formal model of the system that serves as a system specification. These formal methods are mainly concerned with mathematically analyzing the specification; with transforming the specification to a more detailed, semantically equivalent representation; or with formally verifying that one representation of the system is semantically equivalent to another representation.

The need for assurance in safety-critical systems has been one of the principal drivers in the development of formal methods. Comprehensive system testing is extremely expensive and cannot be guaranteed to uncover all of the faults in a system. This is particularly true of systems that are distributed, so that system components are running concurrently. Several safety-critical railway systems were developed using formal methods in the 1990s (Dehbonei and Mejia 1995; Behm et al. 1999). Companies such as Airbus routinely use formal methods in their software development for critical systems (Souyris et al. 2009).

Formal methods may be used at different stages in the V & V process:

1. A formal specification of the system may be developed and mathematically analyzed for inconsistency. This technique is effective in discovering specification errors and omissions. Model checking, discussed in the next section, is a particularly effective approach to specification analysis.

2. You can formally verify, using mathematical arguments, that the code of a software system is consistent with its specification. This requires a formal specification. It is effective in discovering programming and some design errors.

Because of the wide semantic gap between a formal system specification and program code, it is difficult and expensive to prove that a separately developed program is

consistent with its specification. Work on program verification is now mostly based on transformational development. In a transformational development process, a formal specification is systematically transformed through a series of representations to program code. Software tools support the development of the transformations and help verify that corresponding representations of the system are consistent. The B method is probably the most widely used formal transformational method (Abrial 2010). It has been used for the development of train control systems and avionics software.

Advocates of formal methods claim that the use of these methods leads to more reliable and safer systems. Formal verification demonstrates that the developed program meets its specification and that implementation errors will not compromise the dependability of the system. If you develop a formal model of concurrent systems using a specification written in a language such as CSP (Schneider 1999), you can discover conditions that might result in deadlock in the final program, and you will be able to address these problems. This is very difficult to do by testing alone.

However, formal specification and proof do not guarantee that the software will be safe in practical use:

1. The specification may not reflect the real requirements of users and other system stakeholders. As I discussed in Chapter 10, system system stakeholders rarely understand formal notations, so they cannot directly read the formal specification to find errors and omissions. This means that there it is likely that the formal specification is not an accurate representation of the system requirements.
2. The proof may contain errors. Program proofs are large and complex, so, like large and complex programs, they usually contain errors.
3. The proof may make incorrect assumptions about the way that the system is used. If the system is not used as anticipated, then the system's behavior lies outside the scope of the proof.

Verifying a nontrivial software system takes a great deal of time. It requires mathematical expertise and specialized software tools, such as theorem provers. It is an expensive process, and, as the system size increases, the costs of formal verification increase disproportionately.

Many software engineers therefore think that formal verification is not cost-effective. They believe that the same level of confidence in the system can be achieved more cheaply by using other validation techniques, such as inspections and system testing. However, companies such as Airbus that make use of formal verification claim that unit testing of components is not required, which leads to significant cost savings (Moy et al. 2013).

I am convinced that that formal methods and formal verification have an important role to play in the development of critical software systems. Formal specifications are very effective in discovering some types of specification problems that may lead to system failure. Although formal verification remains impractical for large systems, it can be used to verify critical safety and security critical core components.

Figure 12.8 Model checking

12.3.3 Model checking

Formally verifying programs using a deductive approach is difficult and expensive, but alternative approaches to formal analysis have been developed that are based on a more restricted notion of correctness. The most successful of these approaches is called model checking (Jhala and Majumdar 2009). Model checking involves creating a formal state model of a system and checking the correctness of that model using specialized software tools. The stages involved in model checking are shown in Figure 12.8.

Model checking has been widely used to check hardware systems designs. It is increasingly being used in critical software systems such as the control software in NASA's Mars exploration vehicles (Regan and Hamilton 2004; Holzmann 2014) and by Airbus in avionics software development (Bochot et al. 2009).

Many different model-checking tools have been developed. SPIN was an early example of a software model checker (Holzmann, 2003). More recent systems include SLAM from Microsoft (Ball, Levin, and Rajamani 2011) and PRISM (Kwiatkowska, Norman, and Parker 2011).

The models used by model-checking systems are extended finite-state models of the software. Models are expressed in the language of whatever model-checking system is used—for example, the SPIN model checker uses a language called Promela. A set of desirable system properties are identified and written in a formal notation, usually based on temporal logic. For example, in the wilderness weather system, a property to be checked might be that the system will always reach the "transmitting" state from the "recording" state.

The model checker then explores all paths through the model (i.e., all possible state transitions), checking that the property holds for each path. If it does, then the model checker confirms that the model is correct with respect to that property. If it does not hold for a particular path, the model checker outputs a counterexample illustrating where the property is not true. Model checking is particularly useful in the validation of concurrent systems, which are notoriously difficult to test because of their sensitivity to time. The checker can explore interleaved, concurrent transitions and discover potential problems.

A key issue in model checking is the creation of the system model. If the model has to be created manually (from a requirements or design document), it is an expensive process as model creation takes a great deal of time. In addition, there is the possibility that the model created will not be an accurate model of the requirements or design. It is therefore

best if the model can be created automatically from the program source code. Model checkers are available that work directly from programs in Java, C, C++, and Ada.

Model checking is computationally very expensive because it uses an exhaustive approach to check all paths through the system model. As the size of a system increases, so too does the number of states, with a consequent increase in the number of paths to be checked. For large systems, therefore, model checking may be impractical, due to the computer time required to run the checks. However, better algorithms are under development that can identify parts of the state that do not have to be explored when checking a particular property. As these algorithms are incorporated into model checkers, it will be increasingly possible to use model checking routinely in large-scale critical systems development.

12.3.4 Static program analysis

Automated static analyzers are software tools that scan the source text of a program and detect possible faults and anomalies. They parse the program text and thus recognize the different types of statements in a program. They can then detect whether or not statements are well formed, make inferences about the control flow in the program, and, in many cases, compute the set of all possible values for program data. They complement the error-detection facilities provided by the language compiler, and they can be used as part of the inspection process or as a separate V & V process activity.

Automated static analysis is faster and cheaper than detailed code reviews and is very effective in discovering some types of program faults. However, it cannot discover some classes of errors that could be identified in program inspection meetings.

Static analysis tools (Lopes, Vicente, and Silva 2009) work on the source code of a system, and, for some types of analysis at least, no further inputs are required. This means that programmers do not need to learn specialized notations to write program specifications, so the benefits of analysis can be immediately clear. This makes automated static analysis easier to introduce into a development process than formal verification or model checking.

The intention of automatic static analysis is to draw a code reader's attention to anomalies in the program, such as variables that are used without initialization, variables that are unused, or data whose values could go out of range. Examples of the problems that can be detected by static analysis are shown in Figure 12.9.

Of course, the specific checks made by the static analyzer are programming-language-specific and depend on what is and isn't allowed in the language. Anomalies are often a result of programming errors or omissions, so they highlight things that could go wrong when the program is executed. However, these anomalies are not necessarily program faults; they may be deliberate constructs introduced by the programmer, or the anomaly may have no adverse consequences.

Three levels of checking may be implemented in static analyzers:

1. *Characteristic error checking* At this level, the static analyzer knows about common errors that are made by programmers in languages such as Java or C. The tool analyzes the code looking for patterns that are characteristic of that problem

Fault class	Static analysis check
Data faults	Variables used before initialization Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Nonusage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic Memory leaks

Figure 12.9 Automated static analysis checks

and highlights these to the programmer. Though relatively simple, analysis based on common errors can be very cost-effective. Zheng and his collaborators (Zheng et al. 2006) analyzed a large code base in C and C++. They discovered that 90% of the errors in the programs resulted from 10 types of characteristic error.

2. *User-defined error checking* In this approach, the users of the static analyzer define error patterns to be detected. These may relate to the application domain or may be based on knowledge of the specific system that is being developed. An example of an error pattern is "maintain ordering"; for example, method A must always be called before method B. Over time, an organization can collect information about common bugs that occur in their programs and extend the static analysis tools with error patterns to highlight these errors.

3. *Assertion checking* This is the most general and most powerful approach to static analysis. Developers include formal assertions (often written as stylized comments) in their program that state relationships that must hold at that point in a program. For example, the program might include an assertion stating that the value of some variable must lie in the range x..y. The analyzer symbolically executes the code and highlights statements where the assertion may not hold.

Static analysis is effective in finding errors in programs but, commonly, generates a large number of false positives. These are code sections where there are no errors but where the static analyzer's rules have detected a potential for error. The number of false positives can be reduced by adding more information to the program in the form of assertions, but this requires additional work by the developer of the code. Work has to be done in screening out these false positives before the code itself can be checked for errors.

Many organizations now routinely use static analysis in their software development processes. Microsoft introduced static analysis in the development of device

drivers where program failures can have a serious effect. They extended the approach across a much wider range of their software to look for security problems as well as errors that affect program reliability (Ball, Levin, and Rajamani 2011). Checking for well-known problems, such as buffer overflow, is effective for improving security as attackers often base their attacks on those common vulnerabilities. Attacks may target little-used code sections that may not have been thoroughly tested. Static analysis is a cost-effective way of finding these types of vulnerability.

12.4 Safety cases

As I have discussed, many safety-critical, software-intensive systems are regulated. An external authority has significant influence on their development and deployment. Regulators are government bodies whose job is to ensure that commercial companies do not deploy systems that pose threats to public and environmental safety or the national economy. The owners of safety-critical systems must convince regulators that they have made the best possible efforts to ensure that their systems are safe. The regulator assesses the safety case for the system, which presents evidence and arguments that normal operation of the system will not cause harm to a user.

This evidence is collected during the systems development process. It may include information about hazard analysis and mitigation, test results, static analyses, information about the development processes used, records of review meetings, and so on. It is assembled and organized into a safety case, a detailed presentation of why the system owners and developers believe that a system is safe.

A safety case is a set of documents that includes a description of the system to be certified, information about the processes used to develop the system, and, critically, logical arguments that demonstrate that the system is likely to be safe. More succinctly, Bishop and Bloomfield (Bishop and Bloomfield 1998) define a safety case as:

> *A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment*[†].

The organization and contents of a safety case depend on the type of system that is to be certified and its context of operation. Figure 12.10 shows one possible structure for a safety case, but there are no universal industrial standards in this area. Safety case structures vary, depending on the industry and the maturity of the domain. For example, nuclear safety cases have been required for many years. They are very comprehensive and presented in a way that is familiar to nuclear engineers. However, safety cases for medical devices have been introduced more recently. The case structure is more flexible, and the cases themselves are less detailed than nuclear cases.

A safety case refers to a system as a whole, and, as part of that case, there may be a subsidiary software safety case. When constructing a software safety case, you have to relate software failures to wider system failures and demonstrate either that

[†]Bishop, P., and R. E. Bloomfield. 1998. "A Methodology for Safety Case Development." In Proc. Safety-Critical Systems Symposium. Birmingham, UK: Springer. http://www.adelard.com/papers/sss98web.pdf

Chapter	Description
System description	An overview of the system and a description of its critical components.
Safety requirements	The safety requirements taken from the system requirements specification. Details of other relevant system requirements may also be included.
Hazard and risk analysis	Documents describing the hazards and risks that have been identified and the measures taken to reduce risk. Hazard analyses and hazard logs.
Design analysis	A set of structured arguments (see Section 12.4.1) that justify why the design is safe.
Verification and validation	A description of the V & V procedures used and, where appropriate, the test plans for the system. Summaries of the test results showing defects that have been detected and corrected. If formal methods have been used, a formal system specification and any analyses of that specification. Records of static analyses of the source code.
Review reports	Records of all design and safety reviews.
Team competences	Evidence of the competence of all of the team involved in safety-related systems development and validation.
Process QA	Records of the quality assurance processes (see Chapter 24) carried out during system development.
Change management processes	Records of all changes proposed, actions taken, and, where appropriate, justification of the safety of these changes. Information about configuration management procedures and configuration management logs.
Associated safety cases	References to other safety cases that may impact the safety case.

Figure 12.10 Possible contents of a software safety case

these software failures will not occur or that they will not be propagated in such a way that dangerous system failures may occur.

Safety cases are large and complex documents, and so they are very expensive to produce and maintain. Because of these high costs, safety-critical system developers have to take the requirements of the safety case into account in the development process:

1. Graydon et al. (Graydon, Knight, and Strunk 2007) argue that the development of a safety case should be tightly integrated with system design and implementation. This means that system design decisions may be influenced by the requirements of the safety case. Design choices that may add significantly to the difficulties and costs of case development can then be avoided.

2. Regulators have their own views on what is acceptable and unacceptable in a safety case. It therefore makes sense for a development team to work with them from early in the development to establish what the regulator expects from the system safety case.

The development of safety cases is expensive because of the costs of the record keeping required as well as the costs of comprehensive system validation and safety assurance processes. System changes and rework also add to the costs of a safety

Figure 12.11 Structured arguments

case. When software or hardware changes are made to a system, a large part of the safety case may have to be rewritten to demonstrate that the system safety has not been affected by the change.

12.4.1 Structured arguments

The decision on whether or not a system is operationally safe should be based on logical arguments. These arguments should demonstrate that the evidence presented supports the claims about a system's security and dependability. These claims may be absolute (event X will or will not happen) or probabilistic (the probability of occurrence of event Y is 0.n). An argument links the evidence and the claim. As shown in Figure 12.11, an argument is a relationship between what is thought to be the case (the claim) and a body of evidence that has been collected. The argument essentially explains why the claim, which is an assertion about system security or dependability, can be inferred from the available evidence.

Arguments in a safety case are usually presented as "claim based" arguments. Some claim about system safety is made, and, on the basis of available evidence, an argument is presented as to why that claim holds. For example, the following argument might be used to justify a claim that computations carried out by the control software in an insulin pump will not lead to an overdose of insulin being delivered. Of course, this is a very simplified presentation of the argument. In a real safety case, more detailed references to the evidence would be presented.

Claim: The maximum single dose computed by the insulin pump will not exceed maxDose, where maxDose has been assessed as a safe single dose for a particular patient.

Evidence: Safety argument for insulin pump software control program (covered later in this section).

Evidence: Test datasets for the insulin pump. In 400 tests, which provided complete code coverage, the value of the dose of insulin to be delivered, currentDose, never exceeded maxDose.

Figure 12.12 A safety claim hierarchy for the insulin pump

Evidence: A static analysis report for the insulin pump control program. The static analysis of the control software revealed no anomalies that affected the value of **currentDose**, the program variable that holds the dose of insulin to be delivered.

Argument: The evidence presented demonstrates that the maximum dose of insulin that can be computed is equal to **maxDose**.

It is therefore reasonable to assume, with a high level of confidence, that the evidence justifies the claim that the insulin pump will not compute a dose of insulin to be delivered that exceeds the maximum single safe dose.

The evidence presented is both redundant and diverse. The software is checked using several different mechanisms with significant overlap between them. As I discussed in Chapter 10, using redundant and diverse processes increases confidence. If omissions and mistakes are not detected by one validation process, there is a good chance that they will be found by one of the other processes.

There will normally be many claims about the safety of a system, with the validity of one claim often depending on whether or not other claims are valid. Therefore, claims may be organized in a hierarchy. Figure 12.12 shows part of this claim hierarchy for the insulin pump. To demonstrate that a high-level claim is valid, you first have to work through the arguments for lower-level claims. If you can show that each of these lower-level claims is justified, then you may be able to infer that the higher-level claims are justified.

12.4.2 Software safety arguments

A general assumption that underlies work in system safety is that the number of system faults that can lead to safety hazards is significantly less than the total number of faults that may exist in the system. Safety assurance can therefore concentrate on

these faults, which have hazard potential. If it can be demonstrated that these faults cannot occur or, if they occur, that the associated hazard will not result in an accident, then the system is safe. This is the basis of software safety arguments.

Software safety arguments are a type of structured argument which demonstrates that a program meets its safety obligations. In a safety argument, it is not necessary to prove that the program works as intended. It is only necessary to show that program execution cannot result in it reaching a potentially unsafe state. Safety arguments are therefore cheaper to make than correctness arguments. You don't have to consider all program states—you can simply concentrate on states that could lead to a hazard.

Safety arguments demonstrate that, assuming normal execution conditions, a program should be safe. They are usually based on contradiction, where you assume that the system is unsafe and then show that it is impossible to reach an unsafe state. The steps involved in creating a safety argument are:

1. You start by assuming that an unsafe state, which has been identified by the system hazard analysis, can be reached by executing the program.
2. You write a predicate (a logical expression) that defines this unsafe state.
3. You then systematically analyze a system model or the program and show that, for all program paths leading to that state, the terminating condition of these paths, also defined as a predicate, contradicts the unsafe state predicate. If this is the case, you may then claim that the initial assumption of an unsafe state is incorrect.
4. When you have repeated this analysis for all identified hazards, then you have strong evidence that the system is safe.

Safety arguments can be applied at different levels, from requirements through design models to code. At the requirements level, you are trying to demonstrate that there are no missing safety requirements and that the requirements do not make invalid assumptions about the system. At the design level, you might analyze a state model of the system to find unsafe states. At the code level, you consider all of the paths through the safety-critical code to show that the execution of all paths leads to a contradiction.

As an example, consider the code outlined in Figure 12.13, which is a simplified description of part of the implementation of the insulin delivery system. The code computes the dose of insulin to be delivered and then applies some safety checks that this is not an overdose for that patient. Developing a safety argument for this code involves demonstrating that the dose of insulin administered is never greater than the maximum safe level for a single dose. This dose is established for each individual diabetic user in discussions with their medical advisors.

To demonstrate safety, you do not have to prove that the system delivers the "correct" dose, but merely that it never delivers an overdose to the patient. You work on the assumption that `maxDose` is the safe level for that system user.

To construct the safety argument, you identify the predicate that defines the unsafe state, which is that `currentDose > maxDose`. You then demonstrate that all program paths lead to a contradiction of this unsafe assertion. If this is the case, the unsafe condition cannot be true. If you can prove a contradiction, you can be confident that

```
— The insulin dose to be delivered is a function of
— blood sugar level, the previous dose delivered and
— the time of delivery of the previous dose

currentDose = computeInsulin () ;

// Safety check—adjust currentDose if necessary.

// if statement 1

if (previousDose == 0)
{
   if (currentDose > maxDose/2)
      currentDose = maxDose/2 ;
}
else
   if (currentDose > (previousDose * 2) )
      currentDose = previousDose * 2 ;

// if statement 2

if ( currentDose < minimumDose )
   currentDose = 0 ;
else if ( currentDose > maxDose )
   currentDose = maxDose ;
administerInsulin (currentDose) ;
```

Figure 12.13 Insulin dose computation with safety checks

the program will not compute an unsafe dose of insulin. You can structure and present the safety arguments graphically as shown in Figure 12.14.

The safety argument shown in Figure 12.14 presents three possible program paths that lead to the call to the **administerInsulin** method. You have to show that the amount of insulin delivered never exceeds **maxDose**. All possible program paths to **administerInsulin** are considered:

1. Neither branch of if-statement 2 is executed. This can only happen if **currentDose** is outside of the range **minimumDose..maxDose**. The postcondition predicate is therefore:

   currentDose >= minimumDose and currentDose <= maxDose

2. The then-branch of if-statement 2 is executed. In this case, the assignment setting **currentDose** to zero is executed. Therefore, its postcondition predicate is **currentDose = 0**.

3. The else-if-branch of if-statement 2 is executed. In this case, the assignment setting **currentDose to maxDose** is executed. Therefore, after this statement has been executed, we know that the postcondition is **currentDose = maxDose**.

In all three cases, the postcondition predicates contradict the unsafe precondition that **currentDose > maxDose**. As both cannot be true, we can claim that our initial assumption was incorrect, and so the computation is safe.

To construct a structured argument that a program does not make an unsafe computation, you first identify all possible paths through the code that could lead to a potentially

Figure 12.14 Informal safety argument based on demonstrating contradictions

unsafe assignment. You work backwards from the unsafe state and consider the last assignment to all of the state variables on each path leading to this unsafe state. If you can show that none of the values of these variables is unsafe, then you have shown that your initial assumption (that the computation is unsafe) is incorrect.

Working backwards is important because it means that you can ignore all intermediate states apart from the final states that lead to the exit condition for the code. The previous values don't matter to the safety of the system. In this example, all you need be concerned with is the set of possible values of currentDose immediately before the administerInsulin method is executed. You can ignore computations, such as if-statement 1 in Figure 12.13 in the safety argument because their results are overwritten in later program statements.

KEY POINTS

- Safety-critical systems are systems whose failure can lead to human injury or death.
- A hazard-driven approach may be used to understand the safety requirements for safety-critical systems. You identify potential hazards and decompose them (using methods such as fault tree analysis) to discover their root causes. You then specify requirements to avoid or recover from these problems.
- It is important to have a well-defined, certified process for safety-critical systems development. The process should include the identification and monitoring of potential hazards.
- Static analysis is an approach to V & V that examines the source code (or other representation) of a system, looking for errors and anomalies. It allows all parts of a program to be checked, not just those parts that are exercised by system tests.
- Model checking is a formal approach to static analysis that exhaustively checks all states in a system for potential errors.
- Safety and dependability cases collect all of the evidence that demonstrates a system is safe and dependable. Safety cases are required when an external regulator must certify the system before it is used.

FURTHER READING

Safeware: System Safety and Computers. Although now 20 years old, this book still offers the best and most thorough coverage of safety-critical systems. It is particularly strong in its description of hazard analysis and the derivation of requirements from it. (N. Leveson, Addison-Wesley, 1995).

"Safety-Critical Software." A special edition of *IEEE Software* magazine that focuses on safety-critical systems. It includes papers on model-based development of safety-critical systems, model checking and formal methods. (*IEEE Software*, 30 (3), May/June 2013).

"Constructing Safety Assurance Cases for Medical Devices." This short paper gives a practical example of how a safety case can be created for an analgesic pump. (A. Ray and R. Cleaveland, Proc. Workshop on Assurance Cases for Software-Intensive Systems, San Francisco, 2013) http://dx.doi.org/10.1109/ASSURE.2013.6614270

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/reliability-and-safety/

EXERCISES

12.1. Identify six consumer products that are likely to be controlled by safety-critical software systems.

12.2. A software system is to be deployed for a company that has extremely high safety standards and allows for almost no risks, not even minor injuries. How will this affect the look of the risk triangle in Figure 12.3?

12.3. In the insulin pump system, the user has to change the needle and insulin supply at regular intervals and may also change the maximum single dose and the maximum daily dose that may be administered. Suggest three user errors that might occur and propose safety requirements that would avoid these errors resulting in an accident.

12.4. A safety-critical software system for managing roller coasters controls two main components:

- The lock and release of the roller coaster harness which is supposed to keep riders in place as the coaster performs sharp and sudden moves. The roller coaster could not move with any unlocked harnesses.
- The minimum and maximum speeds of the roller coaster as it moves along the various segments of the ride to prevent derailing, given the number of people riding the roller coaster.

Identify three hazards that may arise in this system. For each hazard, suggest a defensive requirement that will reduce the probability that these hazards will result in an accident. Explain why your suggested defense is likely to reduce the risk associated with the hazard.

12.5. A train protection system automatically applies the brakes of a train if the speed limit for a segment of track is exceeded, or if the train enters a track segment that is currently signaled with a red light (i.e., the segment should not be entered). There are two critical-safety requirements for this train protection system:

The train shall not enter a segment of track that is signaled with a red light.

The train shall not exceed the specified speed limit for a section of track.

Assuming that the signal status and the speed limit for the track segment are transmitted to on-board software on the train before it enters the track segment, propose five possible functional system requirements for the onboard software that may be generated from the system safety requirements.

12.6. Explain when it may be cost-effective to use formal specification and verification in the development of safety-critical software systems. Why do you think that some critical systems engineers are against the use of formal methods?

12.7. Explain why using model checking is sometimes a more cost-effective approach to verification than verifying a program's correctness against a formal specification.

12.8. List four types of systems that may require software safety cases, explaining why safety cases are required.

12.9. The door lock control mechanism in a nuclear waste storage facility is designed for safe operation. It ensures that entry to the storeroom is only permitted when radiation shields are

```
1       entryCode = lock.getEntryCode () ;
2       if (entryCode == lock.authorizedCode)
3       {
4               shieldStatus = Shield.getStatus ();
5               radiationLevel = RadSensor.get ();
6               if (radiationLevel < dangerLevel)
7                       state = safe;
8               else
9                       state = unsafe;
10              if (shieldStatus == Shield.inPlace() )
11                      state = safe;
12              if (state == safe)
13                      {
14                              Door.locked = false ;
15                              Door.unlock ();
16                      }
17              else
18              {
19                      Door.lock ( );
20                      Door.locked := true ;
21              }
22      }
```

Figure 12.15 Door entry code

in place or when the radiation level in the room falls below some given value (dangerLevel). So:

(i) If remotely controlled radiation shields are in place within a room, an authorized operator may open the door.

(ii) If the radiation level in a room is below a specified value, an authorized operator may open the door.

(iii) An authorized operator is identified by the input of an authorized door entry code.

The code shown in Figure 12.15 controls the door-locking mechanism. Note that the safe state is that entry should not be permitted. Using the approach discussed in this chapter, develop a safety argument for this code. Use the line numbers to refer to specific statements. If you find that the code is unsafe, suggest how it should be modified to make it safe.

12.10. Should software engineers working on the specification and development of safety-related systems be professionally certified or licensed in some way? Explain your reasoning.

REFERENCES

Abrial, J. R. 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.

Ball, T., V. Levin, and S. K. Rajamani. 2011. “A Decade of Software Model Checking with SLAM.” *Communications of the ACM* 54 (7) (July 1): 68. doi:10.1145/1965724.1965743.

Behm, P., P. Benoit, A. Faivre, and J-M. Meynadier. 1999. "Meteor: A Successful Application of B in a Large Project." In *Formal Methods' 99*, 369–387. Berlin: Springer-Verlag. doi:10.1007/3-540-48119-2_22.

Bishop, P., and R. E. Bloomfield. 1998. "A Methodology for Safety Case Development." In *Proc. Safety-Critical Systems Symposium*. Birmingham, UK: Springer. http://www.adelard.com/papers/sss98web.pdf

Bochot, T., P. Virelizier, H. Waeselynck, and V. Wiels. 2009. "Model Checking Flight Control Systems: The Airbus Experience." In *Proc. 31st International Conf. on Software Engineering, Companion Volume*, 18–27. Leipzig: IEEE Computer Society Press. doi:10.1109/ICSE-COMPANION.2009.5070960.

Dehbonei, B., and F. Mejia. 1995. "Formal Development of Safety-Critical Software Systems in Railway Signalling." In *Applications of Formal Methods*, edited by M. Hinchey and J. P. Bowen, 227–252. London: Prentice-Hall.

Graydon, P. J., J. C. Knight, and E. A. Strunk. 2007. "Assurance Based Development of Critical Systems." In *Proc. 37th Annual IEEE Conf. on Dependable Systems and Networks*, 347–357. Edinburgh, Scotland. doi:10.1109/DSN.2007.17.

Holzmann, G. J. 2014. "Mars Code." *Comm ACM* 57 (2): 64–73. doi:10.1145/2560217.2560218.

Jhala, R., and R. Majumdar. 2009. "Software Model Checking." *Computing Surveys* 41 (4). doi:10.1145/1592434.1592438.

Kwiatkowska, M., G. Norman, and D. Parker. 2011. "PRISM 4.0: Verification of Probabilistic Real-Time Systems." In *Proc. 23rd Int. Conf. on Computer Aided Verification*, 585–591. Snowbird, UT: Springer-Verlag. doi:10.1007/978-3-642-22110-1_47.

Leveson, N. G., S. S. Cha, and T. J. Shimeall. 1991. "Safety Verification of Ada Programs Using Software Fault Trees." *IEEE Software* 8 (4): 48–59. doi:10.1109/52.300036.

Lopes, R., D. Vicente, and N. Silva. 2009. "Static Analysis Tools, a Practical Approach for Safety-Critical Software Verification." In *Proceedings of DASIA 2009 Data Systems in Aerospace*. Noordwijk, Netherlands: European Space Agency.

Lutz, R. R. 1993. "Analysing Software Requirements Errors in Safety-Critical Embedded Systems." In *RE' 93*, 126–133. San Diego, CA: IEEE. doi:0.1109/ISRE.1993.324825.

Moy, Y., E. Ledinot, H. Delseny, V. Wiels, and B. Monate. 2013. "Testing or Formal Verification: DO-178C Alternatives and Industrial Experience." *IEEE Software* 30 (3) (May 1): 50–57. doi:10.1109/MS.2013.43.

Perrow, C. 1984. *Normal Accidents: Living with High-Risk Technology*. New York: Basic Books.

Regan, P., and S. Hamilton. 2004. "NASA's Mission Reliable." *IEEE Computer* 37 (1): 59–68. doi:10.1109/MC.2004.1260727.

Schneider, S. 1999. *Concurrent and Real-Time Systems: The CSP Approach*. Chichester, UK: John Wiley & Sons.

Souyris, J., V. Weils, D. Delmas, and H. Delseny. 2009. "Formal Verification of Avionics Software Products." In *Formal Methods' 09: Proceedings of the 2nd World Congress on Formal Methods*, 532–546. Springer-Verlag. doi:10.1007/978-3-642-05089-3_34.

Storey, N. 1996. *Safety-Critical Computer Systems*. Harlow, UK: Addison-Wesley.

Veras, P. C., E. Villani, A. M. Ambrosio, N. Silva, M. Vieira, and H. Madeira. 2010. "Errors in Space Software Requirements: A Field Study and Application Scenarios." In *21st Int. Symp. on Software Reliability Engineering*. San Jose, CA. doi:10.1109/ISSRE.2010.37.

Zheng, J., L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. 2006. "On the Value of Static Analysis for Fault Detection in Software." *IEEE Trans. on Software Eng*. 32 (4): 240–253. doi:10.1109/TSE.2006.38.

13

Security engineering

Objectives

The objective of this chapter is to introduce security issues that you should consider when you are developing application systems. When you have read this chapter, you will:

- understand the importance of security engineering and the difference between application security and infrastructure security;
- know how a risk-based approach can be used to derive security requirements and analyze system designs;
- know of software architectural patterns and design guidelines for secure systems engineering;
- understand why security testing and assurance is difficult and expensive.

Contents

13.1 Security and dependability
13.2 Security and organizations
13.3 Security requirements
13.4 Secure systems design
13.5 Security testing and assurance

The widespread adoption of the Internet in the 1990s introduced a new challenge for software engineers—designing and implementing systems that were secure. As more and more systems were connected to the Internet, a variety of different external attacks were devised to threaten these systems. The problems of producing dependable systems were hugely increased. Systems engineers had to consider threats from malicious and technically skilled attackers as well as problems resulting from accidental mistakes in the development process.

It is now essential to design systems to withstand external attacks and to recover from such attacks. Without security precautions, attackers will inevitably compromise a networked system. They may misuse the system hardware, steal confidential data, or disrupt the services offered by the system.

You have to take three security dimensions into account in secure systems engineering:

1. *Confidentiality* Information in a system may be disclosed or made accessible to people or programs that are not authorized to have access to that information. For example, the theft of credit card data from an e-commerce system is a confidentiality problem.
2. *Integrity* Information in a system may be damaged or corrupted, making it unusual or unreliable. For example, a worm that deletes data in a system is an integrity problem.
3. *Availability* Access to a system or its data that is normally available may not be possible. A denial-of-service attack that overloads a server is an example of a situation where the system availability is compromised.

These dimensions are closely related. If an attack makes the system unavailable, then you will not be able to update information that changes with time. This means that the integrity of the system may be compromised. If an attack succeeds and the integrity of the system is compromised, then it may have to be taken down to repair the problem. Therefore, the availability of the system is reduced.

From an organizational perspective, security has to be considered at three levels:

1. *Infrastructure security,* which is concerned with maintaining the security of all systems and networks that provide an infrastructure and a set of shared services to the organization.
2. *Application security,* which is concerned with the security of individual application systems or related groups of systems.
3. *Operational security,* which is concerned with the secure operation and use of the organization's systems.

Figure 13.1 is a diagram of an application system stack that shows how an application system relies on an infrastructure of other systems in its operation. The lower levels of the infrastructure are hardware, but the software infrastructure for application systems may include:

Figure 13.1 System layers where security may be compromised

- an operating system platform, such as Linux or Windows;
- other generic applications that run on that system, such as web browsers and email clients;
- a database management system;
- middleware that supports distributed computing and database access; and
- libraries of reusable components that are used by the application software.

Network systems are software controlled, and networks may be subject to security threats where an attacker intercepts and reads or changes network packets. However, this requires specialized equipment, so the majority of security attacks are on the software infrastructure of systems. Attackers focus on software infrastructures because infrastructure components, such as web browsers, are universally available. Attackers can probe these systems for weaknesses and share information about vulnerabilities that they have discovered. As many people use the same software, attacks have wide applicability.

Infrastructure security is primarily a system management problem, where system managers configure the infrastructure to resist attacks. System security management includes a range of activities such as user and permission management, system software deployment and maintenance, and attack monitoring, detection, and recovery:

1. User and permission management involves adding and removing users from the system, ensuring that appropriate user authentication mechanisms are in place, and setting up the permissions in the system so that users only have access to the resources they need.
2. System software deployment and maintenance involves installing system software and middleware and configuring these properly so that security vulnerabilities are avoided. It also involves updating this software regularly with new versions or patches, which repair security problems that have been discovered.

3. Attack monitoring, detection, and recovery involves monitoring the system for unauthorized access, detecting and putting in place strategies for resisting attacks, and organizing backups of programs and data so that normal operation can be resumed after an external attack.

Operational security is primarily a human and social issue. It focuses on ensuring that the people using the system do not behave in such a way that system security is compromised. For example, users may leave themselves logged on to a system while it is unattended. An attacker can then easily get access to the system. Users often behave in an insecure way to help them do their jobs more effectively, and they have good reason to behave in an insecure way. A challenge for operational security is to raise awareness of security issues and to find the right balance between security and system effectiveness.

The term *cybersecurity* is now commonly used in discussions of system security. Cybersecurity is a very wide-ranging term that covers all aspects of the protection of citizens, businesses, and critical infrastructures from threats that arise from their use of computers and the Internet. Its scope includes all system levels from hardware and networks through application systems to mobile devices that may be used to access these systems. I discuss general cybersecurity issues, including infrastructure security, in Chapter 14, which covers resilience engineering.

In this chapter, I focus on issues of application security engineering—security requirements, design for security, and security testing. I don't cover general security techniques that may be used, such as encryption, and access control mechanisms or attack vectors, such as viruses and worms. General textbooks on computer security (Pfleeger and Pfleeger 2007; Anderson 2008; Stallings and Brown 2012) discuss these techniques in detail.

13.1 Security and dependability

Security is a system attribute that reflects the ability of the system to protect itself from malicious internal or external attacks. These external attacks are possible because most computers and mobile devices are networked and are therefore accessible by outsiders. Examples of attacks might be the installation of viruses and Trojan horses, unauthorized use of system services, or unauthorized modification of a system or its data.

If you really want a system to be as secure as possible, it is best not to connect it to the Internet. Then, your security problems are limited to ensuring that authorized users do not abuse the system and to controlling the use of devices such as USB drives. In practice, however, networked access provides huge benefits for most systems, so disconnecting from the Internet is not a viable security option.

For some systems, security is the most important system dependability attribute. Military systems, systems for electronic commerce, and systems that involve the processing and interchange of confidential information must be designed so that

Term	Definition
Asset	Something of value that has to be protected. The asset may be the software system itself or the data used by that system.
Attack	An exploitation of a system's vulnerability where an attacker has the goal of causing some damage to a system asset or assets. Attacks may be from outside the system (external attacks) or from authorized insiders (insider attacks).
Control	A protective measure that reduces a system's vulnerability. Encryption is an example of a control that reduces a vulnerability of a weak access control system.
Exposure	Possible loss or harm to a computing system. This can be loss or damage to data or can be a loss of time and effort if recovery is necessary after a security breach.
Threat	Circumstances that have potential to cause loss or harm. You can think of a threat as a system vulnerability that is subjected to an attack.
Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm.

Figure 13.2 Security terminology

Unauthorized access to the Mentcare system

Clinic staff log on to the Mentcare system using a username and password. The system requires passwords to be at least eight letters long but allows any password to be set without further checking. A criminal finds out that a well-paid sports star is receiving treatment for mental health problems. He would like to gain illegal access to information in this system so that he can blackmail the star.

By posing as a concerned relative and talking with the nurses in the mental health clinic, he discovers how to access the system and personal information about the nurses and their families. By checking name badges, he discovers the names of some of the people allowed access. He then attempts to log on to the system by using these names and systematically guessing possible passwords, such as the names of the nurses' children.

Figure 13.3 A security story for the Mentcare system

they achieve a high level of security. If an airline reservation system is unavailable, for example, this causes inconvenience and some delays in issuing tickets. However, if the system is insecure, then an attacker could delete all bookings and it would be practically impossible for normal airline operations to continue.

As with other aspects of dependability, a specialized terminology is associated with security (Pfleeger and Pfleeger 2007). This terminology is explained in Figure 13.2. Figure 13.3 is a security story from the Mentcare system that I use to illustrate some of these terms. Figure 13.4 takes the security concepts defined in Figure 13.2 and shows how they apply to this security story.

System vulnerabilities may arise because of requirements, design, or implementation problems, or they may stem from human, social, or organizational failings. People may choose easy-to-guess passwords or write down their passwords in places where they can be found. System administrators make errors in setting up access control or configuration files, and users don't install or use protection software. However, we cannot simply class these problems as human errors. User mistakes or omissions often reflect poor systems design decisions that require, for example, frequent password changes (so that users write down their passwords) or complex configuration mechanisms.

Term	Example
Asset	The record of each patient who is receiving or has received treatment.
Attack	An impersonation of an authorized user.
Control	A password checking system that disallows user passwords that are proper names or words that are normally included in a dictionary.
Exposure	Potential financial loss from future patients who do not seek treatment because they do not trust the clinic to maintain their data. Financial loss from legal action by the sports star. Loss of reputation.
Threat	An unauthorized user will gain access to the system by guessing the credentials (login name and password) of an authorized user.
Vulnerability	Authentication is based on a password system that does not require strong passwords. Users can then set easily guessable passwords.

Figure 13.4 Examples of security terminology

Four types of security threats may arise:

1. Interception threats that allow an attacker to gain access to an asset. So, a possible threat to the Mentcare system might be a situation where an attacker gains access to the records of an individual patient.
2. Interruption threats that allow an attacker to make part of the system unavailable. Therefore, a possible threat might be a denial-of-service attack on a system database server.
3. Modification threats that allow an attacker to tamper with a system asset. In the Mentcare system, a modification threat would be where an attacker alters or destroys a patient record.
4. Fabrication threats that allow an attacker to insert false information into a system. This is perhaps not a credible threat in the Mentcare system but would certainly be a threat in a banking system, where false transactions might be added to the system that transfers money to the perpetrator's bank account.

The controls that you might put in place to enhance system security are based on the fundamental notions of avoidance, detection, and recovery:

1. *Vulnerability avoidance* Controls that are intended to ensure that attacks are unsuccessful. The strategy here is to design the system so that security problems are avoided. For example, sensitive military systems are not connected to the Internet so that external access is more difficult. You should also think of encryption as a control based on avoidance. Any unauthorized access to encrypted data means that the attacker cannot read the encrypted data. It is expensive and time consuming to crack strong encryption.
2. *Attack detection and neutralization* Controls that are intended to detect and repel attacks. These controls involve including functionality in a system that monitors its operation and checks for unusual patterns of activity. If these

attacks are detected, then action may be taken, such as shutting down parts of the system or restricting access to certain users.

3. *Exposure limitation and recovery* Controls that support recovery from problems. These can range from automated backup strategies and information "mirroring" through to insurance policies that cover the costs associated with a successful attack on the system.

Security is closely related to the other dependability attributes of reliability, availability, safety, and resilience:

1. *Security and reliability* If a system is attacked and the system or its data are corrupted as a consequence of that attack, then this may induce system failures that compromise the reliability of the system.

   Errors in the development of a system can lead to security loopholes. If a system does not reject unexpected inputs or if array bounds are not checked, then attackers can exploit these weaknesses to gain access to the system. For example, failure to check the validity of an input may mean that an attacker can inject and execute malicious code.

2. *Security and availability* A common attack on a web-based system is a denial-of-service attack, where a web server is flooded with service requests from a range of different sources. The aim of this attack is to make the system unavailable. A variant of this attack is where a profitable site is threatened with this type of attack unless a ransom is paid to the attackers.

3. *Security and safety* Again, the key problem is an attack that corrupts the system or its data. Safety checks are based on the assumption that we can analyze the source code of safety-critical software and that the executing code is a completely accurate translation of that source code. If this is not the case, because an attacker has changed the executing code, safety-related failures may be induced and the safety case made for the software is invalid.

   Like safety, we cannot assign a numeric value to the security of a system, nor can we exhaustively test the system for security. Both safety and security can be thought of as "negative" or "shall not" characteristics in that they are concerned with things that should not happen. As we can never prove a negative, we can never prove that a system is safe or secure.

4. *Security and resilience* Resilience, covered in Chapter 14, is a system characteristic that reflects its ability to resist and recover from damaging events. The most probable damaging event on networked software systems is a cyberattack of some kind, so most of the work now done in resilience is aimed at deterring, detecting, and recovering from such attacks.

Security has to be maintained if we are to create reliable, available, and safe software-intensive systems. It is not an add-on, which can be added later but has to be considered at all stages of the development life cycle from early requirements to system operation.

13.2 Security and organizations

Building secure systems is expensive and uncertain. It is impossible to predict the costs of a security failure, so companies and other organizations find it difficult to judge how much they should spend on system security. In this respect, security and safety are different. There are laws that govern workplace and operator safety, and developers of safety-critical systems have to comply with these irrespective of the costs. They may be subject to legal action if they use an unsafe system. However, unless a security failure discloses personal information, there are no laws that prevent an insecure system from being deployed.

Companies assess the risks and losses that may arise from certain types of attacks on system assets. They may then decide that it is cheaper to accept these risks rather than build a secure system that can deter or repel the external attacks. Credit card companies apply this approach to fraud prevention. It is usually possible to introduce new technology to reduce credit card fraud. However, it is often cheaper for these companies to compensate users for their losses due to fraud than to buy and deploy fraud-reduction technology.

Security risk management is therefore a business rather than a technical issue. It has to take into account the financial and reputational losses from a successful system attack as well as the costs of security procedures and technologies that may reduce these losses. For risk management to be effective, organizations should have a documented information security policy that sets out:

1. *The assets that must be protected* It does not necessarily make sense to apply stringent security procedures to all organizational assets. Many assets are not confidential, and a company can improve its image by making these assets freely available. The costs of maintaining the security of information that is in the public domain are much less than the costs of keeping confidential information secure.

2. *The level of protection that is required for different types of assets* Not all assets need the same level of protection. In some cases (e.g., for sensitive personal information), a high level of security is required; for other information, the consequences of loss may be minor, so a lower level of security is adequate. Therefore, some information may be made available to any authorized and logged-in user; other information may be much more sensitive and only available to users in certain roles or positions of responsibility.

3. *The responsibilities of individual users, managers, and the organization* The security policy should set out what is expected of users—for example, use strong passwords, log out of computers, and lock offices. It also defines what users can expect from the company, such as backup and information-archiving services, and equipment provision.

4. *Existing security procedures and technologies that should be maintained* For reasons of practicality and cost, it may be essential to continue to use existing approaches to security even where these have known limitations. For example,

a company may require the use of a login name/password for authentication, simply because other approaches are likely to be rejected by users.

Security policies often set out general information access strategies that should apply across the organization. For example, an access strategy may be based on the clearance or seniority of the person accessing the information. Therefore, a military security policy may state: "Readers may only examine documents whose classification is the same as or below the reader's vetting level." This means that if a reader has been vetted to a "secret" level, he or she may access documents that are classed as secret, confidential, or open but not documents classed as top secret.

The point of security policies is to inform everyone in an organization about security, so these should not be long and detailed technical documents. From a security engineering perspective, the security policy defines, in broad terms, the security goals of the organization. The security engineering process is concerned with implementing these goals.

13.2.1 Security risk assessment

Security risk assessment and management are organizational activities that focus on identifying and understanding the risks to information assets (systems and data) in the organization. In principle, an individual risk assessment should be carried out for all assets; in practice, however, this may be impractical if a large number of existing systems and databases need to be assessed. In those situations, a generic assessment may be applied to all of them. However, individual risk assessments should be carried out for new systems.

Risk assessment and management is an organizational activity rather than a technical activity that is part of the software development life cycle. The reason for this is that some types of attack are not technology-based but rather rely on weaknesses in more general organizational security. For example, an attacker may gain access to equipment by pretending to be an accredited engineer. If an organization has a process to check with the equipment supplier that an engineer's visit is planned, this can deter this type of attack. This approach is much simpler than trying to address the problem using a technological solution.

When a new system is to be developed, security risk assessment and management should be a continuing process throughout the development life cycle from initial specification to operational use. The stages of risk assessment are:

1. *Preliminary risk assessment* The aim of this initial risk assessment is to identify generic risks that are applicable to the system and to decide if an adequate level of security can be achieved at a reasonable cost. At this stage, decisions on the detailed system requirements, the system design, or the implementation technology have not been made. You don't know of potential technology vulnerabilities or the controls that are included in reused system components or middleware. The risk assessment should therefore focus on the identification and analysis of high-level risks to the system. The outcomes of the risk assessment process are used to help identify security requirements.

2. *Design risk assessment* This risk assessment takes place during the system development life cycle and is informed by the technical system design and implementation decisions. The results of the assessment may lead to changes to the security requirements and the addition of new requirements. Known and potential vulnerabilities are identified, and this knowledge is used to inform decision making about the system functionality and how it is to be implemented, tested, and deployed.

3. *Operational risk assessment* This risk assessment process focuses on the use of the system and the possible risks that can arise. For example, when a system is used in an environment where interruptions are common, a security risk is that a logged-in user leaves his or her computer unattended to deal with a problem. To counter this risk, a timeout requirement may be specified so that a user is automatically logged out after a period of inactivity.

   Operational risk assessment should continue after a system has been installed to take account of how the system is used and proposals for new and changed requirements. Assumptions about the operating requirement made when the system was specified may be incorrect. Organizational changes may mean that the system is used in different ways from those originally planned. These changes lead to new security requirements that have to be implemented as the system evolves.

13.3 Security requirements

The specification of security requirements for systems has much in common with the specification of safety requirements. You cannot specify safety or security requirements as probabilities. Like safety requirements, security requirements are often "shall not" requirements that define unacceptable system behavior rather than required system functionality.

However, security is a more challenging problem than safety, for a number of reasons:

1. When considering safety, you can assume that the environment in which the system is installed is not hostile. No one is trying to cause a safety-related incident. When considering security, you have to assume that attacks on the system are deliberate and that the attacker may have knowledge of system weaknesses.

2. When system failures occur that pose a risk to safety, you look for the errors or omissions that have caused the failure. When deliberate attacks cause system failure, finding the root cause may be more difficult as the attacker may try to conceal the cause of the failure.

3. It is usually acceptable to shut down a system or to degrade system services to avoid a safety-related failure. However, attacks on a system may be denial-of-service attacks, which are intended to compromise system availability. Shutting down the system means that the attack has been successful.

4. Safety-related events are accidental and are not created by an intelligent adversary. An attacker can probe a system's defenses in a series of attacks, modifying the attacks as he or she learns more about the system and its responses.

These distinctions mean that security requirements have to be more extensive than safety requirements. Safety requirements lead to the generation of functional system requirements that provide protection against events and faults that could cause safety-related failures. These requirements are mostly concerned with checking for problems and taking actions if these problems occur. By contrast, many types of security requirements cover the different threats faced by a system.

Firesmith (Firesmith 2003) identified 10 types of security requirements that may be included in a system specification:

1. Identification requirements specify whether or not a system should identify its users before interacting with them.
2. Authentication requirements specify how users are identified.
3. Authorization requirements specify the privileges and access permissions of identified users.
4. Immunity requirements specify how a system should protect itself against viruses, worms, and similar threats.
5. Integrity requirements specify how data corruption can be avoided.
6. Intrusion detection requirements specify what mechanisms should be used to detect attacks on the system.
7. Nonrepudiation requirements specify that a party in a transaction cannot deny its involvement in that transaction.
8. Privacy requirements specify how data privacy is to be maintained.
9. Security auditing requirements specify how system use can be audited and checked.
10. System maintenance security requirements specify how an application can prevent authorized changes from accidentally defeating its security mechanisms.

Of course, you will not see all of these types of security requirements in every system. The particular requirements depend on the type of system, the situation of use, and the expected users.

Preliminary risk assessment and analysis aim to identify the generic security risks for a system and its associated data. This risk assessment is an important input to the security requirements engineering process. Security requirements can be proposed to support the general risk management strategies of avoidance, detection and mitigation.

1. Risk avoidance requirements set out the risks that should be avoided by designing the system so that these risks simply cannot arise.

Figure 13.5 The preliminary risk assessment process for security requirements

2. Risk detection requirements define mechanisms that identify the risk if it arises and neutralize the risk before losses occur.
3. Risk mitigation requirements set out how the system should be designed so that it can recover from and restore system assets after some loss has occurred.

A risk-driven security requirements process is shown in Figure 13.5. The process stages are:

1. *Asset identification,* where the system assets that may require protection are identified. The system itself or particular system functions may be identified as assets as well as the data associated with the system.
2. *Asset value assessment,* where you estimate the value of the identified assets.
3. *Exposure assessment,* where you assess the potential losses associated with each asset. This process should take into account direct losses such as the theft of information, the costs of recovery, and the possible loss of reputation.
4. *Threat identification,* where you identify the threats to system assets.
5. *Attack assessment,* where you decompose each threat into attacks that might be made on the system and the possible ways in which these attacks may occur. You may use attack trees (Schneier 1999) to analyze the possible attacks. These are similar to fault trees, (Chapter 12) as you start with a threat at the root of the tree and then identify possible causal attacks and how these might be made.
6. *Control identification,* where you propose the controls that might be put in place to protect an asset. The controls are the technical mechanisms, such as encryption, that you can use to protect assets.
7. *Feasibility assessment,* where you assess the technical feasibility and the costs of the proposed controls. It is not worth having expensive controls to protect assets that don't have a high value.

Asset	Value	Exposure
The information system	High. Required to support all clinical consultations. Potentially safety critical.	High. Financial loss as clinics may have to be canceled. Costs of restoring system. Possible patient harm if treatment cannot be prescribed.
The patient database	High. Required to support all clinical consultations. Potentially safety critical.	High. Financial loss as clinics may have to be canceled. Costs of restoring system. Possible patient harm if treatment cannot be prescribed.
An individual patient record	Normally low, although may be high for specific high-profile patients	Low direct losses but possible loss of reputation.

Figure 13.6 Asset analysis in a preliminary risk assessment report for the Mentcare system

8. *Security requirements definition,* where knowledge of the exposure, threats, and control assessments is used to derive system security requirements. These requirements may apply to the system infrastructure or the application system.

The Mentcare patient management system is a security-critical system. Figures 13.6 and 13.7 are fragments of a report that documents the risk analysis of that software system. Figure 13.6 is an asset analysis that describes the assets in the system and their value. Figure 13.7 shows some of the threats that a system may face.

Once a preliminary risk assessment has been completed, then requirements can be proposed that aim to avoid, detect, and mitigate risks to the system. However, creating these requirements is not a formulaic or automated process. It requires inputs from both engineers and domain experts to suggest requirements based on their understanding of the risk analysis and the functional requirements of the software system. Some examples of the Mentcare system security requirements and associated risks are:

1. Patient information shall be downloaded, at the start of a clinic session, from the database to a secure area on the system client.

   Risk: Damage from denial-of-service attack. Maintaining local copies means that access is still possible.

2. All patient information on the system client shall be encrypted.

   Risk: External access to patient records. If data is encrypted, then attacker must have access to the encryption key to discover patient information.

3. Patient information shall be uploaded to the database when a clinic session is over and deleted from the client computer.

   Risk: External access to patience records through stolen laptop.

4. A log of all changes made to the system database and the initiator of these changes shall be maintained on a separate computer from the database server.

   Risk: Insider or external attacks that corrupt current data. A log should allow up-to-date records to be re-created from a backup.

Threat	Probability	Control	Feasibility
An unauthorized user gains access as system manager and makes system unavailable	Low	Only allow system management from specific locations that are physically secure.	Low cost of implementation, but care must be taken with key distribution and to ensure that keys are available in the event of an emergency.
An unauthorized user gains access as system user to confidential information	High	Require all users to authenticate themselves using a biometric mechanism.	Technically feasible but high- cost solution. Possible user resistance.
		Log all changes to patient information to track system usage.	Simple and transparent to implement and also supports recovery.

Figure 13.7 Threat and control analysis in a preliminary risk assessment report

The first two requirements are related—patient information is downloaded to a local machine, so that consultations may continue if the patient database server is attacked or becomes unavailable. However, this information must be deleted so that later users of the client computer cannot access the information. The fourth requirement is a recovery and auditing requirement. It means that changes can be recovered by replaying the change log and that it is possible to discover who has made the changes. This accountability discourages misuse of the system by authorized staff.

13.3.1 Misuse cases

The derivation of security requirements from a risk analysis is a creative process involving engineers and domain experts. One approach that has been developed to support this process for users of the UML is the idea of misuse cases (Sindre and Opdahl 2005). Misuse cases are scenarios that represent malicious interactions with a system. You can use these scenarios to discuss and identify possible threats and, therefore also determine the system's security requirements. They can be used alongside use cases when deriving the system requirements (Chapters 4 and 5).

Misuse cases are associated with use case instances and represent threats or attacks associated with these use cases. They may be included in a use case diagram but should also have a more complete and detailed textual description. In Figure 13.8, I have taken the use cases for a medical receptionist using the Mentcare system and have added misuse cases. These are normally represented as black ellipses.

As with use cases, misuse cases can be described in several ways. I think that it is most helpful to describe them as a supplement to the original use case description. I also think it is best to have a flexible format for misuse cases as different types of attack have to be described in different ways. Figure 13.9 shows the original description of the Transfer Data use case (Figure 5.4), with the addition of a misuse case description.

The problem with misuse cases mirrors the general problem of use cases, which is that interactions between end-users and a system do not capture all of the system

Figure 13.8 Misuse cases

Mentcare system: Transfer data	
Actors	Medical receptionist, Patient records system (PRS)
Description	A receptionist may transfer data from the Mentcare system to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

Mentcare system: Intercept transfer (Misuse case)	
Actors	Medical receptionist, Patient records system (PRS), Attacker
Description	A receptionist transfers data from his or her PC to the Mentcare system on the server. An attacker intercepts the data transfer and takes a copy of that data.
Data (assets)	Patient's personal information, treatment summary
Attacks	A network monitor is added to the system, and packets from the receptionist to the server are intercepted. A spoof server is set up between the receptionist and the database server so that receptionist believes they are interacting with the real system.
Mitigations	All networking equipment must be maintained in a locked room. Engineers accessing the equipment must be accredited. All data transfers between the client and server must be encrypted. Certificate-based client–server communication must be used.
Requirements	All communications between the client and the server must use the Secure Socket Layer (SSL). The https protocol uses certificate-based authentication and encryption.

Figure 13.9 Misuse case descriptions

requirements. Misuse cases can be used as part of the security requirements engineering process, but you also need to consider risks that are associated with system stakeholders who do not interact directly with the system.

13.4 Secure systems design

It is very difficult to add security to a system after it has been implemented. Therefore, you need to take security issues into account during the systems design process and make design choices that enhance the security of a system. In this section, I focus on two application-independent issues relevant to secure systems design:

1. *Architectural design*—how do architectural design decisions affect the security of a system?
2. *Good practice*—what is accepted good practice when designing secure systems?

Of course, these are not the only design issues that are important for security. Every application is different, and security design also has to take into account the purpose, criticality, and operational environment of the application. For example, if you are designing a military system, you need to adopt their security classification model (secret, top secret, etc.) If you are designing a system that maintains personal information, you may have to take into account data protection legislation that places restrictions on how data is managed.

Using redundancy and diversity, which is essential for dependability, may mean that a system can resist and recover from attacks that target specific design or implementation characteristics. Mechanisms to support a high level of availability may help the system to recover from denial-of-service attacks, where the aim of an attacker is to bring down the system and stop it from working properly.

Designing a system to be secure inevitably involves compromises. It is usually possible to design multiple security measures into a system that will reduce the chances of a successful attack. However, these security measures may require additional computation and so affect the overall performance of the system. For example, you can reduce the chances of confidential information being disclosed by encrypting that information. However, this means that users of the information have to wait for it to be decrypted, which may slow down their work.

There are also tensions between security and usability—another emergent system property. Security measures sometimes require the user to remember and provide additional information (e.g., multiple passwords). However, sometimes users forget this information, so the additional security means that they can't use the system.

System designers have to find a balance between security, performance, and usability. This depends on the type of system being developed, the expectations of its users, and its operational environment. For example, in a military system, users are familiar with high-security systems and so accept and follow processes that require frequent checks. In a system for stock trading, where speed is essential, interruptions of operation for security checks would be completely unacceptable.

Denial-of-service attacks

Denial-of-service attacks attempt to bring down a networked system by bombarding it with a huge number of service requests, usually from hundreds of attacking systems. These place a load on the system for which it was not designed and they exclude legitimate requests for system service. Consequently, the system may become unavailable either because it crashes with the heavy load or has to be taken offline by system managers to stop the flow of requests.

http://software-engineering-book.com/web/denial-of-service/

13.4.1 Design risk assessment

Security risk assessment during requirements engineering identifies a set of high-level security requirements for a system. However, as the system is designed and implemented, architectural and technology decisions made during the system design process influence the security of a system. These decisions generate new design requirements and may mean that existing requirements have to change.

System design and the assessment of design-related risks are interleaved processes (Figure 13.10). Preliminary design decisions are made, and the risks associated with these decisions are assessed. This assessment may lead to new requirements to mitigate the risks that have been identified or design changes to reduce these risks. As the system design evolves and is developed in more detail, the risks are reassessed and the results are fed back to the system designers. The design risk assessment process ends when the design is complete and the remaining risks are acceptable.

When assessing risks during design and implementation, you have more information about what needs to be protected, and you also will know something about the vulnerabilities in the system. Some of these vulnerabilities will be inherent in the design choices made. For example, an inherent vulnerability in password-based authentication is that an authorized user reveals their password to an unauthorized user. So, if password-based authentication is used, the risk assessment process may suggest new requirements to mitigate the risk. For example, there may be a requirement for multifactor authentication where users must authenticate themselves using some personal knowledge as well as a password.

Figure 13.10 Interleaved design and risk assessment

Figure 13.11 Design risk assessment

Figure 13.11 is a model of the design risk assessment process. The key difference between preliminary risk analysis and design risk assessment is that, at the design stage, you now have information about information representation and distribution and the database organization for the high-level assets that have to be protected. You also know about important design decisions such as the software to be reused, infrastructure controls and protection, and so forth. Based on this information, your assessment can identify changes to the security requirements and the system design to provide additional protection for the important system assets.

Two examples from the Mentcare system illustrate how protection requirements are influenced by decisions on information representation and distribution:

1. You may make a design decision to separate personal patient information and information (design assets) about treatments received, with a key linking these records. The treatment information is technical and so much less sensitive than the personal patient information. If the key is protected, then an attacker will only be able to access routine information, without being able to link this to an individual patient.

2. Assume that, at the beginning of a session, a design decision is made to copy patient records to a local client system. This allows work to continue if the server is unavailable. It makes it possible for a healthcare worker to access patient records from a laptop, even if no network connection is available. However, you now have two sets of records to protect and the client copies are subject to additional risks, such as theft of the laptop computer. You therefore have to think about what controls should be used to reduce risk. You may therefore include a requirement that client records held on laptops or other personal computers may have to be encrypted.

Figure 13.12 Vulnerabilities associated with technology choices

To illustrate how decisions on development technologies influence security, assume that the health care provider has decided to build a Mentcare system using an off-the-shelf information system for maintaining patient records. This system has to be configured for each type of clinic in which it is used. This decision has been made because it appears to offer the most extensive functionality for the lowest development cost and fastest deployment time.

When you develop an application by reusing an existing system, you have to accept the design decisions made by the developers of that system. Let us assume that some of these design decisions are:

1. System users are authenticated using a login name/password combination. No other authentication method is supported.
2. The system architecture is client–server, with clients accessing data through a standard web browser on a client computer.
3. Information is presented to users as an editable web form. They can change information in place and upload the revised information to the server.

For a generic system, these design decisions are perfectly acceptable, but design risk assessment shows that they have associated vulnerabilities. Examples of these possible vulnerabilities are shown in Figure 13.12.

Once vulnerabilities have been identified, you then have to decide what steps you can take to reduce the associated risks. This will often involve making decisions

about additional system security requirements or the operational process of using the system. Examples of these requirements might be:

1. A password checker program shall be made available and shall be run daily to check all user passwords. User passwords that appear in the system dictionary shall be identified, and users with weak passwords shall be reported to system administrators.
2. Access to the system shall only be allowed to client computers that have been approved and registered with the system administrators.
3. Only one approved web browser shall be installed on client computers.

As an off-the-shelf system is used, it isn't possible to include a password checker in the application system itself, so a separate system must be used. Password checkers analyze the strength of user passwords when they are set up and notify users if they have chosen weak passwords. Therefore, vulnerable passwords can be identified reasonably quickly after they have been set up, and action can then be taken to ensure that users change their password.

The second and third requirements mean that all users will always access the system through the same browser. You can decide what is the most secure browser when the system is deployed and install that on all client computers. Security updates are simplified because there is no need to update different browsers when security vulnerabilities are discovered and fixed.

The process model shown in Figure 13.10 assumes a design process where the design is developed to a fairly detailed level before implementation begins. This is not the case for agile processes where the design and the implementation are developed together, with the code refactored as the design is developed. Frequent delivery of system increments does not allow time for a detailed risk assessment, even if information on assets and technology choices is available.

The issues surrounding security and agile development have been widely discussed (Lane 2010; Schoenfield 2013). So far, the issue has not really been resolved—some people think that a fundamental conflict exists between security and agile development, and others believe that this conflict can be resolved using security-focused stories (Safecode 2012). This remains an outstanding problem for developers of agile methods. Meanwhile, many security-conscious companies refuse to use agile methods because they conflict with their security and risk analysis policies.

13.4.2 Architectural design

Software architecture design decisions can have profound effects on the emergent properties of a software system. If an inappropriate architecture is used, it may be very difficult to maintain the confidentiality and integrity of information in the system or to guarantee a required level of system availability.

Figure 13.13 A layered protection architecture

In designing a system architecture that maintains security, you need to consider two fundamental issues:

1. *Protection*—how should the system be organized so that critical assets can be protected against external attack?
2. *Distribution*—how should system assets be distributed so that the consequences of a successful attack are minimized?

These issues are potentially conflicting. If you put all your assets in one place, then you can build layers of protection around them. As you only have to build a single protection system, you may be able to afford a strong system with several protection layers. However, if that protection fails, then all your assets are compromised. Adding several layers of protection also affects the usability of a system, so it may mean that it is more difficult to meet system usability and performance requirements.

On the other hand, if you distribute assets, they are more expensive to protect because protection systems have to be implemented for each distributed asset. Typically, then, you cannot afford to implement as many protection layers. The chances are greater that the protection will be breached. However, if this happens, you don't suffer a total loss. It may be possible to duplicate and distribute information assets so that if one copy is corrupted or inaccessible, then the other copy can be used. However, if the information is confidential, keeping additional copies increases the risk that an intruder will gain access to this information.

For the Mentcare system, a client–server architecture with a shared central database is used. To provide protection, the system has a layered architecture with the

critical protected assets at the lowest level in the system. Figure 13.13 illustrates this multilevel system architecture in which the critical assets to be protected are the records of individual patients.

To access and modify patient records, an attacker has to penetrate three system layers:

1. *Platform-level protection.* The top level controls access to the platform on which the patient record system runs. This usually involves a user signing-on to a particular computer. The platform will also normally include support for maintaining the integrity of files on the system, backups, and so on.
2. *Application-level protection.* The next protection level is built into the application itself. It involves a user accessing the application, being authenticated, and getting authorization to take actions such as viewing or modifying data. Application-specific integrity management support may be available.
3. *Record-level protection.* This level is invoked when access to specific records is required, and involves checking that a user is authorized to carry out the requested operations on that record. Protection at this level might also involve encryption to ensure that records cannot be browsed using a file browser. Integrity checking using, for example, cryptographic checksums can detect changes that have been made outside the normal record update mechanisms.

The number of protection layers that you need in any particular application depends on the criticality of the data. Not all applications need protection at the record level, and, therefore, coarser-grain access control is more commonly used. To achieve security, you should not allow the same user credentials to be used at each level. Ideally, if you have a password-based system, then the application password should be different from both the system password and the record-level password. However, multiple passwords are difficult for users to remember, and they find repeated requests to authenticate themselves irritating. Therefore, you often have to compromise on security in favor of system usability.

If protection of data is a critical requirement, then a centralized client–server architecture is usually the most effective security architecture. The server is responsible for protecting sensitive data. However, if the protection is compromised, then the losses associated with an attack are high, as all data may be lost or damaged. Recovery costs may also be high (e.g., all user credentials may have to be reissued). Centralized systems are also more vulnerable to denial-of-service attacks, which overload the server and make it impossible for anyone to access the system database.

If the consequences of a server breach are high, you may decide to use an alternative distributed architecture for the application. In this situation, the system's assets are distributed across a number of different platforms, with separate protection mechanisms used for each of these platforms. An attack on one node might mean that some assets are unavailable, but it would still be possible to provide some system services. Data can be replicated across the nodes in the system so that recovery from attacks is simplified.

Figure 13.14 illustrates the architecture of a banking system for trading in stocks and funds on the New York, London, Frankfurt, and Hong Kong markets. The system

Figure 13.14 Distributed assets in an equity trading system

is distributed so that data about each market is maintained separately. Assets required to support the critical activity of equity trading (user accounts and prices) are replicated and available on all nodes. If a node of the system is attacked and becomes unavailable, the critical activity of equity trading can be transferred to another country and so can still be available to users.

I have already discussed the problem of finding a balance between security and system performance. A problem of secure system design is that in many cases, the architectural style that is best for the security requirements may not be the best one for meeting the performance requirements. For example, say an application has an absolute requirement to maintain the confidentiality of a large database and another requirement for very fast access to that data. A high-level of protection suggests that layers of protection are required, which means that there must be communications between the system layers. This has an inevitable performance overhead and so will slow down access to the data.

If an alternative architecture is used, then implementing protection and guaranteeing confidentiality may be more difficult and expensive. In such a situation, you have to discuss the inherent conflicts with the customer who is paying for the system and agree on how these conflicts are to be resolved.

13.4.3 Design guidelines

There are no easy ways to ensure system security. Different types of systems require different technical measures to achieve a level of security that is acceptable to the system owner. The attitudes and requirements of different groups of users profoundly affect what is and is not acceptable. For example, in a bank, users are likely to accept a higher level of security, and hence more intrusive security procedures than, say, in a university.

However, some general guidelines have wide applicability when designing system security solutions. These guidelines encapsulate good design practice for secure systems engineering. General design guidelines for security, such as those discussed, below, have two principal uses:

1. They help raise awareness of security issues in a software engineering team. Software engineers often focus on the short-term goal of getting the software working and delivered to customers. It is easy for them to overlook security issues. Knowledge of these guidelines can mean that security issues are considered when software design decisions are made.

2. They can be used as a review checklist that can be used in the system validation process. From the high-level guidelines discussed here, more specific questions can be derived that explore how security has been engineered into a system.

Security guidelines are sometimes very general principles such as "Secure the weakest link in a system," "Keep it simple," and "Avoid security through obscurity." I think these general guidelines are too vague to be of real use in the design process. Consequently, I have focused here on more specific design guidelines. The 10 design guidelines, summarized in Figure 13.15, have been taken from different sources (Schneier 2000; Viega and McGraw 2001; Wheeler 2004).

Guideline 1: Base security decisions on an explicit security policy

An organizational security policy is a high-level statement that sets out fundamental security conditions for an organization. It defines the "what" of security rather than the "how." so the policy should not define the mechanisms to be used to provide and enforce security. In principle, all aspects of the security policy should be reflected in the system requirements. In practice, especially if agile development is used, this is unlikely to happen.

Designers should use the security policy as a framework for making and evaluating design decisions. For example, say you are designing an access control system for the Mentcare system. The hospital security policy may state that only accredited clinical staff may modify electronic patient records. This leads to requirements to check the accreditation of anyone attempting to modify the system and to reject modifications from unaccredited people.

The problem that you may face is that many organizations do not have an explicit systems security policy. Over time, changes may have been made to systems in response to identified problems, but with no overarching policy document to guide the evolution of a system. In such situations, you need to work out and document the policy from examples and confirm it with managers in the company.

Design guidelines for security	
1	Base security decisions on an explicit security policy
2	Use defense in depth
3	Fail securely
4	Balance security and usability
5	Log user actions
6	Use redundancy and diversity to reduce risk
7	Specify the format of system inputs
8	Compartmentalize your assets
9	Design for deployment
10	Design for recovery

Figure 13.15 Design guidelines for secure systems engineering

Guideline 2: Use defense in depth

In any critical system, it is good design practice to try to avoid a single point of failure. That is, a single failure in part of the system should not result in an overall systems failure. In security terms, this means that you should not rely on a single mechanism to ensure security; rather, you should employ several different techniques. This concept is sometimes called "defense in depth."

An example of defense in depth is multifactor authentication. For example, if you use a password to authenticate users to a system, you may also include a challenge/response authentication mechanism where users have to pre-register questions and answers with the system. After they have input their login credentials, they must then answer questions correctly before being allowed access.

Guideline 3: Fail securely

System failures are inevitable in all systems, and, in the same way that safety-critical systems should always fail-safe; security-critical systems should always "fail-secure." When the system fails, you should not use fallback procedures that are less secure than the system itself. Nor should system failure mean that an attacker can access data that would not normally be allowed.

For example, in the Mentcare system, I suggested a requirement that patient data should be downloaded to a system client at the beginning of a clinic session. This speeds up access and means that access is possible if the server is unavailable. Normally, the server deletes this data at the end of the clinic session. However, if the server has failed, then it is possible that the information on the client will be maintained. A fail-secure approach in those circumstances is to encrypt all patient data stored on the client. This means that an unauthorized user cannot read the data.

Guideline 4: Balance security and usability

The demands of security and usability are often contradictory. To make a system secure, you have to introduce checks that users are authorized to use the system and

that they are acting in accordance with security policies. All of these inevitably make demands on users—they may have to remember login names and passwords, only use the system from certain computers, and so on. These mean that it takes users more time to get started with the system and use it effectively. As you add security features to a system, it usually becomes more difficult to use. I recommend Cranor and Garfinkel's book (Cranor and Garfinkel 2005), which discusses a wide range of issues in the general area of security and usability.

There comes a point when it is counterproductive to keep adding on new security features at the expense of usability. For example, if you require users to input multiple passwords or to change their passwords to impossible to remember character strings at frequent intervals, they will simply write down these passwords. An attacker (especially an insider) may then be able to find the passwords that have been written down and gain access to the system.

Guideline 5: Log user actions

If it is practically possible to do so, you should always maintain a log of user actions. This log should, at least, record who did what, the assets used and the time and date of the action. If you maintain this as a list of executable commands, you can replay the log to recover from failures. You also need tools that allow you to analyze the log and detect potentially anomalous actions. These tools can scan the log and find anomalous actions, and thus help detect attacks and trace how the attacker gained access to the system.

Apart from helping recover from failure, a log of user actions is useful because it acts as a deterrent to insider attacks. If people know that their actions are being logged, then they are less likely to do unauthorized things. This is most effective for casual attacks, such as a nurse looking up patient records of neighbors, or for detecting attacks where legitimate user credentials have been stolen through social engineering. Of course, this approach is not foolproof, as technically skilled insiders may also be able to access and change the log.

Guideline 6: Use redundancy and diversity to reduce risk

Redundancy means that you maintain more than one version of software or data in a system. Diversity, when applied to software, means that the different versions should not rely on the same platform or be implemented using the same technologies. Therefore, platform or technology vulnerabilities will not affect all versions and so will lead to a common failure.

I have already discussed examples of redundancy—maintaining patient information on both the server and the client, first in the Mentcare system and then in the distributed equity trading system shown in Figure 13.14. In the patient records system, you could use diverse operating systems on the client and the server (e.g., Linux on the server, Windows on the client). This ensures that an attack based on an operating system vulnerability will not affect both the server and the client. Of course, running multiple operating systems leads to higher systems management costs. You have to trade off security benefits against this increased cost.

Guideline 7: Specify the format of system inputs

A common attack on a system involves providing the system with unexpected inputs that cause it to behave in an unanticipated way. These inputs may simply cause a system crash, resulting in a loss of service, or the inputs could be made up of malicious code that is executed by the system. Buffer overflow vulnerabilities, first demonstrated in the Internet worm (Spafford 1989) and commonly used by attackers, may be triggered using long input strings. So-called SQL poisoning, where a malicious user inputs an SQL fragment that is interpreted by a server, is another fairly common attack.

You can avoid many of these problems if you specify the format and structure of the system inputs that are expected. This specification should be based on your knowledge of the expected system inputs. For example, if a surname is to be input, you might specify that all characters must be alphabetic with no numbers or punctuation (apart from a hyphen) allowed. You might also limit the length of the name. For example, no one has a family name with more than 40 characters, and no addresses are more than 100 characters long. If a numeric value is expected, no alphabetic characters should be allowed. This information is then used in input checks when the system is implemented.

Guideline 8: Compartmentalize your assets

Compartmentalizing means that you should not provide users with access to all information in a system. Based on a general "need to know" security principle, you should organize the information in a system into compartments. Users should only have access to the information that they need for their work, rather than to all of the information in a system. This means that the effects of an attack that compromises an individual user account may be contained. Some information may be lost or damaged, but it is unlikely that all of the information in the system will be affected.

For example, the Mentcare system could be designed so that clinic staff will normally only have access to the records of patients who have an appointment at their clinic. They should not normally have access to all patient records in the system. Not only does this limit the potential loss from insider attacks, but it also means that if an intruder steals their credentials, then they cannot damage all patient records.

Having said this, you also may have to have mechanisms in the system to grant unexpected access—say to a patient who is seriously ill and requires urgent treatment without an appointment. In those circumstances, you might use some alternative secure mechanism to override the compartmentalization in the system. In such situations, where security is relaxed to maintain system availability, it is essential that you use a logging mechanism to record system usage. You can then check the logs to trace any unauthorized use.

Guideline 9: Design for deployment

Many security problems arise because the system is not configured correctly when it is deployed in its operational environment. Deployment means installing the software

on the computers where it will execute and setting software parameters to reflect the execution environment and the preferences of the system user. Mistakes such as forgetting to turn off debugging facilities or forgetting to change the default administration password can introduce vulnerabilities into a system.

Good management practice can avoid many security problems that arise from configuration and deployment mistakes. However, software designers have the responsibility to "design for deployment." You should always provide support for deployment that reduces the chances of users and system administrators making mistakes when configuring the software.

I recommend four ways to incorporate deployment support in a system:

1. *Include support for viewing and analyzing configurations* You should always include facilities in a system that allow administrators or permitted users to examine the current configuration of the system.
2. *Minimize default privileges* You should design software so that the default configuration of a system provides minimum essential privileges.
3. *Localize configuration settings* When designing system configuration support, you should ensure that everything in a configuration that affects the same part of a system is set up in the same place.
4. *Provide easy ways to fix security vulnerabilities* You should include straightforward mechanisms for updating the system to repair security vulnerabilities that have been discovered.

Deployment issues are less of a problem than they used to be as more and more software does not require client installation. Rather, the software runs as a service and is accessed through a web browser. However, server software is still vulnerable to deployment errors and omissions, and some types of system require dedicated software running on the user's computer.

Guideline 10: Design for recovery

Irrespective of how much effort you put into maintaining systems security, you should always design your system with the assumption that a security failure could occur. Therefore, you should think about how to recover from possible failures and restore the system to a secure operational state. For example, you may include a backup authentication system in case your password authentication is compromised.

For example, say an unauthorized person from outside the clinic gains access to the Mentcare system and you don't know how that person obtained a valid login/password combination. You need to re-initialize the authentication system and not just change the credentials used by the intruder. This is essential because the intruder may also have gained access to other user passwords. You need, therefore, to ensure that all authorized users change their passwords. You also must ensure that the unauthorized person does not have access to the password-changing mechanism.

You therefore have to design your system to deny access to everyone until they have changed their password and to email all users asking them to make the change. You need an alternative mechanism to authenticate real users for password change, assuming that their chosen passwords may not be secure. One way of doing this is to use a challenge/response mechanism, where users have to answer questions for which they have pre-registered answers. This is only invoked when passwords are changed, allowing for recovery from the attack with relatively little user disruption.

Designing for recoverability is an essential element of building resilience into systems. I cover this topic in more detail in Chapter 14.

13.4.4 Secure systems programming

Secure system design means designing security into an application system. However, as well as focusing on security at the design level, it is also important to consider security when programming a software system. Many successful attacks on software rely on program vulnerabilities that were introduced when the program was developed.

The first widely known attack on Internet-based systems happened in 1988 when a worm was introduced into Unix systems across the network (Spafford 1989). This took advantage of a well-known programming vulnerability. If systems are programmed in C, there is no automatic array bound checking. An attacker can include a long string with program commands as an input, and this overwrites the program stack and can cause control to be transferred to malicious code. This vulnerability has been exploited in many other systems programmed in C or C++ since then.

This example illustrates two important aspects of secure systems programming:

1. Vulnerabilities are often language-specific. Array bound checking is automatic in languages such as Java, so this is not a vulnerability that can be exploited in Java programs. However, millions of programs are written in C and C++ as these allow for the development of more efficient software. Thus. simply avoiding the use of these languages is not a realistic option.

2. Security vulnerabilities are closely related to program reliability. The above example caused the program concerned to crash, so actions taken to improve program reliability can also improve system security.

In Chapter 11, I introduced programming guidelines for dependable system programming. These are shown in Figure 13.16. These guidelines also help improve the security of a program as attackers focus on program vulnerabilities to gain access to a system. For example, an SQL poisoning attack is based on the attacker filling in a form with SQL commands rather than the text expected by the system. These can corrupt the database or release confidential information. You can completely avoid this problem if you implement input checks (Guideline 2) based on the expected format and structure of the inputs.

Dependable programming guidelines

1. Limit the visibility of information in a program.
2. Check all inputs for validity.
3. Provide a handler for all exceptions.
4. Minimize the use of error-prone constructs.
5. Provide restart capabilities.
6. Check array bounds.
7. Include timeouts when calling external components.
8. Name all constants that represent real-world values.

Figure 13.16 Dependable programming guidelines

13.5 Security testing and assurance

The assessment of system security is increasingly important so that we can be confident that the systems we use are secure. The verification and validation processes for web-based systems should therefore focus on security assessment, where the ability of the system to resist different types of attack is tested. However, as Anderson explains (Anderson 2008), this type of security assessment is very difficult to carry out. Consequently, systems are often deployed with security loopholes. Attackers use these vulnerabilities to gain access to the system or to cause damage to the system or its data.

Fundamentally, security testing is difficult for two reasons:

1. Security requirements, like some safety requirements, are "shall not" requirements. That is, they specify what should not happen rather than system functionality or required behavior. It is not usually possible to define this unwanted behavior as simple constraints to be checked by the system.

   If resources are available, you can demonstrate, in principle at least, that a system meets its functional requirements. However, it is impossible to prove that a system does not do something. Irrespective of the amount of testing, security vulnerabilities may remain in a system after it has been deployed.

   You may, of course, generate functional requirements that are designed to guard the system against some known types of attack. However, you cannot derive requirements for unknown or unanticipated types of attack. Even in systems that have been in use for many years, an ingenious attacker can discover a new attack and can penetrate what was thought to be a secure system.

2. The people attacking a system are intelligent and are actively looking for vulnerabilities that they can exploit. They are willing to experiment with the system and to try things that are far outside normal activity and system use. For example, in a surname field they may enter 1000 characters with a mixture of letters, punctuation, and numbers simply to see how the system responds.

   Once they find a vulnerability, they publicize it and so increase the number of possible attackers. Internet forums have been set up to exchange information about system vulnerabilities. There is also a thriving market in malware where

Security checklist
1. Do all files that are created in the application have appropriate access permissions? The wrong access permissions may lead to these files being accessed by unauthorized users.
2. Does the system automatically terminate user sessions after a period of inactivity? Sessions that are left active may allow unauthorized access through an unattended computer.
3. If the system is written in a programming language without array bound checking, are there situations where buffer overflow may be exploited? Buffer overflow may allow attackers to send code strings to the system and then execute them.
4. If passwords are set, does the system check that passwords are "strong"? Strong passwords consist of mixed letters, numbers, and punctuation, and are not normal dictionary entries. They are more difficult to break than simple passwords.
5. Are inputs from the system's environment always checked against an input specification? Incorrect processing of badly formed inputs is a common cause of security vulnerabilities.

Figure 13.17 Examples of entries in a security checklist

attackers can get access to kits that help them easily develop malware such as worms and keystroke loggers.

Attackers may try to discover the assumptions made by system developers and then challenge these assumptions to see what happens. They are in a position to use and explore a system over a period of time and analyze it using software tools to discover vulnerabilities that they may be able to exploit. They may, in fact, have more time to spend on looking for vulnerabilities than system test engineers, as testers must also focus on testing the system.

You may use a combination of testing, tool-based analysis, and formal verification to check and analyze the security of an application system:

1. *Experience-based testing* In this case, the system is analyzed against types of attack that are known to the validation team. This may involve developing test cases or examining the source code of a system. For example, to check that the system is not susceptible to the well-known SQL poisoning attack, you might test the system using inputs that include SQL commands. To check that buffer overflow errors will not occur, you can examine all input buffers to see if the program is checking that assignments to buffer elements are within bounds.

   Checklists of known security problems may be created to assist with the process. Figure 13.17 gives some examples of questions that might be used to drive experience-based testing. Checks on whether design and programming guidelines for security have been followed may also be included in a security problem checklist.

2. *Penetration testing* This is a form of experience-based testing where it is possible to draw on experience from outside the development team to test an application system. The penetration testing teams are given the objective of breaching the system security. They simulate attacks on the system and use their ingenuity to discover new ways to compromise the system security. Penetration testing team

members should have previous experience with security testing and finding security weaknesses in systems.

3. *Tool-based analysis* In this approach, security tools such as password checkers are used to analyze the system. Password checkers detect insecure passwords such as common names or strings of consecutive letters. This approach is really an extension of experience-based validation, where experience of security flaws is embodied in the tools used. Static analysis is, of course, another type of tool-based analysis, which has become increasingly used.

   Tool-based static analysis (Chapter 12) is a particularly useful approach to security checking. A static analysis of a program can quickly guide the testing team to areas of a program that may include errors and vulnerabilities. Anomalies revealed in the static analysis can be directly fixed or can help identify tests that need to be done to reveal whether or not these anomalies actually represent a risk to the system. Microsoft uses static analysis routinely to check its software for possible security vulnerabilities (Jenney 2013). Hewlett-Packard offers a tool called Fortify (Hewlett-Packard 2012) specifically designed for checking Java programs for security vulnerabilities.

4. *Formal verification* I have discussed the use of formal program verification in Chapters 10 and 12. Essentially, this involves making formal, mathematical arguments that demonstrate that a program conforms to its specification. Hall and Chapman (Hall and Chapman 2002) demonstrated the feasibility of proving that a system met its formal security requirements more than 10 years ago, and there have been a number of other experiments since then. However, as in other areas, formal verification for security is not widely used. It requires specialist expertise and is unlikely to be as cost-effective as static analysis.

Security testing takes a long time, and, usually, the time available to the testing team is limited. This means that you should adopt a risk-based approach to security testing and focus on what you think are the most significant risks faced by the system. If you have an analysis of the security risks to the system, these can be used to drive the testing process. As well as testing the system against the security requirements derived from these risks, the test team should also try to break the system by adopting alternative approaches that threaten the system assets.

KEY POINTS

- Security engineering focuses on how to develop and maintain software systems that can resist malicious attacks intended to damage a computer-based system or its data.
- Security threats can be threats to the confidentiality, integrity, or availability of a system or its data.

- Security risk management involves assessing the losses that might ensue from attacks on a system, and deriving security requirements that are aimed at eliminating or reducing these losses.
- To specify security requirements, you should identify the assets that are to be protected and define how security techniques and technology should be used to protect these assets.
- Key issues when designing a secure systems architecture include organizing the system structure to protect key assets and distributing the system assets to minimize the losses from a successful attack.
- Security design guidelines sensitize system designers to security issues that they may not have considered. They provide a basis for creating security review checklists.
- Security validation is difficult because security requirements state what should not happen in a system, rather than what should. Furthermore, system attackers are intelligent and may have more time to probe for weaknesses than is available for security testing.

FURTHER READING

Security Engineering: A Guide to Building Dependable Distributed Systems, 2nd ed. This is a thorough and comprehensive discussion of the problems of building secure systems. The focus is on systems rather than software engineering, with extensive coverage of hardware and networking, with excellent examples drawn from real system failures. (R. Anderson, John Wiley & Sons, 2008) http://www.cl.cam.ac.uk/~rja14/book.html

24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them. I think this is one of the best practical books on secure systems programming. The authors discuss the most common programming vulnerabilities and describe how they can be avoided in practice. (M. Howard, D. LeBlanc, and J. Viega, McGraw-Hill, 2009).

Computer Security: Principles and Practice. This is a good general text on computer security issues. It covers security technology, trusted systems, security management, and cryptography. (W. Stallings and L. Brown, Addison-Wesley, 2012).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/security-and-resilience/

EXERCISES

13.1. Describe the security dimensions and security levels that have to be considered in secure systems engineering.

13.2. For the Mentcare system, suggest an example of an asset, an exposure, a vulnerability, an attack, a threat, and a control, in addition to those discussed in this chapter.

13.3. Explain why security is considered a more challenging problem than safety in a system.

13.4. Extend the table in Figure 13.7 to identify two further threats to the Mentcare system, along with associated controls. Use these as a basis for generating software security requirements that implement the proposed controls.

13.5. Explain, using an analogy drawn from a non-software engineering context, why a layered approach to asset protection should be used.

13.6. Explain why it is important to log user actions in the development of secure systems.

13.7. For the equity trading system discussed in Section 13.4.2, whose architecture is shown in Figure 13.14, suggest two further plausible attacks on the system and propose possible strategies that could counter these attacks.

13.8. Explain why it is important when writing secure systems to validate all user inputs to check that these have the expected format.

13.9. Suggest how you would go about validating a password protection system for an application that you have developed. Explain the function of any tools that you think may be useful.

13.10. The Mentcare system has to be secure against attacks that might reveal confidential patient information. Suggest three possible attacks against this system that might occur. Using this information, extend the checklist in Figure 13.17 to guide testers of the Mentcare system.

REFERENCES

Anderson, R. 2008. *Security Engineering, 2nd ed.* Chichester, UK: John Wiley & Sons.

Cranor, L. and S. Garfinkel. 2005. *Designing Secure Systems That People Can Use.* Sebastopol, CA: O'Reilly Media Inc.

Firesmith, D. G. 2003. "Engineering Security Requirements." *Journal of Object Technology* 2 (1): 53–68. http://www.jot.fm/issues/issue_2003_01/column6

Hall, A., and R. Chapman. 2002. "Correctness by Construction: Developing a Commercially Secure System." *IEEE Software* 19 (1): 18–25. doi:10.1109/52.976937.

Hewlett-Packard. 2012. "Securing Your Enterprise Software: Hp Fortify Code Analyzer." http://h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA4-2455ENW&cc=us&lc=en

Jenney, P. 2013. "Static Analysis Strategies: Success with Code Scanning." http://msdn.microsoft.com/en-us/security/gg615593.aspx

Lane, A. 2010. "Agile Development and Security." https://securosis.com/blog/agile-development-and-security

Pfleeger, C. P., and S. L. Pfleeger. 2007. *Security in Computing, 4th ed.* Boston: Addison-Wesley.

Safecode. 2012. "Practical Security Stories and Security Tasks for Agile Development Environments." http://www.safecode.org/publications/SAFECode_Agile_Dev_Security0712.pdf

Schneier, B. 1999. "Attack Trees." *Dr Dobbs Journal* 24 (12): 1–9. https://www.schneier.com/paper-attacktrees-ddj-ft.html

———. 2000. *Secrets and Lies: Digital Security in a Networked World.* New York: John Wiley & Sons.

Schoenfield, B. 2013. "Agile and Security: Enemies for Life?" http://brookschoenfield.com/?p=151

Sindre, G., and A. L. Opdahl. 2005. "Eliciting Security Requirements through Misuse Cases." *Requirements Engineering* 10 (1): 34–44. doi:10.1007/s00766-004-0194-4.

Spafford, E. 1989. "The Internet Worm: Crisis and Aftermath." *Comm ACM* 32 (6): 678–687. doi:10.1145/63526.63527.

Stallings, W., and L. Brown. 2012. *Computer Security: Principles, d Practice.* (*2nd ed.*) Boston: Addison-Wesley.

Viega, J., and G. McGraw. 2001. *Building Secure Software.* Boston: Addison-Wesley.

Wheeler, D. A. 2004. *Secure Programming for Linux and Unix.* Self-published. http://www.dwheeler.com/secure-programs/

14

Resilience engineering

Objectives

The objective of this chapter is to introduce the idea of resilience engineering where systems are designed to withstand adverse external events such as operator errors and cyberattacks. When you have read this chapter, you will:

- understand the differences between resilience, reliability, and security and why resilience is important for networked systems;
- be aware of the fundamental issues in building resilient systems, namely, recognition of problems, resistance to failures and attacks, recovery of critical services, and system reinstatement;
- understand why resilience is a sociotechnical rather than a technical issue and the role of system operators and managers in providing resilience;
- have been introduced to a system design method that supports resilience.

Contents

14.1 Cybersecurity
14.2 Sociotechnical resilience
14.3 Resilient systems design

In April 1970, the Apollo 13 manned mission to the moon suffered a catastrophic failure. An oxygen tank exploded in space, resulting in a serious loss of atmospheric oxygen and oxygen for the fuel cells that powered the spacecraft. The situation was life threatening, with no possibility of rescue. There were no contingency plans for this situation. However, by using equipment in unintended ways and by adapting standard procedures, the combined efforts of the spacecraft crew and ground staff worked around the problems. The spacecraft was brought back to earth safely, and all the crew survived. The overall system (people, equipment, and processes) was *resilient*. It adapted to cope with and recover from the failure.

I introduced the idea of resilience in Chapter 10, as one of the fundamental attributes of system dependability. I defined resilience in Chapter 10 as:

> *The resilience of a system is a judgment of how well that system can maintain the continuity of its critical services in the presence of disruptive events, such as equipment failure and cyberattacks.*

This is not a "standard" definition of resilience—different authors such as Laprie (Laprie 2008) and Hollnagel (Hollnagel 2006) propose general definitions based on the ability of a system to withstand change. That is, a resilient system is one that can operate successfully when some of the fundamental assumptions made by the system designers no longer hold.

For example, an initial design assumption may be that users will make mistakes but will not deliberately seek out system vulnerabilities to be exploited. If the system is used in an environment where it may be subject to cyberattacks, this is no longer true. A resilient system can cope with the environmental change and can continue to operate successfully.

While these definitions are more general, my definition of resilience is closer to how the term is now used in practice by governments and industry. It embeds three essential ideas:

1. The idea that some of the services offered by a system are critical services whose failure could have serious human, social, or economic effects.
2. The idea that some events are disruptive and can affect the ability of a system to deliver its critical services.
3. The idea that resilience is a judgment—there are no resilience metrics, and resilience cannot be measured. The resilience of a system can only be assessed by experts, who can examine the system and its operational processes.

Fundamental work on system resilience started in the safety-critical systems community, where the aim was to understand what factors led to accidents being avoided and survived. However, the increasing number of cyberattacks on networked systems has meant that resilience is now often seen as a security issue. It is essential to build systems that can withstand malicious cyberattacks and continue to deliver services to their users.

Obviously, resilience engineering is closely related to reliability and security engineering. The aim of reliability engineering is to ensure that systems do not fail. A system failure is an externally observable event, which is often a consequence of a fault in the system. Therefore, techniques such as fault avoidance and fault tolerance, as discussed in Chapter 11, have been developed to reduce the number of system faults and to trap faults before they lead to system failure.

In spite of our best efforts, faults will always be present in a large, complex system, and they may lead to system failure. Delivery schedules are short, and testing budgets are limited. Development teams are working under pressure, and it is practically impossible to detect all of the faults and security vulnerabilities in a software system. We are building systems that are so complex (see Chapter 19) that we cannot possibly understand all of the interactions between the system components. Some of these interactions may be a trigger for overall system failure.

Resilience engineering does not focus on avoiding failure but rather on accepting the reality that failures will occur. It makes two important assumptions:

1. Resilience engineering assumes that it is impossible to avoid system failures and so is concerned with limiting the costs of these failures and recovering from them.
2. Resilience engineering assumes that good reliability engineering practices have been used to minimize the number of technical faults in a system. It therefore places more emphasis on limiting the number of system failures that arise from external events such as operator errors or cyberattacks.

In practice, technical system failures are often triggered by events that are external to the system. These events may involve operator actions or user errors that are unexpected. Over the last few years, however, as the number of networked systems has increased, these events have often been cyberattacks. In a cyberattack, a malicious person or group tries to damage the system or to steal confidential information. These are now more significant than user or operator errors as a potential source of system failure.

Because of the assumption that failures will inevitably occur, resilience engineering is concerned with both the immediate recovery from failure to maintain critical services and the longer-term reinstatement of all system services. As I discuss in Section 14.3, this means that system designers have to include system features to maintain the state of the system's software and data. In the event of a failure, essential information may then be restored.

Four related resilience activities are involved in the detection of and recovery from system problems:

1. *Recognition* The system or its operators should be able to recognize the symptoms of a problem that may lead to system failure. Ideally, this recognition should be possible before the failure occurs.
2. *Resistance* If the symptoms of a problem or signs of a cyberattack are detected early, then resistance strategies may be invoked that reduce the probability that the system will fail. These resistance strategies may focus on isolating critical parts of the system so that they are unaffected by problems elsewhere. Resistance includes

Figure 14.1 Resilience activities

proactive resistance where defenses are included in a system to trap problems and reactive resistance where actions are taken when a problem is discovered.

3. *Recovery* If a failure occurs, the aim of the recovery activity is to ensure that critical system services are restored quickly so that system users are not seriously affected by the failure.

4. *Reinstatement* In this final activity, all of the system services are restored, and normal system operation can continue.

These activities lead to changes to the system state as shown in Figure 14.1, which shows the state changes in the system in the event of a cyberattack. In parallel with normal system operation, the system monitors network traffic for possible cyberattacks. In the event of a cyberattack, the system moves to a resistance state in which normal services may be restricted.

If resistance successfully repels the attack, normal service is resumed. Otherwise, the system moves to a recovery state where only critical services are available. Repairs to the damage caused by the cyberattack are carried out. Finally, when repairs are complete, the system moves to a reinstatement state. In this state, the system's services are incrementally restored. Finally, when all restoration is complete, normal service is resumed.

As the Apollo 13 example illustrates, resilience cannot be "programmed in" to a system. It is impossible to anticipate everything that might go wrong and every context where problems might arise. The key to resilience, therefore, is flexibility and adaptability. As I discuss in Section 14.2, it should be possible for system operators and managers to take actions to protect and repair the system, even if these actions are abnormal or are normally disallowed.

Increasing the resilience of a system of course has significant costs. Software may have to be purchased or modified, and additional investments made in hardware or cloud services to provide backup systems that can be used in the event of a system failure. The benefits from these costs are impossible to calculate because the losses from a failure or attack can only be calculated after the event.

Companies may therefore be reluctant to invest in resilience if they have never suffered a serious attack or associated loss. However, the increasing number of

high-profile cyberattacks that have damaged business and government systems have increased awareness of the need for resilience. It is clear that losses can be very significant, and sometimes businesses may not survive a successful cyberattack. Therefore, there is increasing investment in resilience engineering to reduce the business risks associated with system failure.

14.1 Cybersecurity

Maintaining the security of our networked infrastructure and government, business, and personal computer systems is one of the most significant problems facing our society. The ubiquity of the Internet and our dependence on computer systems have created new criminal opportunities for theft and social disruption. It is very difficult to measure the losses due to cybercrime. However, in 2013, it was estimated that losses to the global economy due to cybercrime were between $100 billion and $500 billion (InfoSecurity 2013).

As I suggested in Chapter 13, cybersecurity is a broader issue than system security engineering. Software security engineering is a primarily technical activity that focuses on techniques and technologies to ensure that application systems are secure. Cybersecurity is a sociotechnical concern. It covers all aspects of ensuring the protection of citizens, businesses, and critical infrastructures from threats that arise from their use of computers and the Internet. While technical issues are important, technology on its own cannot guarantee security. Factors that contribute to cybersecurity failures include:

- organizational ignorance of the seriousness of the problem,
- poor design and lax application of security procedures,
- human carelessness, and
- inappropriate trade-offs between usability and security.

Cybersecurity is concerned with all of an organization's IT assets from networks through to application systems. The vast majority of these assets are externally procured, and companies do not understand their detailed operation. Systems such as web browsers are large and complex programs, and inevitably they contain bugs that can be a source of vulnerability. The different systems in an organization are related to each other in many different ways. They may be stored on the same disk, share data, rely on common operating systems components, and so on. The organizational "system of systems" is incredibly complex. It is impossible to ensure that it is free of security vulnerabilities.

Consequently, you should generally assume that your systems are vulnerable to cyberattack and that, at some stage, a cyberattack is likely to occur. A successful cyberattack can have very serious financial consequences for businesses, so it is essential that attacks are contained and losses minimized. Effective resilience engineering at the organizational and systems levels can repel attacks and bring systems back into operation quickly and so limit the losses incurred.

In Chapter 13, where I discussed security engineering, I introduced concepts that are fundamental to resilience planning. Some of these concepts are:

1. *Assets,* which are systems and data that have to be protected. Some assets are more valuable than others and so require a higher level of protection.
2. *Threats,* which are circumstances that can cause harm by damaging or stealing organizational IT infrastructure or system assets.
3. *Attacks,* which are manifestations of a threat where an attacker aims to damage or steal IT assets, such as websites or personal data.

Three types of threats have to be considered in resilience planning:

1. *Threats to the confidentiality of assets* In this case, data is not damaged, but it is made available to people who should not have access to it. An example of a threat to confidentiality is when a credit card database held by a company is stolen, with the potential for illegal use of card information.
2. *Threats to the integrity of assets* These are threats where systems or data are damaged in some way by a cyberattack. This may involve introducing a virus or a worm into software or corrupting organizational databases.
3. *Threats to the availability of assets* These are threats that aim to deny use of assets by authorized users. The best-known example is a denial-of-service attack that aims to take down a website and so make it unavailable for external use.

These are not independent threat classes. An attacker may compromise the integrity of a user's system by introducing malware, such as a botnet component. This may then be invoked remotely as part of a distributed denial-of-service attack on another system. Other types of malware may be used to capture personal details and so allow confidential assets to be accessed.

To counter these threats, organizations should put controls in place that make it difficult for attackers to access or damage assets. It is also important to raise awareness of cybersecurity issues so that people know why these controls are important and so are less likely to reveal information to an attacker.

Examples of controls that may be used are:

1. *Authentication,* where users of a system have to show that they are authorized to access the system. The familiar login/password approach to authentication is a universally used but rather weak control.
2. *Encryption,* where data is algorithmically scrambled so that an unauthorized reader cannot access the information. Many companies now require that laptop disks are encrypted. If the computer is lost or stolen, this reduces the likelihood that the confidentiality of the information will be breached.
3. *Firewalls,* where incoming network packets are examined, then accepted or rejected according to a set of organizational rules. Firewalls can be used to

ensure that only traffic from trusted sources is allowed to pass from the external Internet into the local organizational network.

A set of controls in an organization provides a layered protection system. An attacker has to get through all of the protection layers for the attack to succeed. However, there is a trade-off between protection and efficiency. As the number of layers of protection increases, the system slows down. The protection systems consume an increasing amount of memory and processor resources, leaving less available to do useful work. The more security, the more inconvenient it is for users and the more likely that they will adopt insecure practices to increase system usability.

As with other aspects of system dependability, the fundamental means of protecting against cyberattacks depends on redundancy and diversity. Recall that redundancy means having spare capacity and duplicated resources in a system. Diversity means that different types of equipment, software, and procedures are used so that common failures are less likely to occur across a number of systems. Examples of where redundancy and diversity are valuable for cyber-resilience are:

1. For each system, copies of data and software should be maintained on separate computer systems. Shared disks should be avoided if possible. This supports recovery after a successful cyberattack (recovery and reinstatement).
2. Multi-stage diverse authentication can protect against password attacks. As well as login/password authentication, additional authentication steps may be involved that require users to provide some personal information or a code generated by their mobile device (resistance).
3. Critical servers may be overprovisioned; that is, they may be more powerful than is required to handle their expected load. The spare capacity means that attacks may be resisted without necessarily degrading the normal response of the server. Furthermore, if other servers are damaged, spare capacity is available to run their software while they are being repaired (resistance and recovery).

Planning for cybersecurity has to be based on assets and controls and the 4 Rs of resilience engineering—recognition, resistance, recovery, and reinstatement. Figure 14.2 shows a planning process that may be followed. The key stages in this process are:

1. *Asset classification* The organization's hardware, software, and human assets are examined and classified depending on how essential they are to normal operations. They may be classed as critical, important, or useful.
2. *Threat identification* For each of the assets (or at least the critical and important assets), you should identify and classify threats to that asset. In some cases, you may try to estimate the probability that a threat will arise, but such estimates are often inaccurate as you don't have enough information about potential attackers.
3. *Threat recognition* For each threat or, sometimes asset/threat pair, you should identify how an attack based on that threat might be recognized. You may

Figure 14.2 Cyber-resilience planning

decide that additional software needs to be bought or written for threat recognition or that regular checking procedures are put in place.

4. *Threat resistance* For each threat or asset/threat pair, you should identify possible resistance strategies. These either may be embedded in the system (technical strategies) or may rely on operational procedures. You may also need to think of threat neutralization strategies so that the threat does not recur.

5. *Asset recovery* For each critical asset or asset/threat pair, you should work out how that asset could be recovered in the event of a successful cyberattack. This may involve making extra hardware available or changing backup procedures to make it easier to access redundant copies of data.

6. *Asset reinstatement* This is a more general process of asset recovery where you define procedures to bring the system back into normal operation. Asset reinstatement should be concerned with all assets and not simply assets that are critical to the organization.

Information about all of these stages should be maintained in a cyber-resilience plan. This plan should be regularly updated, and, wherever possible, the strategies identified should be tested in mock attacks on the system.

Another important part of cyber-resilience planning is to decide how to support a flexible response in the event of a cyberattack. Paradoxically, resilience and security requirements often conflict. The aim of security is usually to limit privilege as far as possible so that users can only do what the security policy of the organization allows. However, to deal with problems, a user or system operator may have to take the initiative and take actions that are normally carried out by someone with a higher level of privilege.

For example, the system manager of a medical system may not normally be allowed to change the access rights of medical staff to records. For security reasons, access permissions have to be formally authorized, and two people need to be involved in making the change. This reduces the chances of system managers colluding with attackers and allowing access to confidential medical information.

Now, imagine that the system manager notices that a logged-in user is accessing a large number of records outside of normal working hours. The manager suspects

that an account has been compromised and that the user accessing the records is not actually the authorized user. To limit the damage, the user's access rights should be removed and a check then made with the authorized user to see if the accesses were actually illegal. However, the security procedures limiting the rights of system managers to change users' permissions make this impossible.

Resilience planning should take such situations into account. One way of doing so is to include an "emergency" mode in systems where normal checks are ignored. Rather than forbidding operations, the system logs what has been done and who was responsible. Therefore, the audit trail of emergency actions can be used to check that a system manager's actions were justified. Of course, there is scope for misuse here, and the existence of an emergency mode is itself a potential vulnerability. Therefore, organizations have to trade off possible losses against the benefits of adding more features to a system to support resilience.

14.2 Sociotechnical resilience

Fundamentally, resilience engineering is a sociotechnical rather than a technical activity. As I explained in Chapter 10, a sociotechnical system includes hardware, software, and people and is influenced by the culture, policies, and procedures of the organization that owns and uses the system. To design a resilient system, you have to think about sociotechnical systems design and not exclusively focus on software. Resilience engineering is concerned with adverse external events that can lead to system failure. Dealing with these events is often easier and more effective in the broader sociotechnical system.

For example, the Mentcare system maintains confidential patient data, and a possible external cyberattack may aim to steal that data. Technical safeguards such as authentication and encryption may be used to protect the data, but these are not effective if an attacker has access to the credentials of a genuine system user. You could try to solve this problem at the technical level by using more complex authentication procedures. However, these procedures annoy users and may lead to vulnerabilities as they write down authentication information. A better strategy may be to introduce organizational policies and procedures that emphasize the importance of not sharing login credentials and that tell users about easy ways to create and maintain strong passwords.

Resilient systems are flexible and adaptable so that they can cope with the unexpected. It is very difficult to create software that can adapt to cope with problems that have not been anticipated. However, as we saw from the Apollo 13 accident, people are very good at this. Therefore, to achieve resilience, you should take advantage of the fact that people are an inherent part of sociotechnical systems. Rather than try to anticipate and deal with all problems in software, you should leave some types of problem solving to the people responsible for operating and managing the software system.

To understand why you should leave some types of problem solving to people, you have to consider the hierarchy of sociotechnical systems that includes technical, software-intensive systems. Figure 14.3 shows that technical systems S1 and S2 are

Figure 14.3 Nested technical and sociotechnical systems

part of a broader sociotechnical system ST1. That sociotechnical system includes operators who monitor the condition of S1 and S2 and who can take actions to resolve problems in these systems. If system S1 (say) fails, then the operators in ST1 may detect that failure and take recovery actions before the software failure leads to failure in the broader sociotechnical system. Operators may also invoke recovery and reinstatement procedures to get S1 back to its normal operating state.

Operational and management processes are the interface between the organization and the technical systems that are used. If these processes are well designed, they allow people to discover and to cope with technical system failures, as well as ensuring that operator errors are minimized. As I discuss in Section 14.2.2, rigid processes that are overautomated are not inherently resilient. They do not allow people to use their skills and knowledge to adapt and change processes to cope with the unexpected and deal with unanticipated failures.

The system ST1 is one of a number of sociotechnical systems in the organization. If the system operators cannot contain a technical system failure, then this may lead to a failure in the sociotechnical system ST1. Managers at the organizational level then must detect the problem and take steps to recover from it. Resilience is therefore an organizational as well as a system characteristic.

Hollnagel (Hollnagel 2010), who was an early advocate of resilience engineering, argues that it is important for organizations to study and learn from successes as well as failure. High-profile safety and security failures lead to inquiries and changes in practice and procedures. However, rather than respond to these failures, it is better to avoid them by observing how people deal with problems and maintain resilience. This good practice can then be disseminated throughout the organization. Figure 14.4 shows four characteristics that Hollnagel suggests reflect the resilience of an organization. These characteristics are:

1. *The ability to respond* Organizations have to be able to adapt their processes and procedures in response to risks. These risks may be anticipated risks, or they may be detected threats to the organization and its systems. For example, if a new security threat is detected and publicized, a resilient organization can make changes quickly so that this threat does not disrupt its operations.

2. *The ability to monitor* Organizations should monitor both their internal operations and their external environment for threats before they arise. For example, a company should monitor how its employees follow security policies.

Figure 14.4 Characteristics of resilient organizations

If potentially insecure behavior is detected, the company should respond by taking actions to understand why this has occurred and to change employee behavior.

3. *The ability to anticipate* A resilient organization should not simply focus on its current operations but should anticipate possible future events and changes that may affect its operations and resilience. These events may include technological innovations, changes in regulations or laws, and modifications in customer behavior. For example, wearable technology is starting to become available, and companies should now be thinking about how this might affect their current security policies and procedures.

4. *The ability to learn* Organizational resilience can be improved by learning from experience. It is particularly important to learn from successful responses to adverse events such as the effective resistance of a cyberattack. Learning from success allows good practice to be disseminated throughout the organization.

As Hollnagel says, to become resilient organizations have to address all of these issues to some extent. Some will focus more on one quality than others. For example, a company running a large-scale data center may focus mostly on monitoring and responsiveness. However, a digital library that manages long-term archival information may have to anticipate how future changes may affect its business as well as respond to any immediate security threats.

14.2.1 Human error

Early work on resilience engineering was concerned with accidents in safety-critical systems and with how the behavior of human operators could lead to safety-related system failures. This led to an understanding of system defenses that is equally applicable to systems that have to withstand malicious as well as accidental human actions.

We know that people make mistakes, and, unless a system is completely automated, it is inevitable that users and system operators will sometimes do the wrong thing. Unfortunately, these human errors sometimes lead to serious system failures. Reason (Reason, 2000) suggests that the problem of human error can be viewed in two ways:

1. *The person approach* Errors are considered to be the responsibility of the individual and "unsafe acts" (such as an operator failing to engage a safety barrier)

are a consequence of individual carelessness or reckless behavior. People who adopt this approach believe that human errors can be reduced by threats of disciplinary action, more stringent procedures, retraining, and so on. Their view is that the error is the fault of the individual responsible for making the mistake.

2. *The systems approach* The basic assumption is that people are fallible and will make mistakes. People make mistakes because they are under pressure from high workloads, because of poor training, or because of inappropriate system design. Good systems should recognize the possibility of human error and include barriers and safeguards that detect human errors and allow the system to recover before failure occurs. When a failure does occur, the best way to avoid its recurrence is to understand how and why the system defenses did not trap the error. Blaming and punishing the person who triggered the failure does not improve long-term system safety.

I believe that the systems approach is the right one and that systems engineers should assume that human errors will occur during system operation. Therefore, to improve the resilience of a system, designers have to think about the defenses and barriers to human error that could be part of a system. They should also think about whether these barriers should be built into the technical components of the system. If not, they could be part of the processes, procedures, and guidelines for using the system. For example, two operators may be required to check critical system inputs.

The barriers and safeguards that protect against human errors may be technical or sociotechnical. For example, code to validate all inputs is a technical defense; an approval procedure for critical system updates that needs two people to confirm the update is a sociotechnical defense. Using diverse barriers means that shared vulnerabilities are less likely and that a user error is more likely to be trapped before system failure.

In general, you should use redundancy and diversity to create a set of defensive layers (Figure 14.5), where each layer uses a different approach to deter attackers or to trap component failures or human errors. Dark blue barriers are software checks; light blue barriers are checks carried out by people.

As an example of this approach to defense in depth, some of the checks for controller errors that may be part of an air traffic control system include:

1. *A conflict alert warning as part of an air traffic control system* When a controller instructs an aircraft to change its speed or altitude, the system extrapolates its trajectory to see if it intersects with any other aircraft. If so, it sounds an alarm.

2. *Formalized recording procedures for air traffic management* The same ATC system may have a clearly defined procedure setting out how to record the control instructions that have been issued to aircraft. These procedures help controllers check if they have issued the instruction correctly and make the information visible to others for checking.

3. *Collaborative checking* Air traffic control involves a team of controllers who constantly monitor each other's work. When a controller makes a mistake, others usually detect and correct it before an incident occurs.

Figure 14.5 Defensive layers

Reason (Reason 2000) draws on the idea of defensive layers in a theory of how human errors lead to system failures. He introduces the so-called Swiss cheese model, which suggests that defensive layers are not solid barriers but are instead like slices of Swiss cheese. Some types of Swiss cheese, such as Emmenthal, have holes of varying sizes in them. Reason suggests that vulnerabilities, or what he calls latent conditions in the layers, are analogous to these holes.

These latent conditions are not static—they change depending on the state of the system and the people involved in system operation. To continue with the analogy, the holes change size and move around the defensive layers during system operation. For example, if a system relies on operators checking each other's work, a possible vulnerability is that both make the same mistake. This is unlikely under normal conditions so, in the Swiss cheese model, the hole is small. However, when the system is heavily loaded and the workload of both operators is high, then mistakes are more likely. The size of the hole representing this vulnerability increases.

Failure in a system with layered defenses occurs when there is some external trigger event that has the potential to cause damage. This event might be a human error (which Reason calls an active failure) or it could be a cyberattack. If all of the defensive barriers fail, then the system as a whole will fail. Conceptually, this corresponds to the holes in the Swiss cheese slices lining up, as shown in Figure 14.6.

This model suggests that different strategies can be used to increase system resilience to adverse external events:

1. Reduce the probability of the occurrence of an external event that might trigger system failures. To reduce human errors, you may introduce improved training for operators or give operators more control over their workload so that they are not overloaded. To reduce cyberattacks, you may reduce the number of people who have privileged system information and so reduce the chances of disclosure to an attacker.

2. Increase the number of defensive layers. As a general rule, the more layers that you have in a system, the less likely it is that the holes will line up and a system failure will occur. However, if these layers are not independent, then they may share a common vulnerability. Thus, the barriers are likely to have the same "hole" in the same place, so there is only a limited benefit in adding a new layer.

Figure 14.6 Reason's Swiss cheese model of system failure

3. Design a system so that diverse types of barriers are included. This means that the "holes" will probably be in different places, and so there is less chance of the holes lining up and failing to trap an error.

4. Minimize the number of latent conditions in a system. Effectively, this means reducing the number and size of system "holes." However, this may significantly increase systems engineering costs. Reducing the number of bugs in the system increases testing and V & V costs. Therefore, this option may not be cost-effective.

In designing a system, you need to consider all of these options and make choices about what might be the most cost-effective ways to improve the system's defenses. If you are building custom software, then using software checking to increase the number and diversity of layers may be the best option. However, if you are using off-the-shelf software, then you may have to consider how sociotechnical defenses may be added. You may decide to change training procedures to reduce the chances of problems occurring and to make it easier to deal with incidents when they arise.

14.2.2 Operational and management processes

All software systems have associated operational processes that reflect the assumptions of the designers about how these systems will be used. Some software systems, particularly those that control or are interfaced to special equipment, have trained operators who are an intrinsic part of the control system. Decisions are made during the design stage about which functions should be part of the technical system and which functions should be the operator's responsibility. For example, in an imaging system in a hospital, the operator may have the responsibility of checking the quality of the images immediately after they have been processed. This check allows the imaging procedure to be repeated if there is a problem.

Operational processes are the processes that are involved in using the system for its defined purpose. For example, operators of an air traffic control system follow specific processes when aircraft enter and leave airspace, when they have to change height or speed, when an emergency occurs, and so on. For new systems, these operational processes have to be defined and documented during the system development process. Operators may have to be trained and other work processes adapted to make effective use of the new system.

Most software systems, however, do not have trained operators but have system users, who use the system as part of their work or to support their personal interests. For personal systems, the designers may describe the expected use of the system but have no control over how users will actually behave. For enterprise IT systems, however, training may be provided for users to teach them how to use the system. Although user behavior cannot be controlled, it is reasonable to expect that they will normally follow the defined process.

Enterprise IT systems will also usually have system administrators or managers who are responsible for maintaining that system. While they are not part of the business process supported by the system, their job is to monitor the software system for errors and problems. If problems arise, system managers take action to resolve them and restore the system to its normal operational state.

In the previous section, I discussed the importance of defense in depth and the use of diverse mechanisms to check for adverse events that could lead to system failure. Operational and management processes are an important defense mechanism, and, in designing a process, you need to find a balance between efficient operation and problem management. These are often in conflict as shown in Figure 14.7 as increasing efficiency removes redundancy and diversity from a system.

Over the past 25 years, businesses have focused on so-called process improvement. To improve the efficiency of operational and management processes, companies study how their processes are enacted and look for particularly efficient and inefficient practice. Efficient practice is codified and documented, and software may be developed to support this "optimum" process. Inefficient practice is replaced by more efficient ways of doing things. Sometimes process control mechanisms are introduced to ensure that system operators and managers follow this "best practice."

The problem with process improvement is that it often makes it harder for people to cope with problems. What seems to be "inefficient" practice often arises because people maintain redundant information or share information because they know this makes it easier to deal with problems when things go wrong. For example, air traffic controllers may print flight details as well as rely on the flight database because they will then have information about flights in the air if the system database becomes unavailable.

People have a unique capability to respond effectively to unexpected situations, even when they have never had direct experience of these situations. Therefore, when things go wrong, operators and system managers can often recover the situation, although they may sometimes have to break rules and "work around" the defined process. You should therefore design operational processes to be flexible and adaptable. The operational processes should not be too constraining; they should not require operations to be done in a particular order; and the system software should not rely on a specific process being followed.

For example, an emergency service control room system is used to manage emergency calls and to initiate a response to these calls. The "normal" process of handling a call is to log the caller's details and then send a message to the appropriate emergency service giving details of the incident and the address. This procedure provides an audit trail of the actions taken. A subsequent investigation can check that the emergency call has been properly handled.

Efficient process operation	Problem management
Process optimization and control	Process flexibility and adaptability
Information hiding and security	Information sharing and visibility
Automation to reduce operator workload with fewer operators and managers	Manual processes and spare operator/manager capacity to deal with problems
Role specialization	Role sharing

Figure 14.7 Efficiency and resilience

Now imagine that this system is subject to a denial-of-service attack, which makes the messaging system unavailable. Rather than simply not responding to calls, the operators may use their personal mobile phones and their knowledge of call responders to call the emergency service units directly so that they can respond to serious incidents.

Management and provision of information are also important for resilient operation. To make a process more efficient, it may make sense to present operators with the information they need, when they need it. From a security perspective, information should not be accessible unless the operator or manager needs that information. However, a more liberal approach to information access can improve system resilience.

If operators are only presented with information that the process designer thinks they "need to know," then they may be unable to detect problems that do not directly affect their immediate tasks. When things go wrong, the system operators do not have a broad picture of what is happening in the system, so it is more difficult for them to formulate strategies for dealing with problems. If they cannot access some information in the system for security reasons, then they may be unable to stop attacks and repair the damage that has been caused.

Automating the system management process means that a single manager may be able to manage a large number of systems. Automated systems can detect common problems and take actions to recover from these problems. Fewer people are needed for system operations and management, and so costs are reduced. However, process automation has two disadvantages:

1. Automated management systems may go wrong and take incorrect actions. As problems develop, the system may take unexpected actions that make the situation worse and that cannot be understood by the system managers.

2. Problem solving is a collaborative process. If fewer managers are available, it is likely to take longer to work out a strategy to recover from a problem or cyberattack.

Therefore, process automation can have both positive and negative effects on system resilience. If the automated system works properly, it can detect problems, invoke cyberattack resistance if necessary, and start automated recovery procedures. However, if the automated system can't handle the problem, fewer people will be available to tackle the problem and the system may have been damaged by the process automation doing the wrong thing.

In an environment where there are different types of system and equipment, it may be impractical to expect all operators and managers to be able to deal with all of

the different systems. Individuals may therefore specialize so that they become expert and knowledgeable about a small number of systems. This leads to more efficient operation but has consequences for the resilience of the system.

The problem with role specialization is that there may not be anyone available at a particular time who understands the interactions between systems. Consequently, it is difficult to cope with problems if the specialist is not available. If people work with several systems, they come to understand the dependencies and relationships between them and so can tackle problems that affect more than one system. With no specialist available, it becomes much more difficult to contain the problem and repair any damage that has been caused.

You may use risk assessment, as discussed in Chapter 13, to help make decisions on the balance between process efficiency and resilience. You consider all of the risks where operator or manager intervention may be required and assess the likelihood of these risks and the extent of the possible losses that might arise. For risks that may lead to serious damage and extensive loss and for risks that are likely to occur, you should favor resilience over process efficiency.

14.3 Resilient systems design

Resilient systems can resist and recover from adverse incidents such as software failures and cyberattacks. They can deliver critical services with minimal interruptions and can quickly return to their normal operating state after an incident has occurred. In designing a resilient system, you have to assume that system failures or penetration by an attacker will occur, and you have to include redundant and diverse features to cope with these adverse events.

Designing systems for resilience involves two closely related streams of work:

1. *Identifying critical services and assets* Critical services and assets are those elements of the system that allow a system to fulfill its primary purpose. For example, the primary purpose of a system that handles ambulance dispatch in response to emergency calls is to get help to people who need it as quickly as possible. The critical services are those concerned with taking calls and dispatching ambulances to the medical emergency. Other services such as call logging and ambulance tracking are less important.

2. *Designing system components that support problem recognition, resistance, recovery, and reinstatement* For example, in an ambulance dispatch system, a watchdog timer (see Chapter 12) may be included to detect if the system is not responding to events. Operators may have to authenticate with a hardware token to resist the possibility of unauthorized access. If the system fails, calls may be diverted to another center so that the essential services are maintained. Copies of the system database and software on alternative hardware may be maintained to allow for reinstatement after an outage.

Figure 14.8 Stages in survivability analysis

The fundamental notions of recognition, resistance, and recovery were the basis of early work in resilience engineering by Ellison et al. (Ellison et al. 1999, 2002). They designed a method of analysis called survivable systems analysis. This method is used to assess vulnerabilities in systems and to support the design of system architectures and features that promote system survivability.

Survivable systems analysis is a four-stage process (Figure 14.8) that analyzes the current or proposed system requirements and architecture, identifies critical services, attack scenarios, and system "softspots," and proposes changes to improve the survivability of a system. The key activities in each of these stages are as follows:

1. *System understanding* For an existing or proposed system, review the goals of the system (sometimes called the mission objectives), the system requirements, and the system architecture.
2. *Critical service identification* The services that must always be maintained and the components that are required to maintain these services are identified.
3. *Attack simulation* Scenarios or use cases for possible attacks are identified, along with the system components that would be affected by these attacks.
4. *Survivability analysis* Components that are both essential and compromisable by an attack are identified, and survivability strategies based on resistance, recognition, and recovery are identified.

The fundamental problem with this approach to survivability analysis is that its starting point is the requirements and architecture documentation for a system. This is a reasonable assumption for defense systems (the work was sponsored by the U.S. Department of Defense), but it poses two problems for business systems:

1. It is not explicitly related to the business requirements for resilience. I believe that these are a more appropriate starting point than technical system requirements.

Figure 14.9
Resilience engineering

2. It assumes that there is a detailed requirements statement for a system. In fact, resilience may have to be "retrofitted" to a system where there is no complete or up-to-date requirements document. For new systems, resilience may itself be a requirement, or systems may be developed using an agile approach. The system architecture may be designed to take resilience into account.

A more general resilience engineering method, as shown in Figure 14.9, takes the lack of detailed requirements into account as well as explicitly designing recovery and reinstatement into the system. For the majority of components in a system, you will not have access to their source code and will not be able to make changes to them. Your strategy for resilience has to be designed with this limitation in mind.

There are five interrelated streams of work in this approach to resilience engineering:

1. You identify business resilience requirements. These requirements set out how the business as a whole must maintain the services that it delivers to customers and, from this, resilience requirements for individual systems are developed. Providing resilience is expensive, and it is important not to overengineer systems with unnecessary resilience support.
2. You plan how to reinstate a system or a set of systems to their normal operating state after an adverse event. This plan has to be integrated with the business's

normal backup and archiving strategy that allows recovery of information after a technical or human error. It should also be part of a wider disaster recovery strategy. You have to take account of the possibility of physical events such as fire and flooding and study how to maintain critical information in separate locations. You may decide to use cloud backups for this plan.

3. You identify system failures and cyberattacks that can compromise a system, and you design recognition and resilience strategies to cope with these adverse events.
4. You plan how to recover critical services quickly after they have been damaged or taken offline by a failure or cyberattack. This step usually involves providing redundant copies of the critical assets that provide these services and switching to these copies when required.
5. Critically, you should test all aspects of your resilience planning. This testing involves identifying failure and attack scenarios and playing these scenarios out against your system.

Maintaining the availability of critical services is the essence of resilience. Accordingly, you have to know:

- the system services that are the most critical for a business,
- the minimal quality of service that must be maintained,
- how these services might be compromised,
- how these services can be protected, and
- how you can recover quickly if the services become unavailable.

As part of the analysis of critical services, you have to identify the system assets that are essential for delivering these services. These assets may be hardware (servers, network, etc.), software, data, and people. To build a resilient system, you have to think about how to use redundancy and diversity to ensure that these assets remain available in the event of a system failure.

For all of these activities, the key to providing a rapid response and recovery plan after an adverse event is to have additional software that supports resistance, recovery, and reinstatement. This may be commercial security software or resilience support that is programmed into application systems. It may also include scripts and specially written programs that are developed for recovery and reinstatement. If you have the right support software, the processes of recovery and reinstatement can be partially automated and quickly invoked and executed after a system failure.

Resilience testing involves simulating possible system failures and cyberattacks to test whether the resilience plans that have been drawn up work as expected. Testing is essential because we know from experience that the assumptions made in resilience planning are often invalid and that planned actions do not always work. Testing for resilience can reveal these problems so that the resilience plan can be refined.

Figure 14.10 The client–server architecture of the Mentcare system

Testing can be very difficult and expensive as, obviously, the testing cannot be carried out on an operational system. The system and its environment may have to be duplicated for testing, and staff may have to be released from their normal responsibilities to work on the test system. To reduce costs, you can use "desk testing." The testing team assumes a problem has occurred and tests their reactions to it; they do not simulate that problem on a real system. While this approach can provide useful information about system resilience, it is less effective than testing in discovering deficiencies in the resilience plan.

As an example of this approach, let us look at resilience engineering for the Mentcare system. To recap, this system is used to support clinicians treating patients in a variety of locations who have mental health problems. It provides patient information and records of consultations with doctors and specialist nurses. It includes a number of checks that can flag patients who may be potentially dangerous or suicidal. Figure 14.10 shows the architecture of this system.

The system is consulted by doctors and nurses before and during a consultation, and patient information is updated after the consultation. To ensure the effectiveness of clinics, the business resilience requirements are that the critical system services are available during normal working hours, that the patient data should not be permanently damaged or lost by a system failure or cyberattack, and that patient information should not be released to unauthorized people.

Two critical services in the system have to be maintained:

1. *An information service* that provides information about a patient's current diagnosis and treatment plan.
2. *A warning service* that highlights patients who could pose a danger to others or to themselves.

Notice that the critical service is not the availability of the complete patient record. Doctors and nurses only need to go back to previous treatments occasionally,

so clinical care is not seriously affected if a full record is not available. Therefore, it is possible to deliver effective care using a summary record that only includes information about the patient and recent treatment.

The assets required to deliver these services in normal system operations are:

1. The patient record database that maintains all patient information.
2. A database server that provides access to the database for local client computers.
3. A network for client/server communications.
4. Local laptop or desktop computers used by clinicians to access patient information.
5. A set of rules that identify patients who are potentially dangerous and that can flag patient records. Client software highlights dangerous patients to system users.

To plan recognition, resistance, and recovery strategies, you need to develop a set of scenarios that anticipate adverse events that might compromise the critical services offered by the system. Examples of these adverse events are:

1. The unavailability of the database server either through a system failure, a network failure, or a denial-of-service cyberattack.
2. The deliberate or accidental corruption of the patient record database or the rules that define what is meant by a "dangerous patient."
3. Infection of client computers with malware.
4. Access to client computers by unauthorized people who gain access to patient records.

Figure 14.11 shows possible recognition and resistance strategies for these adverse events. Notice that these are not just technical approaches but also include workshops to inform system users about security issues. We know that many security breaches arise because users inadvertently reveal privileged information to an attacker and these workshops reduce the chances of this happening. I don't have space here to discuss all of the options that I identified in Figure 14.11. Instead, I focus on how the system architecture can be modified to be more resilient.

In Figure 14.11, I suggested that maintaining patient information on client computers was a possible redundancy strategy that could help maintain critical services. This leads to the modified software architecture shown in Figure 14.12. The key features of this architecture are:

1. *Summary patient records that are maintained on local client computers* The local computers can communicate directly with each other and exchange information using either the system network or, if necessary, an ad hoc network created using mobile phones. Therefore, if the database is unavailable, doctors and nurses can still access essential patient information. (resistance and recovery)
2. *A backup server to allow for main server failure* This server is responsible for taking regular snapshots of the database as backups. In the event the main server

Event	Recognition	Resistance
Server unavailability	1. Watchdog timer on client that times out if no response to client access 2. Text messages from system managers to clinical users	1. Design system architecture to maintain local copies of critical information 2. Provide peer-to-peer search across clients for patient data 3. Provide staff with smartphones that can be used to access the network in the event of server failure 4. Provide backup server
Patient database corruption	1. Record level cryptographic checksums 2. Regular auto-checking of database integrity 3. Reporting system for incorrect information	1. Replayable transaction log to update database backup with recent transactions 2. Maintenance of local copies of patient information and software to restore database from local copies and backups
Malware infection of client computers	1. Reporting system so that computer users can report unusual behavior 2. Automated malware checks on startup	1. Security awareness workshops for all system users 2. Disabling of USB ports on client computers 3. Automated system setup for new clients 4. Support access to system from mobile devices 5. Installation of security software
Unauthorized access to patient information	1. Warning text messages from users about possible intruders 2. Log analysis for unusual activity	1. Multilevel system authentication process 2. Disabling of USB ports on client computers 3. Access logging and real-time log analysis 4. Security awareness workshops for all system users

Figure 14.11 Recognition and resistance strategies for adverse events

fails, it can also act as the main server for the whole system. This provides continuity of service and recovery after a server failure (resistance and recovery).

3. *Database integrity checking and recovery software* Integrity checking runs as a background task checking for signs of database corruption. If corruption is discovered, it can automatically initiate the recovery of some or all of the data from backups. The transaction log allows these backups to be updated with details of recent changes (recognition and recovery).

To maintain the key services of patient information access and staff warning, we can make use of the inherent redundancy in a client-server system. By downloading information to the client at the start of a clinic session, the consultation can continue without server access. Only the information about the patients who are scheduled to attend consultations that day needs to be downloaded. If there is a need to access other patient information and the server is unavailable, then other client computers may be contacted using peer-to-peer communication to see if the information is available on them.

The service that provides a warning to staff of patients who may be dangerous can easily be implemented using this approach. The records of patients who may harm themselves or others are identified before the download process. When clinical staff access these records, the software can highlight the records to indicate the patients that require special care.

Figure 14.12 An architecture for Mentcare system resilience

The features in this architecture that support the resistance to adverse events are also useful in supporting recovery from these events. By maintaining multiple copies of information and having backup hardware available, critical system services can be quickly restored to normal operation. Because the system need only be available during normal working hours (say, 8 a.m to 6 p.m), the system can be reinstated overnight so that it is available for the following day after a failure.

As well as maintaining critical services, the other business requirements of maintaining the confidentiality and integrity of patient data must also be supported. The architecture shown in Figure 14.12 includes a backup system and explicit database integrity checking to reduce the chances that patient information is damaged accidentally or in a malicious attack. Information on client computers is also available and can be used to support recovery from data corruption or damage.

While maintaining multiple copies of data is a safeguard against data corruption, it poses a risk to confidentiality as all of these copies have to be secured. In this case, this risk can be controlled by:

1. Only downloading the summary records of patients who are scheduled to attend a clinic. This limits the number of records that could be compromised.
2. Encrypting the disk on local client computers. Attackers who do not have the encryption key cannot read the disk if they gain access to the computer.
3. Securely deleting the downloaded information at the end of a clinic session. This further reduces the chances of an attacker gaining access to confidential information.

4. Ensuring that all network transactions are encrypted. If an attacker intercepts these transactions, they cannot get access to the information.

Because of performance degradation, it is probably impractical to encrypt the entire patient database on the server. Strong authentication should therefore be used to protect this information.

KEY POINTS

- The resilience of a system is a judgment of how well that system can maintain the continuity of its critical services in the presence of disruptive events, such as equipment failure and cyberattacks.
- Resilience should be based on the 4 Rs model—recognition, resistance, recovery, and reinstatement.
- Resilience planning should be based on the assumption that networked systems will be subject to cyberattacks by malicious insiders and outsiders and that some of these attacks will be successful.
- Systems should be designed with a number of defensive layers of different types. If these layers are effective, human and technical failures can be trapped and cyberattacks resisted.
- To allow system operators and managers to cope with problems, processes should be flexible and adaptable. Process automation can make it more difficult for people to cope with problems.
- Business resilience requirements should be the starting point for designing systems for resilience. To achieve system resilience, you have to focus on recognition and recovery from problems, recovery of critical services and assets, and reinstatement of the system.
- An important part of design for resilience is identifying critical services, which are those services that are essential if a system is to ensure its primary purpose. Systems should be designed so that these services are protected and, in the event of failure, recovered as quickly as possible.

FURTHER READING

"Survivable Network System Analysis: A Case Study." An excellent paper that introduces the notion of system survivability and uses a case study of a mental health record treatment system to illustrate the application of a survivability method. (R. J. Ellison, R. C. Linger, T. Longstaff, and N. R. Mead, *IEEE Software*, 16 (4), July/August 1999) http://dx.doi.org/10.1109/52.776952

Resilience Engineering in Practice: A Guidebook. This is a collection of articles and case studies on resilience engineering that takes a broad, sociotechnical systems perspective. (E. Hollnagel, J. Paries, D. W. Woods, and J. Wreathall, Ashgate Publishing Co., 2011).

"Cyber Risk and Resilience Management." This is a website with a wide range of resources on cybersecurity and resilience, including a model for resilience management. (Software Engineering Institute, 2013) https://www.cert.org/resilience/

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/security-and-resilience/

EXERCISES

14.1. Explain how the complementary strategies of resistance, recognition, recovery, and reinstatement may be used to provide system resilience.

14.2. What are the types of threats that have to be considered in resilience planning? Provide examples of the controls that organizations should put in place to counter those threats.

14.3. Describe the ways in which human error can be viewed according to Reason (Reason, 2000) and the strategies that can be used to increase resilience according to the Swiss cheese model (Figure 14.6).

14.4. A hospital proposes to introduce a policy that any member of clinical staff (doctors or nurses) who takes or authorizes actions that leads to a patient being injured will be subject to criminal charges. Explain why this is a bad idea, which is unlikely to improve patient safety, and why it is likely to adversely affect the resilience of the organization.

14.5. What is survivable systems analysis and what are the key activities in each of the four stages involved in it as shown in Figure 14.8?

14.6. Explain why process inflexibility can inhibit the ability of a sociotechnical system to resist and recover from adverse events such as cyberattacks and software failure. If you have experience of process inflexibility, illustrate your answer with examples from your experience.

14.7. Suggest how the approach to resilience engineering that I proposed in Figure 14.9 could be used in conjunction with an agile development process for the software in the system. What problems might arise in using agile development for systems where resilience is important?

14.8. In Section 13.4.2, (1) an unauthorized user places malicious orders to move prices and (2) an intrusion corrupts the database of transactions that have taken place. For each of these cyberattacks, identify resistance, recognition, and recovery strategies that might be used.

14.9. In Figure 14.11, I suggested a number of adverse events that could affect the Mentcare system. Draw up a test plan for this system that sets out how you could test the ability of the Mentcare system to recognize, resist, and recover from these events.

14.10. A senior manager in a company is concerned about insider attacks from disaffected staff on the company's IT assets. As part of a resilience improvement program, she proposes that a logging system and data analysis software be introduced to capture and analyze all employee actions but that employees should not be told about this system. Discuss the ethics of both introducing a logging system and doing so without telling system users.

REFERENCES

Ellison, R. J., R. C. Linger, T. Longstaff, and N. R. Mead. 1999. "Survivable Network System Analysis: A Case Study." *IEEE Software* 16 (4): 70–77. doi:10.1109/52.776952.

Ellison, R. J., R. C. Linger, H. Lipson, N. R. Mead, and A. Moore. 2002. "Foundations of Survivable Systems Engineering." *Crosstalk: The Journal of Defense Software Engineering* 12: 10–15. http://resources.sei.cmu.edu/asset_files/WhitePaper/2002_019_001_77700.pdf

Hollnagel, E. 2006. "Resilience—the Challenge of the Unstable." In *Resilience Engineering: Concepts and Precepts*, edited by E. Hollnagel, D. D. Woods, and N.G. Leveson, 9–18.

———. 2010. "RAG—The Resilience Analysis Grid." In *Resilience Engineering in Practice*, edited by E. Hollnagel, J. Paries, D. Woods, and J. Wreathall, 275–295. Farnham, UK: Ashgate Publishing Group.

InfoSecurity. 2013. "Global Cybercrime, Espionage Costs $100–$500 Billion Per Year." http://www.infosecurity-magazine.com/view/33569/global-cybercrime-espionage-costs-100500-billion-per-year

Laprie, J-C. 2008. "From Dependability to Resilience." In *38th Int. Conf. on Dependable Systems and Networks*. Anchorage, Alaska. http://2008.dsn.org/fastabs/dsn08fastabs_laprie.pdf

Reason, J. 2000. "Human Error: Models and Management." *British Medical J.* 320: 768–770. doi:10.1136/bmj.320.7237.768.

PART 3 Advanced Software Engineering

This part of the book covers more advanced software engineering topics. I assume in these chapters that readers understand the basics of the discipline, covered in Chapters 1–9.

Chapters 15–18 focus on the dominant development paradigm for web-based information systems and enterprise systems—software reuse. Chapter 15 introduces the topic and explains the different types of reuse that are possible. I then cover the most common approach to reuse, which is the reuse of application systems. These are configured and adapted to the specific needs of each business.

Chapter 16 is concerned with the reuse of software components rather than entire software systems. In this chapter, I explain what is meant by a component and why standard component models are needed for effective component reuse. I also discuss the general process of component-based software engineering and the problems of component composition.

The majority of large systems are now distributed systems and Chapter 17 covers issues and problems of building distributed systems. I introduce the client-server approach as a fundamental paradigm of distributed systems engineering, and explain ways of implementing this architectural style. The final section explains software as a service–the delivery of software functionality over the Internet, which has changed the market for software products.

Chapter 18 introduces the related topic of service-oriented architectures, which link the notions of distribution and reuse. Services are reusable software components whose functionality can be accessed over the Internet. I discuss two widely-used approaches to service development namely SOAP-based and RESTful services. I explain what is involved in creating services (service engineering) and composing services to create new software systems.

The focus of Chapters 19–21 is systems engineering. In Chapter 19, I introduce the topic and explain why it is important that software engineers should understand systems engineering. I discuss the systems engineering life cycle and the importance of procurement in that life-cycle.

Chapter 20 covers systems of systems (SoS). The large systems that we will build in the 21st century will not be developed from scratch but will be created by integrating existing complex systems. I explain why an understanding of complexity is important in SoS development and discuss architectural patterns for complex systems of systems.

Most software systems are not apps or business systems but are embedded real-time systems. Chapter 21 covers this important topic. I introduce the idea of a real-time embedded system and describe architectural patterns that are used in embedded systems design. I then explain the process of timing analysis and conclude the chapter with a discussion of real-time operating systems.

15

Software reuse

Objectives

The objectives of this chapter are to introduce software reuse and to describe approaches to system development based on large-scale software reuse. When you have read this chapter, you will:

- understand the benefits and problems of reusing software when developing new systems;
- understand the concept of an application framework as a set of reusable objects and how frameworks can be used in application development;
- have been introduced to software product lines, which are made up of a common core architecture and reusable components that are configured for each version of the product;
- have learned how systems can be developed by configuring and composing off-the-shelf application software systems.

Contents

15.1 The reuse landscape
15.2 Application frameworks
15.3 Software product lines
15.4 Application system reuse

Reuse-based software engineering is a software engineering strategy where the development process is geared to reusing existing software. Until around 2000, systematic software reuse was uncommon, but it is now used extensively in the development of new business systems. The move to reuse-based development has been in response to demands for lower software production and maintenance costs, faster delivery of systems, and increased software quality. Companies see their software as a valuable asset. They are promoting reuse of existing systems to increase their return on software investments.

Reusable software of different kinds is now widely available. The open-source movement has meant that there is a huge code base that can be reused. This may be in the form of program libraries or entire applications. Many domain-specific application systems, such as ERP systems, are available that can be tailored and adapted to customer requirements. Some large companies provide a range of reusable components for their customers. Standards, such as web service standards, have made it easier to develop software services and reuse them across a range of applications.

Reuse-based software engineering is an approach to development that tries to maximize the reuse of existing software. The software units that are reused may be of radically different sizes. For example:

1. *System reuse* Complete systems, which may be made up of a number of application programs, may be reused as part of a system of systems (Chapter 20).
2. *Application reuse* An application may be reused by incorporating it without change into other systems or by configuring the application for different customers. Alternatively, application families or software product lines that have a common architecture, but that are adapted to individual customer requirements, may be used to develop a new system.
3. *Component reuse* Components of an application, ranging in size from subsystems to single objects, may be reused. For example, a pattern-matching system developed as part of a text-processing system may be reused in a database management system. Components may be hosted on the cloud or on private servers and may be accessible through an application programming interface (API) as services.
4. *Object and function reuse* Software components that implement a single function, such as a mathematical function, or an object class may be reused. This form of reuse, designed around standard libraries, has been common for the past 40 years. Many libraries of functions and classes are freely available. You reuse the classes and functions in these libraries by linking them with newly developed application code. In areas such as mathematical algorithms and graphics, where specialized, expensive expertise is needed to develop efficient objects and functions, reuse is particularly cost-effective.

All software systems and components that include generic functionality are potentially reusable. However, these systems or components are sometimes so

Benefit	Explanation
Accelerated development	Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time may be reduced.
Effective use of specialists	Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge.
Increased dependability	Reused software, which has been tried and tested in working systems, should be more dependable than new software. Its design and implementation faults should have been found and fixed.
Lower development costs	Development costs are proportional to the size of the software being developed. Reusing software means that fewer lines of code have to be written.
Reduced process risk	The cost of existing software is already known, while the costs of development are always a matter of judgment. This is an important factor for project management because it reduces the margin of error in project cost estimation. This is especially true when large software components such as subsystems are reused.
Standards compliance	Some standards, such as user interface standards, can be implemented as a set of reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users make fewer mistakes when presented with a familiar interface.

Figure 15.1 Benefits of software reuse

specific that it is very expensive to modify them for a new situation. Rather than reuse the code, however, you can reuse the ideas that are the basis of the software. This is called concept reuse.

In concept reuse you do not reuse a software component; rather, you reuse an idea, a way of working, or an algorithm. The concept that you reuse is represented in an abstract notation, such as a system model, which does not include implementation detail. It can, therefore, be configured and adapted for a range of situations. Concept reuse is embodied in approaches such as design patterns (Chapter 7), configurable system products, and program generators. When concepts are reused, the reuse process must include an activity where the abstract concepts are instantiated to create executable components.

An obvious advantage of software reuse is that overall development costs are lower. Fewer software components need to be specified, designed, implemented, and validated. However, cost reduction is only one benefit of software reuse. I have listed other advantages of reusing software in Figure 15.1.

However, there are costs and difficulties associated with reuse (Figure 15.2). There is a significant cost associated with understanding whether or not a component is suitable for reuse in a particular situation, and in testing that component to ensure its dependability. These additional costs mean that the savings in development costs may not be less than anticipated. However, the other benefits of reuse still apply.

Problem	Explanation
Creating, maintaining, and using a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Development processes have to be adapted to ensure that the library is used.
Finding, understanding, and adapting reusable components	Software components have to be discovered in a library, understood, and sometimes adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they include a component search as part of their normal development process.
Increased maintenance costs	If the source code of a reused software system or component is not available, then maintenance costs may be higher because the reused elements of the system may become incompatible with changes made to the system.
Lack of tool support	Some software tools do not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account. This is more likely to be the case for tools that support embedded systems engineering than for object-oriented development tools.
"Not-invented-here" syndrome	Some software engineers prefer to rewrite components because they believe they can improve on them. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.

Figure 15.2 Problems with software reuse

As I discussed in Chapter 2, software development processes have to be adapted to take reuse into account. In particular, there has to be a requirements refinement stage where the requirements for the system are modified to reflect the reusable software that is available. The design and implementation stages of the system may also include explicit activities to look for and evaluate candidate components for reuse.

15.1 The reuse landscape

Over the past 20 years, many techniques have been developed to support software reuse. These techniques exploit the facts that systems in the same application domain are similar and have potential for reuse, that reuse is possible at different levels from simple functions to complete applications, and that standards for reusable components facilitate reuse. Figure 15.3 shows the "reuse landscape"—different ways of implementing software reuse. Each of these approaches to reuse is briefly described in Figure 15.4.

Given this array of techniques for reuse, the key question is "which is the most appropriate technique to use in a particular situation?" Obviously, the answer to this question depends on the requirements for the system being developed, the technology

Figure 15.3 The reuse landscape

and reusable assets available, and the expertise of the development team. Key factors that you should consider when planning reuse are:

1. *The development schedule for the software* If the software has to be developed quickly, you should try to reuse complete systems rather than individual components. Although the fit to requirements may be imperfect, this approach minimizes the amount of development required.

2. *The expected software lifetime* If you are developing a long-lifetime system, you should focus on the maintainability of the system. You should not just think about the immediate benefits of reuse but also of the long-term implications.

   Over its lifetime, you will have to adapt the system to new requirements, which will mean making changes to parts of the system. If you do not have access to the source code of the reusable components, you may prefer to avoid off-the-shelf components and systems from external suppliers. These suppliers may not be able to continue support for the reused software. You may decide that it is safer to reuse open-source systems and components (Chapter 7) as this means you can access and keep copies of the source code.

3. *The background, skills and experience of the development team* All reuse technologies are fairly complex, and you need quite a lot of time to understand and use them effectively. Therefore, you should focus your reuse effort in areas where your development team has expertise.

4. *The criticality of the software and its non-functional requirements* For a critical system that has to be certified by an external regulator you may have to create a safety or security case for the system (discussed in Chapter 12). This is difficult if you don't have access to the source code of the software. If your software has stringent performance requirements, it may be impossible to use strategies such as model-driven engineering (MDE) (Chapter 5). MDE relies on generating code from a reusable domain-specific model of a system. However, the code generators used in MDE often generate relatively inefficient code.

Approach	Description
Application frameworks	Collections of abstract and concrete classes are adapted and extended to create application systems.
Application system integration	Two or more application systems are integrated to provide extended functionality.
Architectural patterns	Standard software architectures that support common types of application system are used as the basis of applications. Described in Chapters 6, 11, and 17.
Aspect-oriented software development	Shared components are woven into an application at different places when the program is compiled. Described in web Chapter 31.
Component-based software engineering	Systems are developed by integrating components (collections of objects) that conform to component-model standards. Described in Chapter 16.
Configurable application systems	Domain-specific systems are designed so that they can be configured to the needs of specific system customers.
Design patterns	Generic abstractions that occur across applications are represented as design patterns showing abstract and concrete objects and interactions. Described in Chapter 7.
ERP systems	Large-scale systems that encapsulate generic business functionality and rules are configured for an organization.
Legacy system wrapping	Legacy systems (Chapter 9) are "wrapped" by defining a set of interfaces and providing access to these legacy systems through these interfaces.
Model-driven engineering	Software is represented as domain models and implementation independent models, and code is generated from these models. Described in Chapter 5.
Program generators	A generator system embeds knowledge of a type of application and is used to generate systems in that domain from a user-supplied system model.
Program libraries	Class and function libraries that implement commonly used abstractions are available for reuse.
Service-oriented systems	Systems are developed by linking shared services, which may be externally provided. Described in Chapter 18.
Software product lines	An application type is generalized around a common architecture so that it can be adapted for different customers.
Systems of systems	Two or more distributed systems are integrated to create a new system. Described in Chapter 20.

Figure 15.4 Approaches that support software reuse

5. *The application domain* In many application domains, such as manufacturing and medical information systems, there are generic products that may be reused by configuring them to a local situation. This is one of the most effective approaches to reuse, and it is almost always cheaper to buy rather than build a new system.

Generator-based reuse

Generator-based reuse involves incorporating reusable concepts and knowledge into automated tools and providing an easy way for tool users to integrate specific code with this generic knowledge. This approach is usually most effective in domain-specific applications. Known solutions to problems in that domain are embedded in the generator system and selected by the user to create a new system.

http://software-engineering-book.com/web/generator-reuse/

6. *The platform on which the system will run* Some components models, such as .NET, are specific to Microsoft platforms. Similarly, generic application systems may be platform-specific, and you may only be able to reuse these if your system is designed for the same platform.

The range of available reuse techniques is such that, in most situations, there is the possibility of some software reuse. Whether or not reuse is achieved is often a managerial rather than a technical issue. Managers may be unwilling to compromise their requirements to allow reusable components to be used. They may not understand the risks associated with reuse as well as they understand the risks of original development. Although the risks of new software development may be higher, some managers may prefer known risks of development to unknown risks of reuse. To promote company-wide reuse, it may be necessary to introduce a reuse program that focuses on the creation of reusable assets and processes to facilitate reuse (Jacobsen, Griss, and Jonsson 1997).

15.2 Application frameworks

Early enthusiasts for object-oriented development suggested that one of the key benefits of using an object-oriented approach was that objects could be reused in different systems. However, experience has shown that objects are often too fine-grained and are often specialized for a particular application. It often takes longer to understand and adapt the object than to reimplement it. It has now become clear that object-oriented reuse is best supported in an object-oriented development process through larger-grain abstractions called frameworks.

As the name suggests, a framework is a generic structure that is extended to create a more specific subsystem or application. Schmidt et al. (Schmidt et al. 2004) define a framework to be

> *an integrated set of software artifacts (such as classes, objects and components) that collaborate to provide a reusable architecture for a family of related applications.*[†]

Frameworks provide support for generic features that are likely to be used in all applications of a similar type. For example, a user interface framework will provide support

[†]Schmidt, D. C., A. Gokhale, and B. Natarajan. 2004. "Leveraging Application Frameworks." ACM Queue 2 (5 (July/August)): 66–75. doi:10.1145/1016998.1017005.

Figure 15.5 The Model-View-Controller pattern

for interface event handling and will include a set of widgets that can be used to construct displays. It is then left to the developer to specialize these by adding specific functionality for a particular application. For example, in a user interface framework, the developer defines display layouts that are appropriate to the application being implemented.

Frameworks support design reuse in that they provide a skeleton architecture for the application as well as the reuse of specific classes in the system. The architecture is implemented by the object classes and their interactions. Classes are reused directly and may be extended using features such as inheritance and polymorphism.

Frameworks are implemented as a collection of concrete and abstract object classes in an object-oriented programming language. Therefore, frameworks are language-specific. Frameworks are available in commonly used object-oriented programming languages such as Java, C#, and C++, as well as in dynamic languages such as Ruby and Python. In fact, a framework can incorporate other frameworks, where each framework is designed to support the development of part of the application. You can use a framework to create a complete application or to implement part of an application, such as the graphical user interface.

The most widely used application frameworks are web application frameworks (WAFs), which support the construction of dynamic websites. The architecture of a WAF is usually based on the Model-View-Controller (MVC) Composite pattern shown in Figure 15.5. The MVC pattern was originally proposed in the 1980s as an approach to GUI design that allowed for multiple presentations of an object and separate styles of interaction with each of these presentations. In essence, it separates the state from its presentation so that the state may be updated from each presentation.

An MVC framework supports the presentation of data in different ways and allows interaction with each of these presentations. When the data is modified through one of the presentations, the system model is changed and the controllers associated with each view update their presentation.

Frameworks are often implementations of design patterns, as discussed in Chapter 7. For example, an MVC framework includes the Observer pattern, the Strategy pattern, the Composite pattern, and a number of others that are discussed by Gamma et al. (Gamma et al. 1995). The general nature of patterns and their use of abstract and concrete classes allow for extensibility. Without patterns, frameworks would almost certainly be impractical.

Figure 15.6 Inversion of control in frameworks

While each framework includes slightly different functionality, web application frameworks usually provide components and classes that support:

1. *Security* WAFs may include classes to help implement user authentication (login) and access control to ensure that users can only access permitted functionality in the system.
2. *Dynamic web pages* Classes are provided to help you define web page templates and to populate these dynamically with specific data from the system database.
3. *Database integration* Frameworks don't usually include a database but assume that a separate database, such as MySQL, will be used. The framework may include classes that provide an abstract interface to different databases.
4. *Session management* Classes to create and manage sessions (a number of interactions with the system by a user) are usually part of a WAF.
5. *User interaction* Web frameworks provide AJAX (Holdener 2008) and/or HTML5 support (Sarris 2013), which allows interactive web pages to be created. They may include classes that allow device-independent interfaces to be created, which adapt automatically to mobile phones and tablets.

To implement a system using a framework, you add concrete classes that inherit operations from abstract classes in the framework. In addition, you define "callbacks"—methods that are called in response to events recognized by the framework. The framework objects, rather than the application-specific objects, are responsible for control in the system. Schmidt et al. (Schmidt, Gokhale, and Natarajan 2004) call this "inversion of control."

In response to events from the user interface and database framework objects invoke "hook methods" that are then linked to user-provided functionality. The user-provided functionality defines how the application should respond to the event (Figure 15.6). For example, a framework will have a method that handles a mouse click from the environment. This method is called the hook method, which you must configure to call the appropriate application methods to handle the mouse click.

Fayad and Schmidt (Fayad and Schmidt 1997) discuss three other classes of framework:

1. *System infrastructure frameworks* support the development of system infrastructures such as communications, user interfaces, and compilers.
2. *Middleware integration frameworks* consist of a set of standards and associated object classes that support component communication and information exchange. Examples of this type of framework include Microsoft's .NET and Enterprise Java Beans (EJB). These frameworks provide support for standardized component models, as discussed in Chapter 16.
3. *Enterprise application frameworks* are concerned with specific application domains such as telecommunications or financial systems (Baumer et al. 1997). These embed application domain knowledge and support the development of end-user applications. These are not now widely used and have been largely superseded by software product lines.†

Applications that are constructed using frameworks can be the basis for further reuse through the concept of software product lines or application families. Because these applications are constructed using a framework, modifying family members to create instances of the system is often a straightforward process. It involves rewriting concrete classes and methods that you have added to the framework.

Frameworks are a very effective approach to reuse. However, they are expensive to introduce into software development processes as they are inherently complex and it can take several months to learn to use them. It can be difficult and expensive to evaluate available frameworks to choose the most appropriate one. Debugging framework-based applications is more difficult than debugging original code because you may not understand how the framework methods interact. Debugging tools may provide information about the reused framework components, which the developer does not understand.

15.3 Software product lines

When a company has to support a number of similar but not identical systems, one of the most effective approaches to reuse is to create a software product line. Hardware control systems are often developed using this approach to reuse as are domain-specific applications in areas such as logistics or medical systems. For example, a printer manufacturer has to develop printer control software, where there is a specific version of the product for each type of printer. These software versions have much in common, so it makes sense to create a core product (the product line) and adapt this for each printer type.

A software product line is a set of applications with a common architecture and shared components, with each application specialized to reflect specific customer requirements. The core system is designed so that it can be configured and adapted to

†Fayad, M. E., and D. C. Schmidt. 1997. "Object-Oriented Application Frameworks." Comm. ACM 40 (10): 32–38. doi:10.1145/262793.262798.

Figure 15.7 The organization of a base system for a product line

suit the needs of different customers or equipment. This may involve the configuration of some components, implementing additional components, and modifying some of the components to reflect new requirements.

Developing applications by adapting a generic version of the application means that a high proportion of the application code is reused in each system. Testing is simplified because tests for large parts of the application may also be reused, thus reducing the overall application development time. Engineers learn about the application domain through the software product line and so become specialists who can work quickly to develop new applications.

Software product lines usually emerge from existing applications. That is, an organization develops an application and then, when a similar system is required, informally reuses code from this in the new application. The same process is used as other similar applications are developed. However, change tends to corrupt application structure so, as more new instances are developed, it becomes increasingly difficult to create a new version. Consequently, a decision to design a generic product line may then be made. This involves identifying common functionality in product instances and developing a base application, which is then used for future development.

This base application (Figure 15.7) is designed to simplify reuse and reconfiguration. Generally, a base application includes:

1. Core components that provide infrastructure support. These are not usually modified when developing a new instance of the product line.
2. Configurable components that may be modified and configured to specialize them to a new application. Sometimes it is possible to reconfigure these components without changing their code by using a built-in component configuration language.
3. Specialized, domain-specific components some or all of which may be replaced when a new instance of a product line is created.

Application frameworks and software product lines have much in common. They both support a common architecture and components, and require new development to create a specific version of a system. The main differences between these approaches are as follows:

1. Application frameworks rely on object-oriented features such as inheritance and polymorphism to implement extensions to the framework. Generally, the framework

code is not modified, and the possible modifications are limited to whatever is supported by the framework. Software product lines are not necessarily created using an object-oriented approach. Application components are changed, deleted, or rewritten. There are no limits, in principle at least, to the changes that can be made.

2. Most application frameworks provide general support rather than domain-specific support. For example, there are application frameworks to create web-based applications. A software product line usually embeds detailed domain and platform information. For example, there could be a software product line concerned with web-based applications for health record management.

3. Software product lines are often control applications for equipment. For example, there may be a software product line for a family of printers. This means that the product line has to provide support for hardware interfacing. Application frameworks are usually software-oriented, and they do not usually include hardware interaction components.

4. Software product lines are made up of a family of related applications, owned by the same organization. When you create a new application, your starting point is often the closest member of the application family, not the generic core application.

If you are developing a software product line using an object-oriented programming language, then you may use an application framework as a basis for the system. You create the core of the product line by extending the framework with domain-specific components using its built-in mechanisms. There is then a second phase of development where versions of the system for different customers are created. For example, you can use a web-based framework to build the core of a software product line that supports web-based help desks. This "help desk product line" may then be further specialized to provide particular types of help desk support.

The architecture of a software product line often reflects a general, application-specific architectural style or pattern. For example, consider a product-line system that is designed to handle vehicle dispatching for emergency services. Operators of this system take calls about incidents, find the appropriate vehicle to respond to the incident, and dispatch the vehicle to the incident site. The developers of such a system may market versions of it for police, fire, and ambulance services.

This vehicle dispatching system is an example of a generic resource allocation and management architecture (Figure 15.8). Resource management systems use a database of available resources and include components to implement the resource allocation policy that has been decided by the company using the system. Users interact with a resource management system to request and release resources and to ask questions about resources and their availability.

You can see how this four-layer structure may be instantiated in Figure 15.9, which shows the modules that might be included in a vehicle dispatching system product line. The components at each level in the product-line system are as follows:

1. At the interaction level, components provide an operator display interface and an interface with the communications systems used.

Figure 15.8 The architecture of a resource management system

Figure 15.9 A product-line architecture of a vehicle dispatcher system

2. At the I/O management level (level 2), components handle operator authentication, generate reports of incidents and vehicles dispatched, support map output and route planning, and provide a mechanism for operators to query the system databases.
3. At the resource management level (level 3), components allow vehicles to be located and dispatched, update the status of vehicles and equipment, and log details of incidents.
4. At the database level, as well as the usual transaction management support, there are separate databases of vehicles, equipment, and maps.

Figure 15.10 Product instance development

To create a new instance of this system, you may have to modify individual components. For example, the police have a large number of vehicles but a relatively small number of vehicle types. By contrast, the fire service has many types of specialized vehicles but relatively few vehicles. Therefore, when you are implementing a system for these different services, you may have to define a different vehicle database structure.

Various types of specialization of a software product line may be developed:

1. *Platform specialization* Versions of the application may be developed for different platforms. For example, versions of the application may exist for Windows, Mac OS, and Linux platforms. In this case, the functionality of the application is normally unchanged; only those components that interface with the hardware and operating system are modified.

2. *Environment specialization* Versions of the application may be created to handle different operating environments and peripheral devices. For example, a system for the emergency services may exist in different versions, depending on the communications hardware used by each service. For example, police radios may have built-in encryption that has to be used. The product-line components are changed to reflect the functionality and characteristics of the equipment used.

3. *Functional specialization* Versions of the application may be created for specific customers who have different requirements. For example, a library automation system may be modified depending on whether it is used in a public library, a reference library, or a university library. In this case, components that implement functionality may be modified and new components added to the system.

4. *Process specialization* The system may be adapted to cope with specific business processes. For example, an ordering system may be adapted to cope with a centralized ordering process in one company and with a distributed process in another.

Figure 15.10 shows the process for extending a software product line to create a new application. The activities in this process are:

1. *Elicit stakeholder requirements* You may start with a normal requirements engineering process. However, because a system already exists, you can demonstrate the system and have stakeholders experiment with it, expressing their requirements as modifications to the functions provided.

2. *Select the existing system that is the closest fit to the requirements* When creating a new member of a product line, you may start with the nearest product instance. The requirements are analyzed, and the family member that is the closest fit is chosen for modification.

3. *Renegotiate requirements* As more details of required changes emerge and the project is planned, some requirements may be renegotiated with the customer to minimize the changes that will have to be made to the base application.

4. *Adapt existing system* New modules are developed for the existing system, and existing system modules are adapted to meet the new requirements.

5. *Deliver new product family member* The new instance of the product line is delivered to the customer. Some deployment-time configuration may be required to reflect the particular environments where the system will be used. At this stage, you should document its key features so that it may be used as a basis for other system developments in the future.

When you create a new member of a product line, you may have to find a compromise between reusing as much of the generic application as possible and satisfying detailed stakeholder requirements. The more detailed the system requirements, the less likely it is that the existing components will meet these requirements. However, if stakeholders are willing to be flexible and to limit the system modifications that are required, you can usually deliver the system more quickly and at a lower cost.

Software product lines are designed to be reconfigurable. This reconfiguration may involve adding or removing components from the system, defining parameters and constraints for system components, and including knowledge of business processes. This configuration may occur at different stages in the development process:

1. *Design-time configuration* The organization that is developing the software modifies a common product-line core by developing, selecting, or adapting components to create a new system for a customer.

2. *Deployment-time configuration* A generic system is designed for configuration by a customer or consultants working with the customer. Knowledge of the customer's specific requirements and the system's operating environment is embedded in the configuration data used by the generic system.

When a system is configured at design time, the supplier starts with either a generic system or an existing product instance. By modifying and extending modules in this system, the supplier creates a specific system that delivers the required customer functionality. This usually involves changing and extending the source code of the system so that greater flexibility is possible than with deployment-time configuration.

Figure 15.11 Deployment-time configuration

Design-time configuration is used when it is impossible to use the existing deployment-time configuration facilities in a system to develop a new system version. However, over time, when you have created several family members with comparable functionality, you may decide to refactor the core product line to include functionality that has been implemented in several application family members. You then make that new functionality configurable when the system is deployed.

Deployment-time configuration involves using a configuration tool to create a specific system configuration that is recorded in a configuration database or as a set of configuration files (Figure 15.11). The executing system, which may either run on a server or as a stand-alone system on a PC, consults this database when executing so that its functionality may be specialized to its execution context.

Several levels of deployment-time configuration may be provided in a system:

1. *Component selection*, where you select the modules in a system that provide the required functionality. For example, in a patient information system, you may select an image management component that allows you to link medical images (X-rays, CT scans, etc.) to the patient's medical record.

2. *Workflow and rule definition*, where you define workflows (how information is processed, stage by stage), and validation rules that should apply to information entered by users or generated by the system.

3. *Parameter definition*, where you specify the values of specific system parameters that reflect the instance of the application that you are creating. For example, you may specify the maximum length of fields for data input by a user or the characteristics of hardware attached to the system.

Deployment-time configuration can be very complex, and for large systems, it may take several months to configure and test a system for a customer. Large configurable systems may support the configuration process by providing software tools, such as planning tools, to support the configuration process. I discuss deployment-time configuration further in Section 15.4.1. This discussion covers the reuse of application systems that have to be configured to work in different operational environments.

15.4 Application system reuse

An application system product is a software system that can be adapted to the needs of different customers without changing the source code of the system. Application systems are developed by a system vendor for a general market; they are not specially developed for an individual customer. These system products are sometimes known as COTS (Commercial Off-the Shelf System) products. However, the term "COTS" is mostly used in military systems, and I prefer to call these system products *application systems*.

Virtually all desktop software for business and many server-based systems are application systems. This software is designed for general use, so it includes many features and functions. It therefore has the potential to be reused in different environments and as part of different applications. Torchiano and Morisio (Torchiano and Morisio 2004) also discovered that open-source products were often used without change and without looking at the source code.

Application system products are adapted by using built-in configuration mechanisms that allow the functionality of the system to be tailored to specific customer needs. For example, in a hospital patient record system, separate input forms and output reports might be defined for different types of patients. Other configuration features may allow the system to accept plug-ins that extend functionality or check user inputs to ensure that they are valid.

This approach to software reuse has been very widely adopted by large companies since the late 1990s, as it offers significant benefits over customized software development:

1. As with other types of reuse, more rapid deployment of a reliable system may be possible.
2. It is possible to see what functionality is provided by the applications, and so it is easier to judge whether or not they are likely to be suitable. Other companies may already use the applications, so experience of the systems is available.
3. Some development risks are avoided by using existing software. However, this approach has its own risks, as I discuss below.
4. Businesses can focus on their core activity without having to devote a lot of resources to IT systems development.
5. As operating platforms evolve, technology updates may be simplified as these are the responsibility of the application system vendor rather than the customer.

Of course, this approach to software engineering has its own problems:

1. Requirements usually have to be adapted to reflect the functionality and mode of operation of the off-the-shelf application system. This can lead to disruptive changes to existing business processes.

Configurable application systems	Application system integration
Single product that provides the functionality required by a customer	Several different application systems are integrated to provide customized functionality
Based on a generic solution and standardized processes	Flexible solutions may be developed for customer processes
Development focus is on system configuration	Development focus is on system integration
System vendor is responsible for maintenance	System owner is responsible for maintenance
System vendor provides the platform for the system	System owner provides the platform for the system

Figure 15.12 Individual and integrated application systems

2. The application system may be based on assumptions that are practically impossible to change. The customer must therefore adapt its business to reflect these assumptions.
3. Choosing the right application system for an enterprise can be a difficult process, especially as many of these systems are not well documented. Making the wrong choice means that it may be impossible to make the new system work as required.
4. There may be a lack of local expertise to support systems development. Consequently, the customer has to rely on the vendor and external consultants for development advice. This advice may be geared to selling products and services, with insufficient time taken to understand the real needs of the customer.
5. The system vendor controls system support and evolution. It may go out of business, be taken over, or make changes that cause difficulties for customers.

Application systems may be used as individual systems or in combination, where two or more systems are integrated. Individual systems consist of a generic application from a single vendor that is configured to customer requirements. Integrated systems involve integrating the functionality of individual systems, often from different vendors, to create a new application system. Figure 15.12 summarizes the differences between these different approaches. I discuss application system integration in Section 15.4.2.

15.4.1 Configurable application systems

Configurable application systems are generic application systems that may be designed to support a particular business type, business activity, or, sometimes, a complete business enterprise. For example, a system produced for dentists may handle appointments, reminders, dental records, patient recall, and billing. At a larger scale, an Enterprise Resource Planning (ERP) system may support the manufacturing, ordering, and customer relationship management processes in a large company.

Domain-specific application systems, such as systems to support a business function (e.g., document management), provide functionality that is likely to be required by a range of potential users. However, they also incorporate built-in assumptions about how

Figure 15.13 The architecture of an ERP system

users work, and these assumptions may cause problems in specific situations. For example, a system to support student registration in a university may assume that students will be registered for one degree at one university. However, if universities collaborate to offer joint degrees, then it may be practically impossible to represent this detail in the system.

Enterprise Resource Planning (ERP) systems, such as those produced by SAP and Oracle, are large-scale, integrated systems designed to support business practices such as ordering and invoicing, inventory management, and manufacturing scheduling (Monk and Wagner 2013). The configuration process for these systems involves gathering detailed information about the customer's business and business processes, and embedding this information in a configuration database. This often requires detailed knowledge of configuration notations and tools and is usually carried out by consultants working alongside system customers.

A generic ERP system includes a number of modules that may be composed in different ways to create a system for a customer. The configuration process involves choosing which modules are to be included, configuring these individual modules, defining business processes and business rules, and defining the structure and organization of the system database. A model of the overall architecture of an ERP system that supports a range of business functions is shown in Figure 15.13.

The key features of this architecture are as follows:

1. A number of modules to support different business functions. These are large grain modules that may support entire departments or divisions of the business. In the example shown in Figure 15.13, the modules that have been selected for inclusion in the system are a module to support purchasing; a module to support supply chain management; a logistics module to support the delivery of goods; and a customer relationship management (CRM) module to maintain customer information.

2. A defined set of business process models, associated with each module, which relate to activities in that module. For example, the ordering process model may define how orders are created and approved. This will specify the roles and activities involved in placing an order.

3. A common database that maintains information about all related business functions. Thus, it should not be necessary to replicate information, such as customer details, in different parts of the business.

4. A set of business rules that apply to all data in the database. Therefore, when data is input from one function, these rules should ensure that it is consistent with the data required by other functions. For example, a business rule may require that all expense claims have to be approved by someone more senior than the person making the claim.

ERP systems are used in almost all large companies to support some or all of their functions. They are, therefore, a very widely used form of software reuse. The obvious limitation of this approach to reuse is that the functionality of the customer's application is restricted to the functionality of the ERP system's built-in modules. If a company needs additional functionality, it may have to develop a separate add-on system to provide this functionality.

Furthermore, the buyer company's processes and operations have to be defined in the ERP system's configuration language. This language embeds the understanding of business processes as seen by the system vendor, and there may be a mismatch between these assumptions and the concepts and processes used in the customer's business. A serious mismatch between the customer's business model and the system model used by the ERP system makes it highly probable that the ERP system will not meet the customer's real needs (Scott 1999).

For example, in an ERP system that was sold to a university, a fundamental system concept was the notion of a customer. In this system, a customer was an external agent that bought goods and services from a supplier. This concept caused great difficulties when configuring the system. Universities do not really have customers. Rather, they have customer-type relationships with a range of people and organizations such as students, research funding agencies, and educational charities. None of these relationships is compatible with a customer relationship where a person or business buys products or services from another. In this particular case, it took several months to resolve this mismatch, and the final solution only partially met the university's requirements.

ERP systems usually require extensive configuration to adapt them to the requirements of each organization where they are installed. This configuration may involve:

1. Selecting the required functionality from the system, for example, by deciding what modules should be included.
2. Establishing a data model that defines how the organization's data will be structured in the system database.
3. Defining business rules that apply to that data.
4. Defining the expected interactions with external systems.
5. Designing the input forms and the output reports generated by the system.
6. Designing new business processes that conform to the underlying process model supported by the system.
7. Setting parameters that define how the system is deployed on its underlying platform.

Once the configuration settings are completed, the new system is then ready for testing. Testing is a major problem when systems are configured rather than programmed using a conventional language. There are two reasons for this:

1. Test automation may be difficult or impossible. There may be no easy access to an API that can be used by testing frameworks such as JUnit, so the system has to be tested manually by testers inputting test data to the system. Furthermore, systems are often specified informally, so defining test cases may be difficult without a lot of help from end-users.
2. Systems errors are often subtle and specific to business processes. The application system or ERP system is a reliable platform, so technical system failures are rare. The problems that occur are often due to misunderstandings between those configuring the system and user stakeholders. System testers without detailed knowledge of the end-user processes cannot detect these errors.

15.4.2 Integrated application systems

Integrated application systems include two or more application systems or, sometimes, legacy systems. You may use this approach when no single application system meets all of your needs or when you wish to integrate a new application system with systems that you are already using. The component systems may interact through their APIs or service interfaces if these are defined. Alternatively, they may be composed by connecting the output of one system to the input of another or by updating the databases used by the applications.

To develop integrated application systems, you have to make a number of design choices:

1. *Which individual application systems offer the most appropriate functionality?* Typically, several system products will be available, which can be combined in different ways. If you don't already have experience with a particular application system, it can be difficult to decide which product is the most suitable.
2. *How will data be exchanged?* Different systems normally use unique data structures and formats. You have to write adaptors that convert from one representation to another. These adaptors are runtime systems that operate alongside the constituent application systems.
3. *What features of a product will actually be used?* Individual application systems may include more functionality than you need, and functionality may be duplicated across different products. You have to decide which features in what product are most appropriate for your requirements. If possible, you should also deny access to unused functionality because this can interfere with normal system operation.

Figure 15.14 An integrated procurement system

Consider the following scenario as an illustration of application system integration. A large organization intends to develop a procurement system that allows staff to place orders from their desk. By introducing this system across the organization, the company estimates that it can save $5 million per year. By centralizing buying, the new procurement system can ensure that orders are always made from suppliers who offer the best prices and should reduce the administration associated with orders. As with manual systems, the system involves choosing the goods available from a supplier, creating an order, having the order approved, sending the order to a supplier, receiving the goods, and confirming that payment should be made.

The company has a legacy ordering system that is used by a central procurement office. This order processing software is integrated with an existing invoicing and delivery system. To create the new ordering system, the legacy system is integrated with a web-based e-commerce platform and an email system that handles communications with users. The structure of the final procurement system is shown in Figure 15.14.

This procurement system should be a client–server system with standard web browsing and email systems used on the client. On the server, the e-commerce platform has to integrate with the existing ordering system through an adaptor. The e-commerce system has its own format for orders, confirmations of delivery, and so forth, and these have to be converted into the format used by the ordering system. The e-commerce system uses the email system to send notifications to users, but the ordering system was never designed for this purpose. Therefore, another adaptor has to be written to convert the notifications from the ordering system into email messages.

Months, sometimes years, of implementation effort can be saved, and the time to develop and deploy a system can be drastically reduced by integrating existing application systems. The procurement system described above was implemented and deployed in a very large company in nine months. It had originally been estimated that it would take three years to develop a procurement system in Java that could be integrated with the legacy ordering system.

Figure 15.15
Application wrapping

Application system integration can be simplified if a service-oriented approach is used. Essentially, a service-oriented approach means allowing access to the application system's functionality through a standard service interface, with a service for each discrete unit of functionality. Some applications may offer a service interface, but sometimes this service interface has to be implemented by the system integrator. Essentially, you have to program a wrapper that hides the application and provides externally visible services (Figure 15.15). This approach is particularly valuable for legacy systems that have to be integrated with newer application systems.

In principle, integrating application systems is the same as integrating any other component. You have to understand the system interfaces and use them exclusively to communicate with the software; you have to trade off specific requirements against rapid development and reuse; and you have to design a system architecture that allows the application systems to operate together.

However, the fact that these products are usually large systems in their own right, and are often sold as separate standalone systems, introduces additional problems. Boehm and Abts (Boehm and Abts 1999) highlight four important system integration problems:

1. *Lack of control over functionality and performance* Although the published interface of a product may appear to offer the required facilities, the system may not be properly implemented or may perform poorly. The product may have hidden operations that interfere with its use in a specific situation. Fixing these problems may be a priority for the system integrator but may not be of real concern for the product vendor. Users may simply have to find workarounds to problems if they wish to reuse the application system.

2. *Problems with system interoperability* It is sometimes difficult to get individual application systems to work together because each system embeds its own assumptions about how it will be used. Garlan et al. (Garlan, Allen, and Ockerbloom 1995), reporting on their experience integrating four application systems, found that three of these products were event-based but that each used a different model of events. Each system assumed that it had exclusive access to the event queue. As a consequence, integration was very difficult. The project

required five times as much effort as originally predicted. The schedule was extended to two years rather than the predicted six months.

In a retrospective analysis of their work 10 years later, Garlan et al. (Garlan, Allen, and Ockerbloom 2009) concluded that the integration problems that they discovered had not been solved. Torchiano and Morisio (Torchiano and Morisio 2004) found that lack of compliance with standards in many application systems meant that integration was more difficult than anticipated.

3. *No control over system evolution* Vendors of application systems make their own decisions on system changes, in response to market pressures. For PC products in particular, new versions are often produced frequently and may not be compatible with all previous versions. New versions may have additional unwanted functionality, and previous versions may become unavailable and unsupported.

4. *Support from system vendors* The level of support available from system vendors varies widely. Vendor support is particularly important when problems arise as developers do not have access to the source code and detailed documentation of the system. While vendors may commit to providing support, changing market and economic circumstances may make it difficult for them to deliver this commitment. For example, a system vendor may decide to discontinue a product because of limited demand, or they may be taken over by another company that does not wish to support the products that have been acquired.

Boehm and Abts reckon that, in many cases, the cost of system maintenance and evolution may be greater for integrated application systems. The above difficulties are life-cycle problems; they don't just affect the initial development of the system. The further removed the people involved in the system maintenance become from the original system developers, the more likely it is that difficulties will arise with the integrated system.

KEY POINTS

- There are many different ways to reuse software. These range from the reuse of classes and methods in libraries to the reuse of complete application systems.
- The advantages of software reuse are lower costs, faster software development, and lower risks. System dependability is increased. Specialists can be used more effectively by concentrating their expertise on the design of reusable components.
- Application frameworks are collections of concrete and abstract objects that are designed for reuse through specialization and the addition of new objects. They usually incorporate good design practice through design patterns.

- Software product lines are related applications that are developed from one or more base applications. A generic system is adapted and specialized to meet specific requirements for functionality, target platform, or operational configuration.
- Application system reuse is concerned with the reuse of large-scale, off-the-shelf systems. These provide a lot of functionality, and their reuse can radically reduce costs and development time. Systems may be developed by configuring a single, generic application system or by integrating two or more application systems.
- Potential problems with application system reuse include lack of control over functionality, performance, and system evolution; the need for support from external vendors; and difficulties in ensuring that systems can interoperate.

FURTHER READING

"Overlooked Aspects of COTS-Based Development." An interesting article that discusses a survey of developers using a COTS-based approach, and the problems that they encountered. (M. Torchiano and M. Morisio, *IEEE Software*, 21 (2), March–April 2004) http://dx.doi.org/10.1109/MS.2004.1270770

CRUISE—Component Reuse in Software Engineering. This e-book covers a wide range of reuse topics, including case studies, component-based reuse, and reuse processes. However, its coverage of application system reuse is limited. (L. Nascimento et al., 2007) http://www.academia.edu/179616/C.R.U.I.S.E_-_Component_Reuse_in_Software_Engineering

"Construction by Configuration: A New Challenge for Software Engineering." In this invited paper, I discuss the problems and difficulties of constructing a new application by configuring existing systems. (I. Sommerville, *Proc. 19th Australian Software Engineering Conference*, 2008) http://dx.doi.org/10.1109/ASWEC.2008.75

"Architectural Mismatch: Why Reuse Is Still So Hard." This article looks back on an earlier paper that discussed the problems of reusing and integrating a number of application systems. The authors concluded that, although some progress has been made, there were still problems in conflicting assumptions made by the designers of the individual systems. (D. Garlan et al., *IEEE Software*, 26 (4), July–August 2009) http://dx.doi.org//10.1109/MS.2009.86

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-reuse/

EXERCISES

15.1. What major technical and nontechnical factors hinder software reuse? Do you personally reuse much software and, if not, why not?

15.2. List the benefits of software reuse and explain why the expected lifetime of the software should be considered when planning reuse.

15.3. How does the base application's design in the product line simplify reuse and reconfiguration?

15.4. Explain what is meant by "inversion of control" in application frameworks. Explain why this approach could cause problems if you integrated two separate systems that were originally created using the same application framework.

15.5. Using the example of the weather station system described in Chapters 1 and 7, suggest a product-line architecture for a family of applications that are concerned with remote monitoring and data collection. You should present your architecture as a layered model, showing the components that might be included at each level.

15.6. Most desktop software, such as word processing software, can be configured in a number of different ways. Examine software that you regularly use and list the configuration options for that software. Suggest difficulties that users might have in configuring the software. Microsoft Office (or one of its open-source alternatives) is a good example to use for this exercise.

15.7. Why have many large companies chosen ERP systems as the basis for their organizational information system? What problems may arise when deploying a large-scale ERP system in an organization?

15.8. What are the significant benefits offered by the application system reuse approach when compared with the custom software development approach?

15.9. Explain why adaptors are usually needed when systems are constructed by integrating application systems. Suggest three practical problems that might arise in writing adaptor software to link two application systems.

15.10. The reuse of software raises a number of copyright and intellectual property issues. If a customer pays a software contractor to develop a system, who has the right to reuse the developed code? Does the software contractor have the right to use that code as a basis for a generic component? What payment mechanisms might be used to reimburse providers of reusable components? Discuss these issues and other ethical issues associated with the reuse of software.

REFERENCES

Baumer, D., G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle, and H. Zullighoven. 1997. "Framework Development for Large Systems." *Comm. ACM* 40 (10): 52–59. doi:10.1145/262793.262804.

Boehm, B., and C. Abts. 1999. "COTS Integration: Plug and Pray?" *Computer* 32 (1): 135–138. doi:10.1109/2.738311.

Fayad, M.E., and D.C. Schmidt. 1997. "Object-Oriented Application Frameworks." *Comm. ACM* 40 (10): 32–38. doi:10.1145/262793.262798.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Garlan, D., R. Allen, and J. Ockerbloom. 1995. "Architectural Mismatch: Why Reuse Is So Hard." *IEEE Software* 12 (6): 17–26. doi:10.1109/52.469757.

———. 2009. "Architectural Mismatch: Why Reuse Is Still so Hard." *IEEE Software* 26 (4): 66–69. doi:10.1109/MS.2009.86.

Holdener, A.T. 2008. *Ajax: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates.

Jacobsen, I., M. Griss, and P. Jonsson. 1997. *Software Reuse*. Reading, MA: Addison-Wesley.

Monk, E., and B. Wagner. 2013. *Concepts in Enterprise Resource Planning, 4th ed.* Independence, KY: CENGAGE Learning.

Sarris, S. 2013. *HTML5 Unleashed*. Indianapolis, IN: Sams Publishing.

Schmidt, D. C., A. Gokhale, and B. Natarajan. 2004. "Leveraging Application Frameworks." *ACM Queue* 2 (5 (July/August)): 66–75. doi:10.1145/1016998.1017005.

Scott, J. E. 1999. "The FoxMeyer Drug's Bankruptcy: Was It a Failure of ERP." In *Proc. Association for Information Systems 5th Americas Conf. on Information Systems*. Milwaukee, WI. http://www.uta.edu/faculty/weltman/OPMA5364TW/FoxMeyer.pdf

Torchiano, M., and M. Morisio. 2004. "Overlooked Aspects of COTS-Based Development." *IEEE Software* 21 (2): 88–93. doi:10.1109/MS.2004.1270770.

16

Component-based software engineering

Objectives

The objective of this chapter is to describe an approach to software reuse based on the composition of standardized, reusable components. When you have read this chapter, you will:

- understand what is meant by a software component that may be included in a program as an executable element;
- understand the key elements of software component models and the support provided by middleware for these models;
- be aware of the key activities in the component-based software engineering (CBSE) process for reuse and the CBSE process with reuse;
- understand three different types of component composition and some of the problems that have to be resolved when components are composed to create new components or systems.

Contents

16.1 Components and component models
16.2 CBSE processes
16.3 Component composition

Component-based software engineering (CBSE) emerged in the late 1990s as an approach to software systems development based on reusing software components. Its creation was motivated by frustration that object-oriented development had not led to extensive reuse, as had been originally suggested. Single-object classes were too detailed and specific and often had to be bound with an application at compile-time. You had to have detailed knowledge of the classes to use them, which usually meant that you had to have the component source code. Selling or distributing objects as individual reusable components was therefore practically impossible.

Components are higher-level abstractions than objects and are defined by their interfaces. They are usually larger than individual objects, and all implementation details are hidden from other components. Component-based software engineering is the process of defining, implementing, and integrating or composing these loosely coupled, independent components into systems.

CBSE has become as an important software development approach for large-scale enterprise systems, with demanding performance and security requirements. Customers are demanding secure and dependable software that is delivered and deployed more quickly. The only way that these demands can be met is to build software by reusing existing components.

The essentials of component-based software engineering are:

1. Independent components that are completely specified by their interfaces. There should be a clear separation between the component interface and its implementation. This means that one implementation of a component can be replaced by another, without the need to change other parts of the system.
2. Component standards that define interfaces and so facilitate the integration of components. These standards are embodied in a component model. They define, at the very minimum, how component interfaces should be specified and how components communicate. Some models go much further and define interfaces that should be implemented by all conformant components. If components conform to standards, then their operation is independent of their programming language. Components written in different languages can be integrated into the same system.
3. Middleware that provides software support for component integration. To make independent, distributed components work together, you need middleware support that handles component communications. Middleware for component support handles low-level issues efficiently and allows you to focus on application-related problems. In addition, middleware for component support may provide support for resource allocation, transaction management, security, and concurrency.
4. A development process that is geared to component-based software engineering. You need a development process that allows requirements to evolve, depending on the functionality of available components.

Component-based development embodies good software engineering practice. It often makes sense to design a system using components, even if you have to develop

Problems with CBSE

CBSE is now a mainstream approach to software engineering and is widely used when creating new systems. However, when used as an approach to reuse, problems include component trustworthiness, component certification, requirements compromises, and prediction of the properties of components, especially when they are integrated with other components.

http://software-engineering-book.com/web/cbse-problems/

rather than reuse these components. Underlying CBSE are sound design principles that support the construction of understandable and maintainable software:

1. Components are independent, so they do not interfere with each other's operation. Implementation details are hidden. The component's implementation can be changed without affecting the rest of the system.
2. Components communicate through well-defined interfaces. If these interfaces are maintained, one component can be replaced by another component providing additional or enhanced functionality.
3. Component infrastructures offer a range of standard services that can be used in application systems. This reduces the amount of new code that has to be developed.

The initial motivation for CBSE was the need to support both reuse and distributed software engineering. A component was seen as an element of a software system that could be accessed, using a remote procedure call mechanism, by other components running on separate computers. Each system that reused a component had to incorporate its own copy of that component. This idea of a component extended the notion of distributed objects, as defined in distributed systems models such as the CORBA specification (Pope 1997). Several different protocols and technology-specific "standards" were introduced to support this view of a component, including Sun's Enterprise Java Beans (EJB), Microsoft's COM and .NET, and CORBA's CCM (Lau and Wang 2007).

Unfortunately, the companies involved in proposing standards could not agree on a single standard for components, thereby limiting the impact of this approach to software reuse. It is impossible for components developed using different approaches to work together. Components that are developed for different platforms, such as .NET or J2EE, cannot interoperate. Furthermore, the standards and protocols proposed were complex and difficult to understand. This was also a barrier to their adoption.

In response to these problems, the notion of a component as a service was developed, and standards were proposed to support service-oriented software engineering. The most significant difference between a component as a service and the original notion of a component is that services are stand-alone entities that are external to a program using them. When you build a service-oriented system, you reference the external service rather than including a copy of that service in your system.

Service-oriented software engineering is a type of component-based software engineering. It uses a simpler notion of a component than that originally proposed in CBSE,

where components were executable routines that were included in larger systems. Each system that used a component embedded its own version of that component. Service-oriented approaches are gradually replacing CBSE with embedded components as an approach to systems development. In this chapter, I discuss the use of CBSE with embedded components; service-oriented software engineering is covered in Chapter 18.

16.1 Components and component models

The software reuse community generally agrees that a component is an independent software unit that can be composed with other components to create a software system. Beyond that, however, people have proposed varying definitions of a software component. Councill and Heineman (Councill and Heineman 2001) define a component as:

> *A software element that conforms to a standard component model and can be independently deployed and composed without modification according to a composition standard.*†

This definition is standards-based so that a software unit that conforms to these standards is a component. Szyperski (Szyperski 2002), however, does not mention standards in his definition of a component but focuses instead on the key characteristics of components:

> *A software component is a unit of composition with contractually-specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*‡

Both of these definitions were developed around the idea of a component as an element that is embedded in a system, rather than a service that is referenced by the system. However, they are equally applicable to service components.

Szyperski also states that a component has no externally observable state; that is, copies of components are indistinguishable. However, some component models, such as the Enterprise Java Beans model, allow stateful components, so these do not correspond with Szyperski's definition. While stateless components are certainly simpler to use, in some systems stateful components are more convenient and reduce system complexity.

What the above definitions have in common is that they agree that components are independent and that they are the fundamental unit of composition in a system. I think that, if we combine these proposals, we get a more rounded description of a reusable component. Figure 16.1 shows what I consider to be the essential characteristics of a component as used in CBSE.

---

†Councill, W. T., and G. T. Heineman. 2001. "Definition of a Software Component and Its Elements." In Component-Based Software Engineering, edited by G T Heineman and W T Councill, 5–20. Boston: Addison Wesley.

‡Szyperski, C. 2002. Component Software: Beyond Object-Oriented Programming, 2nd Ed. Harlow, UK: Addison Wesley.

Component characteristic	Description
Composable	For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes.
Deployable	To be deployable, a component has to be self-contained. It must be able to operate as a stand-alone entity on a component platform that provides an implementation of the component model. This usually means that the component is binary and does not have to be compiled before it is deployed. If a component is implemented as a service, it does not have to be deployed by a user of that component. Rather, it is deployed by the service provider.
Documented	Components have to be fully documented so that potential users can decide whether or not the components meet their needs. The syntax and, ideally, the semantics of all component interfaces should be specified.
Independent	A component should be independent—it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a "requires" interface specification.
Standardized	Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment.

Figure 16.1 Component characteristics

A useful way of thinking about a component is as a provider of one or more services, even if the component is embedded rather than implemented as a service. When a system needs something to be done, it calls on a component to provide that service without caring about where that component is executing or the programming language used to develop the component. For example, a component in a system used in a public library might provide a search service that allows users to search the library catalog. A component that converts from one graphical format to another (e.g., TIFF to JPEG) provides a data conversion service and so on.

Viewing a component as a service provider emphasizes two critical characteristics of a reusable component:

1. The component is an independent executable entity that is defined by its interfaces. You don't need any knowledge of its source code to use it. It can either be referenced as an external service or included directly in a program.
2. The services offered by a component are made available through an interface, and all interactions are through that interface. The component interface is expressed in terms of parameterized operations, and its internal state is never exposed.

In principle, all components have two related interfaces, as shown in Figure 16.2. These interfaces reflect the services that the component provides and the services that the component requires to operate correctly:

1. The "provides" interface defines the services provided by the component. This interface is the component API. It defines the methods that can be called by a user

Figure 16.2 Component interfaces

of the component. In a UML component diagram, the "provides" interface for a component is indicated by a circle at the end of a line from the component icon.

2. The "requires" interface specifies the services that other components in the system must provide if a component is to operate correctly. If these services are not available, then the component will not work. This does not compromise the independence or deployability of a component because the "requires" interface does not define how these services should be provided. In the UML, the symbol for a "requires" interface is a semicircle at the end of a line from the component icon. Notice that "provides" and "requires" interface icons can fit together like a ball and socket.

To illustrate these interfaces, Figure 16.3 shows a model of a component that has been designed to collect and collate information from an array of sensors. It runs autonomously to collect data over a period of time and, on request, provides collated data to a calling component. The "provides" interface includes methods to add, remove, start, stop, and test sensors. The report method returns the sensor data that has been collected, and the listAll method provides information about the attached sensors. Although I have not shown them here, these methods have associated parameters specifying the sensor identifiers, locations, and so on.

The "requires" interface is used to connect the component to the sensors. It assumes that sensors have a data interface, accessed through sensorData, and a management interface, accessed through sensorManagement. This interface has been designed to connect to different types of sensors so that it does not include specific sensor operations such as Test and provideReading. Instead, the commands used by a specific type of sensor are embedded in a string, which is a parameter to the operations in the "requires" interface. Adaptor components parse this parameter string and translate the embedded commands into the specific control interface of each type of sensor. I discuss the use of adaptors later in this chapter, where I show how the data collector component may be connected to a sensor (Figure 16.12).

Figure 16.3 A model of a data collector component

Components and objects

Components are often implemented in object-oriented languages, and, in some cases, accessing the "provides" interface of a component is done through method calls. However, components and object classes are not the same thing. Unlike object classes, components are independently deployable, do not define types, are language-independent, and are based on a standard component model.

http://software-engineering-book.com/web/components-and-objects/

Components are accessed using remote procedure calls (RPCs). Each component has a unique identifier and, using this name, may be called from another computer. The called component uses the same mechanism to access the "required" components that are defined in its interface.

An important difference between a component as an external service and a component as a program element accessed using a remote procedure call is that services are completely independent entities. They do not have an explicit "requires" interface. Of course, they do require other components to support their operation, but these are provided internally. Other programs can use services without the need to implement any additional support required by the service.

16.1.1 Component models

A component model is a definition of standards for component implementation, documentation, and deployment. These standards are for component developers to ensure that components can interoperate. They are also for providers of component execution infrastructures who provide middleware to support component operation. For service components, the most important component model is the Web Service models, and for embedded components, widely used models include the Enterprise Java Beans (EJB) model and Microsoft's .NET model (Lau and Wang 2007).

The basic elements of an ideal component model are discussed by Weinreich and Sametinger (Weinreich and Sametinger 2001). I summarize these model elements in Figure 16.4. This diagram shows that the elements of a component model define the component interfaces, the information that you need to use the component in a program, and how a component should be deployed:

1. *Interfaces* Components are defined by specifying their interfaces. The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters, and exceptions, which should be included in the interface definition. The model should also specify the language used to define the component interfaces.

   For web services, interface specification uses XML-based languages as discussed in Chapter 18; EJB is Java-specific, so Java is used as the interface definition language; in .NET, interfaces are defined using Microsoft's Common

Figure 16.4 Basic elements of a component model

Intermediate Language (CIL). Some component models require specific interfaces that must be defined by a component. These are used to compose the component with the component model infrastructure, which provides standardized services such as security and transaction management.

2. *Usage* In order for components to be distributed and accessed remotely via RPCs, they need to have a unique name or handle associated with them. This has to be globally unique. For example, in EJB, a hierarchical name is generated with the root based on an Internet domain name. Services have a unique URI (Uniform Resource Identifier).

   Component meta-data is data about the component itself, such as information about its interfaces and attributes. The meta-data is important because it allows users of the component to find out what services are provided and required. Component model implementations normally include specific ways (such as the use of a reflection interface in Java) to access this component meta-data.

   Components are generic entities, and, when deployed, they have to be configured to fit into an application system. For example, you could configure the Data collector component (Figure 16.3) by defining the maximum number of sensors in a sensor array. The component model may therefore specify how the binary components can be customized for a particular deployment environment.

3. *Deployment* The component model includes a specification of how components should be packaged for deployment as independent, executable routines. Because components are independent entities, they have to be packaged with all supporting software that is not provided by the component infrastructure, or is not defined in a "requires" interface. Deployment information includes information about the contents of a package and its binary organization.

   Inevitably, as new requirements emerge, components will have to be changed or replaced. The component model may therefore include rules governing when and how component replacement is allowed. Finally, the component model may define the component documentation that should be produced. This is used to find the component and to decide whether it is appropriate.

Figure 16.5 Middleware services defined in a component model

For components that are executable routines rather than external services, the component model defines the services to be provided by the middleware that supports the executing components. Weinreich and Sametinger use the analogy of an operating system to explain component models. An operating system provides a set of generic services that can be used by applications. A component model implementation provides comparable shared services for components. Figure 16.5 shows some of the services that may be provided by an implementation of a component model.

The services provided by a component model implementation fall into two categories:

1. *Platform services*, which enable components to communicate and interoperate in a distributed environment. These are the fundamental services that must be available in all component-based systems.
2. *Support services*, which are common services that many different components are likely to require. For example, many components require authentication to ensure that the user of component services is authorized. It makes sense to provide a standard set of middleware services for use by all components. This reduces the costs of component development, and potential component incompatibilities can be avoided.

Middleware implements the common component services and provides interfaces to them. To make use of the services provided by a component model infrastructure, you can think of the components as being deployed in a "container." A container is an implementation of the support services plus a definition of the interfaces that a component must provide to integrate it with the container. Conceptually, when you add a component to the container, the component can access the support services and the container can access the component interfaces. When in use, the component interfaces themselves are not accessed directly by other components. They are accessed through a container interface that invokes code to access the interface of the embedded component.

Containers are large and complex and, when you deploy a component in a container, you get access to all middleware services. However, simple components may

not need all of the facilities offered by the supporting middleware. The approach taken in web services to common service provision is therefore rather different. For web services, standards have been defined for common services such as transaction management and security, and these standards have been implemented as program libraries. If you are implementing a service component, you only use the common services that you need.

The services associated with a component model have much in common with the facilities provided by object-oriented frameworks, which I discussed in Chapter 15. Although the services provided may not be as comprehensive, framework services are often more efficient than container-based services. As a consequence, some people think that it is best to use frameworks such as SPRING (Wheeler and White 2013) for Java development rather than the fully-featured component model in EJB.

16.2 CBSE processes

CBSE processes are software processes that support component-based software engineering. They take into account the possibilities of reuse and the different process activities involved in developing and using reusable components. Figure 16.6 (Kotonya 2003) presents an overview of the processes in CBSE. At the highest level, there are two types of CBSE processes:

1. *Development for reuse* This process is concerned with developing components or services that will be reused in other applications. It usually involves generalizing existing components.
2. *Development with reuse* This process is the process of developing new applications using existing components and services.

These processes have different objectives and therefore include different activities. In the development for reuse process, the objective is to produce one or more reusable components. You know the components that you will be working with, and you have access to their source code to generalize them. In development with reuse, you don't know what components are available, so you need to discover these components and design your system to make the most effective use of them. You may not have access to the component source code.

You can see from Figure 16.6 that the basic processes of CBSE with and for reuse have supporting processes that are concerned with component acquisition, component management, and component certification:

1. *Component acquisition* is the process of acquiring components for reuse or development into a reusable component. It may involve accessing locally developed components or services or finding these components from an external source.

Figure 16.6 CBSE processes

2. *Component management* is concerned with managing a company's reusable components, ensuring that they are properly catalogued, stored, and made available for reuse.

3. *Component certification* is the process of checking a component and certifying that it meets its specification.

Components maintained by an organization may be stored in a component repository that includes both the components and information about their use.

16.2.1 CBSE for reuse

CBSE for reuse is the process of developing reusable components and making them available for reuse through a component management system. The vision of early supporters of CBSE (Szyperski 2002) was that a thriving component marketplace would develop. There would be specialist component providers and component vendors who would organize the sale of components from different developers. Software developers would buy components to include in a system or pay for services as they were used. However, this vision has not been realized. There are relatively few component suppliers, and buying off-the-shelf components is uncommon.

Consequently, CBSE for reuse is mostly used within organizations that have made a commitment to reuse-driven software engineering. These companies have a base of internally developed components that can be reused. However, these internally developed components may not be reusable without change. They often include application-specific features and interfaces that are unlikely to be required in other programs where the component is reused.

To make components reusable, you have to adapt and extend the application-specific components to create more generic and therefore more reusable versions. Obviously, this adaptation has an associated cost. You have to decide first, whether a component is likely to be reused and second, whether the cost savings from future reuse justify the costs of making the component reusable.

To answer the first of these questions, you have to decide whether or not the component implements one or more stable domain abstractions. Stable domain abstractions are fundamental elements of the application domain that change slowly. For example, in a banking system, domain abstractions might include accounts, account holders, and statements. In a hospital management system, domain abstractions might include patients, treatments, and nurses. These domain abstractions are sometimes called business objects. If the component is an implementation of a commonly used domain abstraction or group of related business objects, it can probably be reused.

To answer the question about cost-effectiveness, you have to assess the costs of changes that are required to make the component reusable. These costs are the costs of component documentation and component validation, and of making the component more generic. Changes that you may make to a component to make it more reusable include:

- removing application-specific methods;
- changing names to make them more general;
- adding methods to provide more complete functional coverage;
- making exception handling consistent for all methods;
- adding a "configuration" interface to allow the component to be adapted to different situations of use;
- integrating required components to increase independence.

The problem of exception handling is a difficult one. In principle, components should not handle exceptions themselves because each application will have its own requirements for exception management. Rather, the component should define what exceptions can arise and should publish these exceptions as part of the interface. For example, a simple component implementing a stack data structure should detect and publish stack overflow and stack underflow exceptions. In practice, however, there are two problems with this process:

1. Publishing all exceptions leads to bloated interfaces that are harder to understand. This may put off potential users of the component.
2. The operation of the component may depend on local exception handling, and changing this may have serious implications for the functionality of the component.

You therefore have to take a pragmatic approach to component exception handling. Common technical exceptions, where recovery is important for the functioning of the component, should be handled locally. These exceptions and how they are handled

should be documented with the component. Other exceptions that are related to the business function of the component should be passed to the calling component for handling.

Mili et al. (Mili et al. 2002) discuss ways of estimating the costs of making a component reusable and the returns from that investment. The benefits of reusing rather than redeveloping a component are not simply productivity gains. There are also quality gains, because a reused component should be more dependable, and time-to-market gains. These are the increased returns that accrue from deploying the software more quickly.

Mili et al. present various formulas for estimating these gains, as does the COCOMO model, discussed in Chapter 23. However, the parameters of these formulas are difficult to estimate accurately, and the formulas must be adapted to local circumstances, making them difficult to use. I suspect that few software project managers use these models to estimate the return on investment from component reusability.

Whether or not a component is reusable depends on its application domain, functionality, and generality. If the domain is a general one and the component implements standard functionality in that domain, then it is more likely to be reusable. As you add generality to a component, you increase its reusability because it can be applied in a wider range of environments. Unfortunately, this normally means that the component has more operations and is more complex, which makes the component harder to understand and use.

There is, therefore, a trade-off between the reusability and understandability of a component. To make a component reusable you have to provide a set of generic interfaces with operations that cater to all of the ways in which the component could be used. Reusability adds complexity and hence reduces component understandability. This makes it more difficult and time consuming to decide whether a component is suitable for reuse. Because of the time involved in understanding a reusable component, it is sometimes more cost-effective to reimplement a simpler component with the specific functionality that is required.

A potential source of components is legacy systems. As I discussed in Chapter 9, legacy systems are systems that fulfill an important business function but are written using obsolete software technologies. As a result, it may be difficult to use them with new systems. However, if you convert these old systems to components, their functionality can be reused in new applications.

Of course, these legacy systems do not normally have clearly defined "requires" and "provides" interfaces. To make these components reusable, you have to create a wrapper that defines the component interfaces. The wrapper hides the complexity of the underlying code and provides an interface for external components to access services that are provided. Although this wrapper is a fairly complex piece of software, the cost of wrapper development may be significantly less than the cost of reimplementing the legacy system.

Once you have developed and tested a reusable component or service, it then has to be managed for future reuse. Management involves deciding how to classify the component so that it can be discovered, making the component available either in a repository or as a service, maintaining information about the use of the component, and keeping track of different component versions. If the component is open-source, you may make it available in a public repository such as GitHub or Sourceforge. If it is intended for use in a company, then you may use an internal repository system.

Figure 16.7 CBSE with reuse

A company with a reuse program may carry out some form of component certification before the component is made available for reuse. Certification means that someone apart from the developer checks the quality of the component. They test the component and certify that it has reached an acceptable quality standard, before it is made available for reuse. However, this process can be expensive, and so many companies simply leave testing and quality checking to the component developers.

16.2.2 CBSE with reuse

The successful reuse of components requires a development process tailored to including reusable components in the software being developed. The CBSE with reuse process has to include activities that find and integrate reusable components. The structure of such a process was discussed in Chapter 2, and Figure 16.7 shows the principal activities within that process. Some of these activities, such as the initial discovery of user requirements, are carried out in the same way as in other software processes. However, the essential differences between CBSE with reuse and software processes for original software development are as follows:

1. The user requirements are initially developed in outline rather than in detail, and stakeholders are encouraged to be as flexible as possible in defining their requirements. Requirements that are too specific limit the number of components that could meet these requirements. However, unlike incremental development, you need a complete description of the requirements so that you can identify as many components as possible for reuse.

2. Requirements are refined and modified early in the process depending on the components available. If the user requirements cannot be satisfied from available components, you should discuss the related requirements that can be supported by the reusable components. Users may be willing to change their minds if this means cheaper or quicker system delivery.

3. There is a further component search and design refinement activity after the system architecture has been designed. Apparently, usable components may turn out

Figure 16.8 The component identification process

to be unsuitable or may not work properly with other chosen components. You may have to find alternatives to these components. Further requirements changes may therefore be necessary, depending on the functionality of these components.

4. Development is a composition process where the discovered components are integrated. This involves integrating the components with the component model infrastructure and, often, developing adaptors that reconcile the interfaces of incompatible components. Of course, additional functionality may also be required over and above that provided by reused components.

The architectural design stage is particularly important. Jacobsen et al. (Jacobsen, Griss, and Jonsson 1997) found that defining a robust architecture is critical for successful reuse. During the architectural design activity, you may choose a component model and implementation platform. However, many companies have a standard development platform (e.g., .NET), so the component model is predetermined. As I discussed in Chapter 6, you also establish the high-level architecture of the system at this stage and make decisions about system distribution and control.

An activity that is unique to the CBSE process is identifying candidate components or services for reuse. This involves a number of subactivities, as shown in Figure 16.8. Initially, your focus should be on search and selection. You need to convince yourself that components are available to meet your requirements. Obviously, you should do some initial checking that the component is suitable, but detailed testing may not be required. In the later stage, after the system architecture has been designed, you should spend more time on component validation. You need to be confident that the identified components are really suited to your application; if not, then you have to repeat the search and selection processes.

The first step in identifying components is to look for components that are available within your company or from trusted suppliers. There are few component vendors, so you are most likely to be looking for components that have been developed in your own organization or in the repositories of open-source software that are available. Software development companies can build their own database of reusable components without the risks inherent in using components from external suppliers. Alternatively, you may decide to search code libraries available on the web, such as Sourceforge, GitHub, or Google Code, to see if source code for the component that you need is available.

Once the component search process has identified possible components, you have to select candidate components for assessment. In some cases, this will be a straightforward task. Components on the list will directly implement the user requirements, and there will not be competing components that match these requirements. In other cases, however, the selection process is more complex. There will not be a clear mapping of requirements onto components. You may find that several components have to be integrated to meet a

The Ariane 5 launcher failure

While developing the Ariane 5 space launcher, the designers decided to reuse the inertial reference software that had performed successfully in the Ariane 4 launcher. The inertial reference software maintains the stability of the rocket. The designers decided to reuse this without change (as you would do with components), although it included additional functionality that was not required in Ariane 5.

In the first launch of Ariane 5, the inertial navigation software failed, and the rocket could not be controlled. The rocket and its payload were destroyed. The cause of the problem was an unhandled exception when a conversion of a fixed-point number to an integer resulted in a numeric overflow. This caused the runtime system to shut down the inertial reference system, and launcher stability could not be maintained. The fault had never occurred in Ariane 4 because it had less powerful engines and the value that was converted could not be large enough for the conversion to overflow.

This illustrates an important problem with software reuse. Software may be based on assumptions about the context where the system will be used, and these assumptions may not be valid in a different situation.

More information about this failure is available at: http://software-engineering-book.com/case-studies/ariane5/

Figure 16.9 An example of validation failure with reused software

specific requirement or group of requirements. You therefore have to decide which of these component compositions provide the best coverage of the requirements.

Once you have selected components for possible inclusion in a system, you should then validate them to check that they behave as advertised. The extent of the validation required depends on the source of the components. If you are using a component that has been developed by a known and trusted source, you may decide that component testing is unnecessary. You simply test the component when it is integrated with other components. On the other hand, if you are using a component from an unknown source, you should always check and test that component before including it in your system.

Component validation involves developing a set of test cases for a component (or, possibly, extending test cases supplied with that component) and developing a test harness to run component tests. The major problem with component validation is that the component specification may not be sufficiently detailed to allow you to develop a complete set of component tests. Components are usually specified informally, with the only formal documentation being their interface specification. This may not include enough information for you to develop a complete set of tests that would convince you that the component's advertised interface is what you require.

As well as testing that a component for reuse does what you require, you may also have to check that the component does not include malicious code or functionality that you don't need. Professional developers rarely use components from untrusted sources, especially if these sources do not provide source code. Therefore, the malicious code problem does not usually arise. However, reused components may often contain functionality that you don't need, and you have to check that this functionality will not interfere with your use of the component.

The problem with unnecessary functionality is that it may be activated by the component itself. While this may have no effect on the application reusing the component, it can slow down the component, cause it to produce surprising results or, in exceptional cases, cause serious system failures. Figure 16.9 summarizes a situation where the failure of a reused software system, which had unnecessary functionality, led to catastrophic system failure.

The problem in the Ariane 5 launcher arose because the assumptions made about the software for Ariane 4 were invalid for Ariane 5. This is a general problem with reusable components. They are originally implemented for a specific application environment and, naturally, embed assumptions about that environment. These assumptions are rarely documented, so when the component is reused, it is impossible to develop tests to check if the assumptions are still valid. If you are reusing a component in a new environment, you may not discover the embedded environmental assumptions until you use the component in an operational system.

16.3 Component composition

Component composition is the process of integrating components with each other, and with specially written "glue code" to create a system or another component. You can compose components in several different ways, as shown in Figure 16.10. From left to right these diagrams illustrate sequential composition, hierarchical composition, and additive composition. In the discussion below, I assume that you are composing two components (A and B) to create a new component:

1. *Sequential composition* In a sequential composition, you create a new component from two existing components by calling the existing components in sequence. You can think of the composition as a composition of the "provides interfaces." That is, the services offered by component A are called, and the results returned by A are then used in the call to the services offered by component B. The components do not call each other in sequential composition but are called by the external application. This type of composition may be used with embedded or service components.

   Some extra glue code may be required to call the component services in the right order and to ensure that the results delivered by component A are compatible with the inputs expected by component B. The "glue code" transforms these outputs to be of the form expected by component B.

2. *Hierarchical composition* This type of composition occurs when one component calls directly on the services provided by another component. That is, component A calls component B. The called component provides the services that are required by the calling component. Therefore, the "provides" interface of the called component must be compatible with the "requires" interface of the calling component.

   Component A calls on component B directly, and, if their interfaces match, there may be no need for additional code. However, if there is a mismatch between the "requires" interface of A and the "provides" interface of B, then some conversion code may be required. As services do not have a "requires" interface, this mode of composition is not used when components are implemented as services accessed over the web.

Figure 16.10 Types of component composition

3. *Additive composition* This occurs when two or more components are put together (added) to create a new component, which combines their functionality. The "provides" interface and "requires" interface of the new component are a combination of the corresponding interfaces in components A and B. The components are called separately through the external interface of the composed component and may be called in any order. A and B are not dependent and do not call each other. This type of composition may be used with embedded or service components.

You might use all the forms of component composition when creating a system. In all cases, you may have to write "glue code" that links the components. For example, for sequential composition, the output of component A typically becomes the input to component B. You need intermediate statements that call component A, collect the result, and then call component B, with that result as a parameter. When one component calls another, you may need to introduce an intermediate component that ensures that the "provides" interface and the "requires" interface are compatible.

When you write new components especially for composition, you should design the interfaces of these components so that they are compatible with other components in the system. You can therefore easily compose these components into a single unit. However, when components are developed independently for reuse, you will often be faced with interface incompatibilities. This means that the interfaces of the components that you wish to compose are not the same. Three types of incompatibility can occur:

1. *Parameter incompatibility* The operations on each side of the interface have the same name, but their parameter types or the number of parameters are different. In Figure 16.11, the location parameter returned by addressFinder is incompatible with the parameters required by the displayMap and printMap methods in mapDB.

2. *Operation incompatibility* The names of the operations in the provides and "requires" interfaces are different. This is a further incompatibility between the components shown in Figure 16.11.

3. *Operation incompleteness* The "provides" interface of a component is a subset of the "requires" interface of another component, or vice versa.

Figure 16.11 Components with incompatible interfaces

In all cases, you tackle the problem of incompatibility by writing an adaptor that reconciles the interfaces of the two components being reused. An adaptor component converts one interface to another.

The precise form of the adaptor depends on the type of composition. Sometimes, as in the next example, the adaptor takes a result from one component and converts it into a form where it can be used as an input to another. In other cases, the adaptor may be called by component A as a proxy for component B. This situation occurs if A wishes to call B, but the details of the "requires" interface of A do not match the details of the "provides" interface of B. The adaptor reconciles these differences by converting its input parameters from A into the required input parameters for B. It then calls B to deliver the services required by A.

To illustrate adaptors, consider the two simple components shown in Figure 16.11, whose interfaces are incompatible. These might be part of a system used by the emergency services. When the emergency operator takes a call, the phone number is input to the **addressFinder** component to locate the address. Then, using the mapper component, the operator prints a map to be sent to the vehicle dispatched to the emergency.

The first component, **addressFinder**, finds the address that matches a phone number. It can also return the owner of the property associated with the phone number and the type of property. The **mapper** component takes a post code (in the United States, a standard ZIP code with the additional four digits identifying property location) and displays or prints a street map of the area around that code at a specified scale.

These components are composable in principle because the property location includes the post or ZIP code. However, you have to write an adaptor component called **postCodeStripper** that takes the location data from **addressFinder** and strips out the post code. This post code is then used as an input to mapper, and the street map is displayed at a scale of 1:10,000. The following code, which is an example of sequential composition, illustrates the sequence of calls that is required to implement this process:

```
address = addressFinder.location (phonenumber) ;
postCode = postCodeStripper.getPostCode (address) ;
mapper.displayMap(postCode, 10000) ;
```

Another case in which an adaptor component may be used is in hierarchical composition, where one component wishes to make use of another but there is an incompatibility

Figure 16.12 An adaptor linking a data collector and a sensor

between the "provides" interface and "requires" interface of the components in the composition. I have illustrated the use of an adaptor in Figure 16.12 where an adaptor is used to link a data collector and a sensor component. These could be used in the implementation of a wilderness weather station system, as discussed in Chapter 7.

The sensor and data collector components are composed using an adaptor that reconciles the "requires" interface of the data collection component with the "provides" interface of the sensor component. The data collector component has been designed with a generic "requires" interface that supports sensor data collection and sensor management. For each of these operations, the parameter is a text string representing the specific sensor commands. For example, to issue a collect command, you would say sensorData("collect"). As I have shown in Figure 16.12, the sensor itself has separate operations such as start, stop, and getdata.

The adaptor parses the input string, identifies the command (e.g., collect), and then calls Sensor.getdata to collect the sensor value. It then returns the result (as a character string) to the data collector component. This interface style means that the data collector can interact with different types of sensor. A separate adaptor, which converts the sensor commands from Data collector to the sensor interface, is implemented for each type of sensor.

The above discussion of component composition assumes you can tell from the component documentation whether or not interfaces are compatible. Of course, the interface definition includes the operation name and parameter types, so you can make some assessment of the compatibility from this. However, you depend on the component documentation to decide whether the interfaces are semantically compatible.

To illustrate this problem, consider the composition shown in Figure 16.13. These components are used to implement a system that downloads images from a camera and stores them in a photograph library. The system user can provide additional information to describe and catalog the photograph. To avoid clutter, I have not shown all interface

Figure 16.13 Photo library composition

```
— The context keyword names the component to which the conditions apply

context addItem

— The preconditions specify what must be true before execution of addItem
pre:    PhotoLibrary.libSize() > 0
        PhotoLibrary.retrieve(pid) = null

— The postconditions specify what is true after execution
post:   libSize () = libSize()@pre + 1
        PhotoLibrary.retrieve(pid) = p
        PhotoLibrary.catEntry(pid) = photodesc

context delete

pre:    PhotoLibrary.retrieve(pid) <> null ;

post:   PhotoLibrary.retrieve(pid) = null
        PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
        PhotoLibrary.libSize() = libSize()@pre—1
```

Figure 16.14 The OCL description of the Photo Library interface

methods here. Rather, I simply show the methods that are needed to illustrate the component documentation problem. The methods in the interface of Photo Library are:

public void addItem (Identifier pid ; Photograph p; CatalogEntry photodesc) ;
public Photograph retrieve (Identifier pid) ;
public CatalogEntry catEntry (Identifier pid) ;

Assume that the documentation for the addItem method in Photo Library is:

This method adds a photograph to the library and associates the photograph identifier and catalog descriptor with the photograph.

This description appears to explain what the component does, but consider the following questions:

- What happens if the photograph identifier is already associated with a photograph in the library?
- Is the photograph descriptor associated with the catalog entry as well as the photograph? That is, if you delete the photograph, do you also delete the catalog information?

There is not enough information in the informal description of addItem to answer these questions. Of course, it is possible to add more information to the natural language description of the method, but in general, the best way to resolve ambiguities is to use a formal language to describe the interface. The specification shown in Figure 16.14 is part of the description of the interface of Photo Library that adds information to the informal description.

Figure 16.14 shows pre- and postconditions that are defined in a notation based on the object constraint language (OCL), which is part of the UML (Warmer and Kleppe 2003). OCL is designed to describe constraints in UML object models; it allows you to express predicates that must always be true, that must be true before a method has executed; and that must be true after a method has executed. These are invariants, preconditions, and postconditions. To access the value of a variable before an operation, you add @pre after its name. Therefore, using age as an example:

```
age = age@pre + 1
```

This statement means that the value of age after an operation is one more than it was before that operation.

OCL-based approaches are primarily used in model-based software engineering to add semantic information to UML models. The OCL descriptions may be used to drive code generators in model-driven engineering. The general approach has been derived from Meyer's Design by Contract approach (Meyer 1992), in which the interfaces and obligations of communicating objects are formally specified and enforced by the runtime system. Meyer suggests that using Design by Contract is essential if we are to develop trusted components (Meyer 2003).

Figure 16.14 shows the specification for the addItem and delete methods in **Photo Library**. The method being specified is indicated by the keyword **context** and the pre- and postconditions by the keywords **pre** and **post**. The preconditions for **addItem** state that:

1. There must not be a photograph in the library with the same identifier as the photograph to be entered.
2. The library must exist—assume that creating a library adds a single item to it so that the size of a library is always greater than zero.
3. The postconditions for **addItem** state that:

   The size of the library has increased by 1 (so only a single entry has been made).

   If you retrieve using the same identifier, then you get back the photograph that you added.

   If you look up the catalog using that identifier, you get back the catalog entry that you made.

The specification of **delete** provides further information. The precondition states that to delete an item, it must be in the library, and, after deletion, the photo can no longer be retrieved and the size of the library is reduced by 1. However, **delete** does not delete the catalog entry—you can still retrieve it after the photo has been deleted. The reason for this is that you may wish to maintain information in the catalog about why a photo was deleted, its new location, and so on.

When you create a system by composing components, you may find that there are potential conflicts between functional and non-functional requirements, the need to deliver a system as quickly as possible, and the need to create a system that

Figure 16.15 Data collection and report generation components

can evolve as requirements change. You may have to take trade-offs into account for component decisions:

1. What composition of components is most effective for delivering the functional requirements for the system?
2. What composition of the components will make it easier to adapt the composite component when its requirements change?
3. What will be the emergent properties of the composed system? These properties include performance and dependability. You can only assess these properties once the complete system is implemented.

Unfortunately, in many situations the solutions to the composition problems may conflict. For example, consider a situation such as that illustrated in Figure 16.15, where a system can be created through two alternative compositions. The system is a data collection and reporting system where data is collected from different sources, stored in a database, and then different reports summarizing that data are produced.

Here, there is a potential conflict between adaptability and performance. Composition (a) is more adaptable, but composition (b) is likely to be faster and more reliable. The advantages of composition (a) are that reporting and data management are separate, so there is more flexibility for future change. The data management system could be replaced, and, if reports are required that the current reporting component cannot produce, that component can also be replaced without having to change the data management component.

In composition (b), a database component with built-in reporting facilities (e.g., Microsoft Access) is used. The key advantage of composition (b) is that there are fewer components, so this will be a faster implementation because there are no component communication overheads. Furthermore, data integrity rules that apply to the database will also apply to reports. These reports will not be able to combine data in incorrect ways. In composition (a), there are no such constraints, so errors in reports could occur.

In general, a good composition principle to follow is the principle of separation of concerns. That is, you should try to design your system so that each component has a clearly defined role. Ideally, component roles should not overlap. However, it may be cheaper to buy one multifunctional component rather than two or three separate components. Furthermore, dependability or performance penalties may be incurred when multiple components are used.

KEY POINTS

- Component-based software engineering is a reuse-based approach to defining, implementing, and composing loosely coupled independent components into systems.
- A component is a software unit whose functionality and dependencies are completely defined by a set of public interfaces. Components can be composed with other components without knowledge of their implementation and can be deployed as an executable unit.
- Components may be implemented as executable routines that are included in a system or as external services that are referenced from within a system.
- A component model defines a set of standards for components, including interface standards, usage standards, and deployment standards. The implementation of the component model provides a set of common services that may be used by all components.
- During the CBSE process, you have to interleave the processes of requirements engineering and system design. You have to trade off desirable requirements against the services that are available from existing reusable components.
- Component composition is the process of "wiring" components together to create a system. Types of composition include sequential composition, hierarchical composition, and additive composition.
- When composing reusable components that have not been written for your application, you may need to write adaptors or "glue code" to reconcile the different component interfaces.
- When choosing compositions, you have to consider the required functionality of the system, the non-functional requirements, and the ease with which one component can be replaced when the system is changed.

FURTHER READING

Component Software: Beyond Object-Oriented Programming, 2nd ed. This updated edition of the first book on CBSE covers technical and nontechnical issues in CBSE. It has more detail on specific technologies than Heineman and Councill's book and includes a thorough discussion of market issues. (C. Szyperski, Addison-Wesley, 2002).

"Specification, Implementation and Deployment of Components." A good introduction to the fundamentals of CBSE. The same issue of the *CACM* includes articles on components and component-based development. (I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan, *Comm. ACM,* 45(10), October 2002) http://dx.doi.org/10.1145/570907.570928

"Software Component Models." This comprehensive discussion of commercial and research component models classifies these models and explains the differences between them. (K-K. Lau and Z. Wang, *IEEE Transactions on Software Engineering,* 33 (10), October 2007) http://dx.doi.org/10.1109/TSE.2007.70726

"Software Components Beyond Programming: From Routines to Services." This is the opening article in a special issue of the magazine that includes several articles on software components. This article discusses the evolution of components and how service-oriented components are replacing executable program routines. (I. Crnkovic, J. Stafford, and C. Szyperski, *IEEE Software*, 28 (3), May/June 2011) http://dx.doi.org/10.1109/MS.2011.62

Object Constraint Language (OCL) Tutorial. A good introduction to the use of the object-constraint language. (J. Cabot, 2012) http://modeling-languages.com/ocl-tutorial/

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-reuse/

A more detailed discussion of the Ariane5 accident:

http://software-engineering-book.com/case-studies/ariane5/

EXERCISES

16.1. What are the design principles underlying the CBSE that support the construction of understandable and maintainable software?

16.2. The principle of component independence means that it ought to be possible to replace one component with another that is implemented in a completely different way. Using an example, explain how such component replacement could have undesired consequences and may lead to system failure.

16.3. In a reusable component, what are the critical characteristics that are emphasized when the component is viewed as a service?

16.4. Why is it important that components should be based on a standard component model?

16.5. Using an example of a component that implements an abstract data type such as a stack or a list, show why it is usually necessary to extend and adapt components for reuse.

16.6. What are the essential differences between CBSE with reuse and software processes for original software development?

16.7. Design the "provides" interface and the "requires" interface of a reusable component that may be used to represent a patient in the Mentcare system that I introduced in Chapter 1.

16.8. Using examples, illustrate the different types of adaptor needed to support sequential composition, hierarchical composition, and additive composition.

16.9. Design the interfaces of components that might be used in a system for an emergency control room. You should design interfaces for a call-logging component that records calls made, and a vehicle discovery component that, given a post code (zip code) and an incident type, finds the nearest suitable vehicle to be dispatched to the incident.

16.10. It has been suggested that an independent certification authority should be established. Vendors would submit their components to this authority, which would validate that the component was trustworthy. What would be the advantages and disadvantages of such a certification authority?

REFERENCES

Councill, W. T., and G. T. Heineman. 2001. "Definition of a Software Component and Its Elements." In *Component-Based Software Engineering*, edited by G. T. Heineman and W. T. Councill, 5–20. Boston: Addison-Wesley.

Jacobsen, I., M. Griss, and P. Jonsson. 1997. *Software Reuse*. Reading, MA: Addison-Wesley.

Kotonya, G. 2003. "The CBSE Process: Issues and Future Visions." In *2nd CBSEnet Workshop*. Budapest, Hungary. http://miro.sztaki.hu/projects/cbsenet/budapest/presentations/Gerald-CBSEProcess.ppt

Lau, K-K., and Z. Wang. 2007. "Software Component Models." *IEEE Trans. on Software Eng.* 33 (10): 709–724. doi:10.1109/TSE.2007.70726.

Meyer, B. 1992. "Applying Design by Contract." *IEEE Computer* 25 (10): 40–51. doi:10.1109/2.161279.

———. 2003. "The Grand Challenge of Trusted Components." In *Proc. 25th Int. Conf. on Software Engineering*. Portland, OR: IEEE Press. doi:10.1109/ICSE.2003.1201252.

Mili, H., A. Mili, S. Yacoub, and E. Addy. 2002. *Reuse-Based Software Engineering*. New York: John Wiley & Sons.

Pope, A. 1997. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Harlow, UK: Addison-Wesley.

Szyperski, C. 2002. *Component Software: Beyond Object-Oriented Programming, 2nd ed.* Harlow, UK: Addison-Wesley.

Warmer, J., and A. Kleppe. 2003. *The Object Constraint Language: Getting Your Models Ready for MDA*. Boston: Addison-Wesley.

Weinreich, R., and J. Sametinger. 2001. "Component Models and Component Services: Concepts and Principles." In *Component-Based Software Engineering*, edited by G. T. Heineman and W. T. Councill, 33–48. Boston: Addison-Wesley.

Wheeler, W., and J. White. 2013. *Spring in Practice*. Greenwich, CT: Manning Publications.

17

Distributed software engineering

Objectives

The objective of this chapter is to introduce distributed systems engineering and distributed systems architectures. When you have read this chapter, you will:

- know the key issues that have to be considered when designing and implementing distributed software systems;
- understand the client–server computing model and the layered architecture of client–server systems;
- have been introduced to commonly used patterns for distributed systems architectures and know the types of system for which each architectural pattern is applicable;
- understand the notion of software as a service, providing web-based access to remotely deployed application systems.

Contents

17.1 Distributed systems
17.2 Client–server computing
17.3 Architectural patterns for distributed systems
17.4 Software as a service

Most computer-based systems are now distributed systems. A distributed system is one involving several computers rather than a single application running on a single machine. Even apparently self-contained applications on a PC or laptop, such as image editors, are distributed systems. They execute on a single computer system but often rely on remote cloud systems for update, storage, and other services. Tanenbaum and Van Steen (Tanenbaum and Van Steen 2007) define a distributed system to be "a collection of independent computers that appears to the user as a single coherent system."†

When you are designing a distributed system, there are specific issues that have to be taken into account simply because the system is distributed. These issues arise because different parts of the system are running on independently managed computers and because the characteristics of the network, such as latency and reliability, may have to be considered in your design.

Coulouris et al. (Coulouris et al. 2011) identify the five benefits of developing systems as distributed systems:

1. *Resource sharing* A distributed system allows the sharing of hardware and software resources—such as disks, printers, files, and compilers—that are associated with computers on a network.
2. *Openness* Distributed systems are normally open systems—systems designed around standard Internet protocols so that equipment and software from different vendors can be combined.
3. *Concurrency* In a distributed system, several processes may operate at the same time on separate computers on the network. These processes may (but need not) communicate with each other during their normal operation.
4. *Scalability* In principle at least, distributed systems are scalable in that the capabilities of the system can be increased by adding new resources to cope with new demands on the system. In practice, the network linking the individual computers in the system may limit the system scalability.
5. *Fault tolerance* The availability of several computers and the potential for replicating information means that distributed systems can be tolerant of some hardware and software failures (see Chapter 11). In most distributed systems, a degraded service can be provided when failures occur; complete loss of service only occurs when there is a network failure.‡

Distributed systems are inherently more complex than centralized systems. This makes them more difficult to design, implement, and test. It is harder to understand the emergent properties of distributed systems because of the complexity of the interactions between system components and system infrastructure. For example, rather than being dependent on the execution speed of one processor, system performance

†Tanenbaum, A. S., and M. Van Steen. 2007. Distributed Systems: Principles and Paradigms, 2nd Ed. Upper Saddle River, NJ: Prentice-Hall.

‡Coulouris, G., J. Dollimore, T. Kindberg, and G. Blair. 2011. Distributed Systems: Concepts and Design, 5th Edition. Harlow, UK.: Addison Wesley.

depends on network bandwidth, network load, and the speed of other computers that are part of the system. Moving resources from one part of the system to another can significantly affect the system's performance.

Furthermore, as all users of the WWW know, distributed systems are unpredictable in their response. Response time depends on the overall load on the system, its architecture, and the network load. As all of these factors may change over a short time, the time taken to respond to a user request may change significantly from one request to another.

The most important developments that have affected distributed software systems in the past few years are service-oriented systems and the advent of cloud computing, delivering infrastructure, platforms, and software as a service. In this chapter, I focus on general issues of distributed systems, and in Section 17.4 I cover the idea of software as a service. In Chapter 18, I discuss other aspects of service-oriented software engineering.

17.1 Distributed systems

As I discussed in the introduction to this chapter, distributed systems are more complex than systems that run on a single processor. This complexity arises because it is practically impossible to have a top-down model of control of these systems. The nodes in the system that deliver functionality are often independent systems that are managed and controlled by their owners. There is no single authority in charge of the entire distributed system. The network connecting these nodes is also a separately managed system. It is a complex system in its own right and cannot be controlled by the owners of systems using the network. There is, therefore, an inherent unpredictability in the operation of distributed systems that has to be taken into account when you are designing a system.

Some of the most important design issues that have to be considered in distributed systems engineering are:

1. *Transparency* To what extent should the distributed system appear to the user as a single system? When is it useful for users to understand that the system is distributed?
2. *Openness* Should a system be designed using standard protocols that support interoperability, or should more specialized protocols be used? Although standard network protocols are now universally used, this is not the case for higher levels of interaction, such as service communication.
3. *Scalability* How can the system be constructed so that it is scalable? That is, how can the overall system be designed so that its capacity can be increased in response to increasing demands made on the system?
4. *Security* How can usable security policies be defined and implemented that apply across a set of independently managed systems?
5. *Quality of service* How should the quality of service that is delivered to system users be specified, and how should the system be implemented to deliver an acceptable quality of service to all users.
6. *Failure management* How can system failures be detected, contained (so that they have minimal effects on other components in the system), and repaired?

CORBA—Common Object Request Broker Architecture

CORBA was proposed as a specification for a middleware system in the 1990s by the Object Management Group. It was intended as an open standard that would allow the development of middleware to support distributed component communications and execution, as well as provide a set of standard services that could be used by these components.

Several implementations of CORBA were produced, but the system was not widely adopted. Users preferred proprietary systems such as those from Microsoft or Oracle, or they moved to service-oriented architectures.

http://software-engineering-book.com/web/corba/

In an ideal world, the fact that a system is distributed would be transparent to users. Users would see the system as a single system whose behavior is not affected by the way that the system is distributed. In practice, this is impossible to achieve because there is no central control over the system as a whole. As a result, individual computers in a system may behave differently at different times. Furthermore, because it always takes a finite length of time for signals to travel across a network, network delays are unavoidable. The length of these delays depends on the location of resources in the system, the quality of the user's network connection, and the network load.

To make a distributed system transparent (i.e., conceal its distributed nature), you have to hide the underlying distribution. You create abstractions that hide the system resources so that the location and implementation of these resources can be changed without having to change the distributed application. Middleware (discussed in Section 17.1.2) is used to map the logical resources referenced by a program onto the actual physical resources and to manage resource interactions.

In practice, it is impossible to make a system completely transparent, and users, generally, are aware that they are dealing with a distributed system. You may therefore decide that it is best to expose the distribution to users. They can then be prepared for some of the consequences of distribution such as network delays and remote node failures.

Open distributed systems are built according to generally accepted standards. Components from any supplier can therefore be integrated into the system and can interoperate with the other system components. At the networking level, openness is now taken for granted, with systems conforming to Internet protocols, but at the component level, openness is still not universal. Openness implies that system components can be independently developed in any programming language and, if these conform to standards, they will work with other components.

The CORBA standard (Pope 1997), developed in the 1990s, was intended to be the universal standard for open distributed systems, However, the CORBA standard never achieved a critical mass of adopters. Rather, many companies preferred to develop systems using proprietary standards for components from companies such as Sun (now Oracle) and Microsoft. These provided better implementations and support software and better long-term support for industrial protocols.

Web service standards (discussed in Chapter 18) for service-oriented architectures were developed to be open standards. However, these standards have met with significant resistance because of their perceived inefficiency. Many developers of service-based systems have opted instead for so-called RESTful protocols because

these have an inherently lower overhead than web service protocols. The use of RESTful protocols is not standardized.

The scalability of a system reflects its ability to deliver high-quality service as demands on the system increase. The three dimensions of scalability are size, distribution, and manageability.

1. *Size* It should be possible to add more resources to a system to cope with increasing numbers of users. Ideally, then, as the number of users increases, the system should increase in size automatically to handle the increased number of users.

2. *Distribution* It should be possible to geographically disperse the components of a system without degrading its performance. As new components are added, it should not matter where these are located. Large companies can often make use of computing resources in their different facilities around the world.

3. *Manageability* It should be possible to manage a system as it increases in size, even if parts of the system are located in independent organizations. This is one of the most difficult challenges of scale as it involves managers communicating and agreeing on management policies. In practice, the manageability of a system is often the factor that limits the extent to which it can be scaled.

Changing the size of a system may involve either scaling up or scaling out. Scaling up means replacing resources in the system with more powerful resources. For example, you may increase the memory in a server from 16 Gb to 64 Gb. Scaling out means adding more resources to the system (e.g., an extra web server to work alongside an existing server). Scaling out is often more cost-effective than scaling up, especially now that cloud computing makes it easy to add or remove servers from a system. However, this only provides performance improvements when concurrent processing is possible.

I have discussed general security issues and issues of security engineering in Part 2 of this book. When a system is distributed, attackers may target any of the individual system components or the network itself. If a part of the system is successfully attacked, then the attacker may be able to use this as a "back door" into other parts of the system.

A distributed system must defend itself against the following types of attack:

1. *Interception*, where an attacker intercepts communications between parts of the system so that there is a loss of confidentiality.

2. *Interruption*, where system services are attacked and cannot be delivered as expected. Denial-of-service attacks involve bombarding a node with illegitimate service requests so that it cannot deal with valid requests.

3. *Modification*, where an attacker gains access to the system and changes data or system services.

4. *Fabrication*, where an attacker generates information that should not exist and then uses this information to gain some privileges. For example, an attacker may generate a false password entry and use this to gain access to a system.

The major difficulty in distributed systems is establishing a security policy that can be reliably applied to all of the components in a system. As I discussed in Chapter 13, a security policy sets out the level of security to be achieved by a system. Security mechanisms, such as encryption and authentication, are used to enforce the security policy. The difficulties in a distributed system arise because different organizations may own parts of the system. These organizations may have mutually incompatible security policies and security mechanisms. Security compromises may have to be made in order to allow the systems to work together.

The quality of service (QoS) offered by a distributed system reflects the system's ability to deliver its services dependably and with a response time and throughput that are acceptable to its users. Ideally, the QoS requirements should be specified in advance and the system designed and configured to deliver that QoS. Unfortunately, this is not always practicable for two reasons:

1. It may not be cost-effective to design and configure the system to deliver a high quality of service under peak load. The peak demands may mean that you need many extra servers than normal to ensure that response times are maintained. This problem has been lessened by the advent of cloud computing where cloud servers may be rented from a cloud provider for as long as they are required. As demand increases, extra servers can be automatically added.
2. The quality-of-service parameters may be mutually contradictory. For example, increased reliability may mean reduced throughput, as checking procedures are introduced to ensure that all system inputs are valid.

Quality of service is particularly important when the system is dealing with time-critical data such as sound or video streams. In these circumstances, if the quality of service falls below a threshold value then the sound or video may become so degraded that it is impossible to understand. Systems dealing with sound and video should include quality of service negotiation and management components. These should evaluate the QoS requirements against the available resources and, if these are insufficient, negotiate for more resources or for a reduced QoS target.

In a distributed system, it is inevitable that failures will occur, so the system has to be designed to be resilient to these failures. Failure is so ubiquitous that one flippant definition of a distributed system suggested by Leslie Lamport, a prominent distributed systems researcher, is:

> *You know that you have a distributed system when the crash of a system that you've never heard of stops you getting any work done.*[†]

This is even truer now that more and more systems are executing in the cloud. Failure management involves applying the fault-tolerance techniques discussed in Chapter 11. Distributed systems should therefore include mechanisms for discovering whether a component of the system has failed, should continue to deliver as many services as possible in spite of that failure, and, as far as possible, should automatically

[†]Leslie Lamport, in Ross J. Anderson, Security Engineering: A Guide to Building Dependable Distributed Systems (2nd ed.), Wiley (April 14, 2008).

Figure 17.1 Procedural interaction between a diner and a waiter

recover from the failure. One important benefit of cloud computing is that it has dramatically reduced the cost of providing redundant system components.

17.1.1 Models of interaction

Two fundamental types of interaction may take place between the computers in a distributed computing system: procedural interaction and message-based interaction. Procedural interaction involves one computer calling on a known service offered by some other computer and waiting for that service to be delivered. Message-based interaction involves the "sending" computer defining information about what is required in a message, which is then sent to another computer. Messages usually transmit more information in a single interaction than a procedure call to another machine.

To illustrate the difference between procedural and message-based interaction, consider a situation where you are ordering a meal in a restaurant. When you have a conversation with the waiter, you are involved in a series of synchronous, procedural interactions that define your order. You make a request, the waiter acknowledges that request, you make another request, which is acknowledged, and so on. This is comparable to components interacting in a software system where one component calls methods from other components. The waiter writes down your order along with the order of other people with you. He or she then passes this order, which includes details of everything that has been ordered, to the kitchen to prepare the food. Essentially, the waiter is passing a message to the kitchen staff, defining the food to be prepared. This is message-based interaction.

I have illustrated this kind of interaction in Figure 17.1, which shows the synchronous ordering process as a series of calls, and in Figure 17.2, which shows a hypothetical XML message that defines an order made by the table of three people. The difference between these forms of information exchange is clear. The waiter takes the order as a series of

```
<starter>
   <dish name = "soup" type = "tomato" />
   <dish name = "soup" type = "fish" />
   <dish name = "pigeon salad" />
</starter>
<main course>
   <dish name = "steak" type = "sirloin" cooking = "medium" />
   <dish name = "steak" type = "fillet" cooking = "rare" />
   <dish name = "sea bass">
</main>
<accompaniment>
   <dish name = "french fries" portions = "2" />
   <dish name = "salad" portions = "1" />
</accompaniment>
```

Figure 17.2 Message-based interaction between a waiter and the kitchen staff

interactions, with each interaction defining part of the order. However, the waiter has a single interaction with the kitchen where the message defines the complete order.

Procedural communication in a distributed system is usually implemented using remote procedure calls (RPCs). In an RPC, components have globally unique names (such as a URL). Using that name, a component can call on the services offered by another component as if it was a local procedure or method. System middleware intercepts this call and passes it on to a remote component. This carries out the required computation and, via the middleware, returns the result to the calling component. In Java, remote method invocations (RMIs) are remote procedure calls.

Remote procedure calls require a "stub" for the called procedure to be accessible on the computer that is initiating the call. This stub defines the interface of the remote procedure. The stub is called, and it translates the procedure parameters into a standard representation for transmission to the remote procedure. Through the middleware, it then sends the request for execution to the remote procedure. The remote procedure uses library functions to convert the parameters into the required format, carries out the computation, and then returns the results via the "stub" that is representing the caller.

Message-based interaction normally involves one component creating a message that details the services required from another component. This message is sent to the receiving component via the system middleware. The receiver parses the message, carries out the computations, and creates a message for the sending component with the required results. This is then passed to the middleware for transmission to the sending component.

A problem with the RPC approach to interaction is that both the caller and the callee need to be available at the time of the communication, and they must know how to refer to each other. In essence, an RPC has the same requirements as a local procedure or method call. By contrast, in a message-based approach, unavailability can be tolerated. If the system component that is processing the message is unavailable, the message simply stays in a queue until the receiver comes back online. Furthermore, it is not necessary for the sender to know the name of the message receiver and vice versa. They simply communicate with the middleware, which is responsible for ensuring that messages are passed to the appropriate system.

Figure 17.3 Middleware in a distributed system

17.1.2 Middleware

The components in a distributed system may be implemented in different programming languages and may execute on different types of processors. Models of data, information representation, and protocols for communication may all be different. A distributed system therefore requires software that can manage these diverse parts and ensure that they can communicate and exchange data.

The term *middleware* is used to refer to this software—it sits in the middle between the distributed components of the system. This concept is illustrated in Figure 17.3, which shows that middleware is a layer between the operating system and application programs. Middleware is normally implemented as a set of libraries, which are installed on each distributed computer, plus a runtime system to manage communications.

Bernstein (Bernstein 1996) describes types of middleware that are available to support distributed computing. Middleware is general-purpose software that is usually bought off-the-shelf rather than written specially by application developers. Examples of middleware include software for managing communications with databases, transaction managers, data converters, and communication controllers.

In a distributed system, middleware provides two distinct types of support:

1. *Interaction support*, where the middleware coordinates interactions between different components in the system. The middleware provides location transparency in that it isn't necessary for components to know the physical locations of other components. It may also support parameter conversion if different programming languages are used to implement components, event detection, communication, and so on.

2. *The provision of common services*, where the middleware provides reusable implementations of services that may be required by several components in the distributed system. By using these common services, components can easily interoperate and provide user services in a consistent way.

I have already given examples of the interaction support that middleware can provide in Section 17.1.1. You use middleware to support remote procedure and remote method calls, message exchange, and so forth.

Common services are those services that may be required by different components irrespective of the functionality of these components. As I discussed in Chapter 16, these may include security services (authentication and authorization), notification and naming services, and transaction management services. For distributed components, you can think of these common services as being provided by a middleware container; for services, they are provided through shared libraries. You then deploy your component, and it can access and use these common services.

17.2 Client–server computing

Distributed systems that are accessed over the Internet are organized as client–server systems. In a client–server system, the user interacts with a program running on their local computer, such as a web browser or app on a mobile device. This interacts with another program running on a remote computer, such as a web server. The remote computer provides services, such as access to web pages, which are available to external clients. This client–server model, as I discussed in Chapter 6, is a general architectural model of an application. It is not restricted to applications distributed across several machines. You can also use it as a logical interaction model where the client and the server run on the same computer.

In a client–server architecture, an application is modeled as a set of services that are provided by servers. Clients may access these services and present results to end-users. Clients need to be aware of the servers that are available but don't have to know anything about other clients. Clients and servers are separate processes, as shown in Figure 17.4. This figure illustrates a situation in which there are four servers (s1–s4) that deliver different services. Each service has a set of associated clients that access these services.

Figure 17.4 shows client and server processes rather than processors. It is normal for several client processes to run on a single processor. For example, on your PC, you may run a mail client that downloads mail from a remote mail server. You may also run a web browser that interacts with a remote web server and a print client that sends documents to a remote printer. Figure 17.5 shows a possible arrangement where the 12 logical clients shown in Figure 17.4 are running on six computers. The four server processes are mapped onto two physical server computers.

Several different server processes may run on the same processor, but, often, servers are implemented as multiprocessor systems in which a separate instance of the server process runs on each machine. Load-balancing software distributes requests for service from clients to different servers so that each server does the same amount of work. This allows a higher volume of transactions with clients to be handled, without degrading the response to individual clients.

Client–server systems depend on there being a clear separation between the presentation of information and the computations that create and process that information. Consequently, you should design the architecture of distributed client–server systems so that they are structured into several logical layers, with clear interfaces

Figure 17.4 Client–server interaction

between these layers. This allows each layer to be distributed to a different computer. Figure 17.6 illustrates this model, showing an application structured into four layers:

1. *A presentation layer* that is concerned with presenting information to the user and managing all user interaction.
2. *A data-handling layer* that manages the data that is passed to and from the client. This layer may implement checks on the data, generate web pages, and so on.
3. *An application processing layer* that is concerned with implementing the logic of the application and so providing the required functionality to end-users.
4. *A database layer* that stores the data and provides transaction management and query services.

The following section explains how different client–server architectures distribute these logical layers in different ways. The client–server model also underlies the notion of software as a service (SaaS), an important way of deploying software and accessing it over the Internet. I cover this topic in Section 17.4.

Figure 17.5 Mapping of clients and servers to networked computers

Figure 17.6 Layered architectural model for client–server application

17.3 Architectural patterns for distributed systems

As I explained in the introduction to this chapter, designers of distributed systems have to organize their system designs to find a balance between performance, dependability, security, and manageability of the system. Because no universal model of system organization is appropriate for all circumstances, various distributed architectural styles have emerged. When designing a distributed application, you should choose an architectural style that supports the critical non-functional requirements of your system.

In this section, I discuss five architectural styles:

1. *Master-slave architecture*, which is used in real-time systems in which guaranteed interaction response times are required.
2. *Two-tier client–server architecture*, which is used for simple client–server systems and in situations where it is important to centralize the system for security reasons.
3. *Multi-tier client–server architecture*, which is used when the server has to process a high volume of transactions.
4. *Distributed component architecture*, which is used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client–server systems.
5. *Peer-to-peer architecture*, which is used when clients exchange locally stored information and the role of the server is to introduce clients to each other. It may also be used when a large number of independent computations may have to be made.

17.3.1 Master–slave architectures

Master–slave architectures for distributed systems are commonly used in real-time systems. In those systems, there may be separate processors associated with data acquisition from the system's environment, data processing and computation,

Figure 17.7 A traffic management system with a master–slave architecture

and actuator management. Actuators, as I discuss in Chapter 21, are devices controlled by the software system that act to change the system's environment. For example, an actuator may control a valve and change its state from "open" to "closed." The "master" process is usually responsible for computation, coordination, and communications, and it controls the "slave" processes. "Slave" processes are dedicated to specific actions, such as the acquisition of data from an array of sensors.

Figure 17.7 shows an example of this architectural model. A traffic control system in a city has three logical processes that run on separate processors. The master process is the control room process, which communicates with separate slave processes that are responsible for collecting traffic data and managing the operation of traffic lights.

A set of distributed sensors collects information on the traffic flow. The sensor control process polls the sensors periodically to capture the traffic flow information and collates this information for further processing. The sensor processor is itself polled periodically for information by the master process that is concerned with displaying traffic status to operators, computing traffic light sequences, and accepting operator commands to modify these sequences. The control room system sends commands to a traffic light control process that converts these into signals to control the traffic light hardware. The master control room system is itself organized as a client–server system, with the client processes running on the operator's consoles.

You use this master–slave model of a distributed system in situations where you can predict the distributed processing that is required and where processing can be easily localized to slave processors. This situation is common in real-time systems, where it is important to meet processing deadlines. Slave processors can be used for computationally intensive operations, such as signal processing and the management of equipment controlled by the system.

Figure 17.8 Thin- and fat-client architectural models

17.3.2 Two-tier client–server architectures

In Section 17.2, I explained the general organization of client–server systems in which part of the application system runs on the user's computer (the client), and part runs on a remote computer (the server). I also presented a layered application model (Figure 17.6) where the different layers in the system may execute on different computers.

A two-tier client–server architecture is the simplest form of client–server architecture. The system is implemented as a single logical server plus an indefinite number of clients that use that server. This is illustrated in Figure 17.8, which shows two forms of this architectural model:

1. *A thin-client model*, where the presentation layer is implemented on the client and all other layers (data handling, application processing, and database) are implemented on a server. The client presentation software is usually a web browser, but apps for mobile devices may also be available.

2. *A fat-client model*, where some or all of the application processing is carried out on the client. Data management and database functions are implemented on the server. In this case, the client software may be a specially written program that is tightly integrated with the server application.

The advantage of the thin-client model is that it is simple to manage the clients. This becomes a major issue when there are a large number of clients, as it may be difficult and expensive to install new software on all of them. If a web browser is used as the client, there is no need to install any software.

The disadvantage of the thin-client approach, however, is that it places a heavy processing load on both the server and the network. The server is responsible for all computation, which may lead to the generation of significant network traffic between the client and the server. Implementing a system using this model may therefore require additional investment in network and server capacity.

The fat-client model makes use of available processing power on the computer running the client software, and distributes some or all of the application processing

Figure 17.9 A fat-client architecture for an ATM system

and the presentation to the client. The server is essentially a transaction server that manages all database transactions. Data handling is straightforward as there is no need to manage the interaction between the client and the application processing system. The fat-client model requires system management to deploy and maintain the software on the client computer.

An example of a situation in which a fat-client architecture is used is in a bank ATM system, which delivers cash and other banking services to users. The ATM is the client computer, and the server is, typically, a mainframe running the customer account database. A mainframe computer is a powerful machine that is designed for transaction processing. It can therefore handle the large volume of transactions generated by ATMs, other teller systems, and online banking. The software in the teller machine carries out a lot of the customer-related processing associated with a transaction.

Figure 17.9 shows a simplified version of the ATM system organization. The ATMs do not connect directly to the customer database, but rather to a teleprocessing (TP) monitor. A TP monitor is a middleware system that organizes communications with remote clients and serializes client transactions for processing by the database. This ensures that transactions are independent and do not interfere with one other. Using serial transactions means that the system can recover from faults without corrupting the system data.

While a fat-client model distributes processing more effectively than a thin-client model, system management is more complex if a special-purpose client, rather than a browser, is used. Application functionality is spread across many computers. When the application software has to be changed, this involves software reinstallation on every client computer. This can be a major cost if there are hundreds of clients in the system. Auto-update of the client software can reduce these costs but introduces its own problems if the client functionality is changed. The new functionality may mean that businesses have to change the ways they use the system.

The extensive use of mobile devices means that it is important to mimimize network traffic wherever possible. These devices now include powerful computers that can carry out local processing. As a consequence, the distinction between thin-client and fat-client architectures has become blurred. Apps can have inbuilt functionality that carries out local processing, and web pages may include Javascript components

Figure 17.10 Three-tier architecture for an Internet banking system

that execute on the user's local computer. The update problem for apps remains an issue, but it has been addressed, to some extent, by automatically updating apps without explicit user intervention. Consequently, while it is sometimes helpful to use these models as a general basis for the architecture of a distributed system, in practice few web-based applications implement all processing on the remote server.

17.3.3 Multi-tier client–server architectures

The fundamental problem with a two-tier client–server approach is that the logical layers in the system—presentation, application processing, data management, and database—must be mapped onto two computer systems: the client and the server. This may lead to problems with scalability and performance if the thin-client model is chosen, or problems of system management if the fat-client model is used. To avoid some of these problems, a "multi-tier client–server" architecture can be used. In this architecture, the different layers of the system, namely presentation, data management, application processing, and database, are separate processes that may execute on different processors.

An Internet banking system (Figure 17.10) is an example of a multi-tier client–server architecture, where there are three tiers in the system. The bank's customer database (usually hosted on a mainframe computer as discussed above) provides database services. A web server provides data management services such as web page generation and some application services. Application services such as facilities to transfer cash, generate statements, pay bills, and so on are implemented in the web server and as scripts that are executed by the client. The user's own computer with an Internet browser is the client. This system is scalable because it is relatively easy to add servers (scale out) as the number of customers increase.

In this case, the use of a three-tier architecture allows the information transfer between the web server and the database server to be optimized. Efficient middleware that supports database queries in SQL (Structured Query Language) is used to handle information retrieval from the database.

Architecture	Applications
Two-tier client–server architecture with thin clients	Legacy system applications that are used when separating application processing and data handling is impractical. Clients may access these as services, as discussed in Section 17.4. Computationally intensive applications such as compilers with little or no requirements for data handling. Data-intensive applications (browsing and querying) with non-intensive application processing. Simple web browsing is the most common example of a situation where this architecture is used.
Two-tier client–server architecture with fat clients	Applications where application processing is provided by off-the-shelf software (e.g., Microsoft Excel) on the client. Applications where computationally intensive processing of data (e.g., data visualization) is required. Mobile applications where internet connectivity cannot be guaranteed. Local processing using cached information from the database is therefore possible.
Multi-tier client–server architecture	Large-scale applications with hundreds or thousands of clients. Applications where both the data and the application are volatile. Applications where data from multiple sources are integrated.

Figure 17.11 Use of client–server architectural patterns

The three-tier client–server model can be extended to a multi-tier variant, where additional servers are added to the system. This may involve using a web server for data management and separate servers for application processing and database services. Multi-tier systems may also be used when applications need to access and use data from different databases. In this case, you may need to add an integration server to the system. The integration server collects the distributed data and presents it to the application server as if it were from a single database. As I discuss in the following section, distributed component architectures may be used to implement multi-tier client–server systems.

Multi-tier client–server systems that distribute application processing across several servers are more scalable than two-tier architectures. The tiers in the system can be independently managed, with additional servers added as the load increases. Processing may be distributed between the application logic and the data-handling servers, thus leading to more rapid response to client requests.

Designers of client–server architectures must take a number of factors into account when choosing the most appropriate distribution architecture. Situations in which the client–server architectures discussed here may be used are described in Figure 17.11.

17.3.4 Distributed component architectures

By organizing processing into layers, as shown in Figure 17.6, each layer of a system can be implemented as a separate logical server. This model works well for many types of application. However, it limits the flexibility of system designers in that they

Figure 17.12 A distributed component architecture

have to decide what services should be included in each layer. In practice, however, it is not always clear whether a service is a data management service, an application service, or a database service. Designers must also plan for scalability and so provide some means for servers to be replicated as more clients are added to the system.

A more general approach to distributed system design is to design the system as a set of services, without attempting to allocate these services to layers in the system. Each service, or group of related services, can be implemented using a separate object or component. In a distributed component architecture (Figure 17.12), the system is organized as a set of interacting components as I discussed in Chapter 16. These components provide an interface to a set of services that they provide. Other components call on these services through middleware, using remote procedure or method calls.

Distributed component systems are reliant on middleware. This manages component interactions, reconciles differences between types of the parameters passed between components, and provides a set of common services that application components can use. The CORBA standard (Orfali, Harkey, and Edwards 1997) defined middleware for distributed component systems, but CORBA implementations have never been widely adopted. Enterprises preferred to use proprietary software such as Enterprise Java Beans (EJB) or .NET.

Using a distributed component model for implementing distributed systems has a number of benefits:

1. It allows the system designer to delay decisions on where and how services should be provided. Service-providing components may execute on any node of the network. There is no need to decide in advance whether a service is part of a data management layer, an application layer, or a user interface layer.

2. It is a very open-system architecture that allows new resources to be added as required. New system services can be added easily without major disruption to the existing system.

3. The system is flexible and scalable. New objects or replicated objects can be added as the load on the system increases, without disrupting other parts of the system.

Figure 17.13 A distributed component architecture for a data-mining system

4. It is possible to reconfigure the system dynamically with components migrating across the network as required. This may be important where there are fluctuating patterns of demand on services. A service-providing component can migrate to the same processor as service-requesting objects, thus improving the performance of the system.

A distributed component architecture can be used as a logical model that allows you to structure and organize the system. In this case, you think about how to provide application functionality solely in terms of services and combinations of services. You then work out how to implement these services. For example, a retail application may have application components concerned with stock control, customer communications, goods ordering, and so on.

Data-mining systems are a good example of a type of system that can be implemented using a distributed component architecture. Data-mining systems look for relationships between the data that may be distributed across databases (Figure 17.13). These systems pull in information from several separate databases, carry out computationally intensive processing, and present easy-to-understand visualizations of the relationships that have been discovered.

An example of such a data-mining application might be a system for a retail business that sells food and books. Retail businesses maintain separate databases with detailed information about food products and books. They use a loyalty card system to keep track of customers' purchases, so there is a large database linking bar codes of products with customer information. The marketing department wants to find relationships between a customer's food and book purchases. For instance, a relatively high proportion of people who buy pizzas might also buy crime novels. With this knowledge, the business can specifically target customers who make specific food purchases with information about new novels when they are published.

In this example, each sales database can be encapsulated as a distributed component with an interface that provides read-only access to its data. Integrator components are each concerned with specific types of relationships, and they collect information from all of the databases to try to deduce the relationships. There might be an integrator component that is concerned with seasonal variations in goods sold, and another integrator that is concerned with relationships between different types of goods.

Visualizer components interact with integrator components to create a visualization or a report on the relationships that have been discovered. Because of the large volumes of data that are handled, visualizer components normally present their results graphically. Finally, a display component may be responsible for delivering the graphical models to clients for final presentation in their web browser.

A distributed component architecture rather than a layered architecture is appropriate for this type of application because you can add new databases to the system without major disruption. Each new database is simply accessed by adding another distributed component. The database access components provide a simplified interface that controls access to the data. The databases that are accessed may reside on different machines. The architecture also makes it easy to mine new types of relationships by adding new integrator objects.

Distributed component architectures suffer from two major disadvantages:

1. They are more complex to design than client–server systems. Multilayer client–server systems appear to be a fairly intuitive way to think about systems. They reflect many human transactions where people request and receive services from other people who specialize in providing these services. The complexity of distributed component architectures increases the costs of implementation.

2. There are no universal standards for distributed component models or middleware. Rather, different vendors, such as Microsoft and Sun, developed different, incompatible middleware. This middleware is complex, and reliance on it significantly increases the complexity of distributed component systems.

As a result of these problems, distributed component architectures are being replaced by service-oriented systems (discussed in Chapter 18). However, distributed component systems have performance benefits over service-oriented systems. RPC communications are usually faster than the message-based interaction used in service-oriented systems. Distributed component architectures are therefore still used for high-throughput systems in which large numbers of transactions have to be processed quickly.

17.3.5 Peer-to-peer architectures

The client–server model of computing that I have discussed in previous sections of the chapter makes a clear distinction between servers, which are providers of services, and clients, which are receivers of services. This model usually leads to an uneven distribution of load on the system, where servers do more work than clients. This may lead to organizations spending a lot on server capacity while there is unused processing capacity on the hundreds or thousands of PCs and mobile devices used to access the system servers.

Peer-to-peer (p2p) systems (Oram 2001) are decentralized systems in which computations may be carried out by any node on the network. In principle at least, no distinctions are made between clients and servers. In peer-to-peer applications, the overall system is designed to take advantage of the computational power and storage available across a potentially huge network of computers. The standards and protocols that enable communications across the nodes are embedded in the application itself, and each node must run a copy of that application.

Peer-to-peer technologies have mostly been used for personal rather than business systems. The fact that there are no central servers means that these systems are harder to monitor; therefore, a higher level of communication privacy is possible.

For example, file-sharing systems based on the BitTorrent protocol are widely used to exchange files on users' PCs. Private instant messaging systems, such as ICQ and Jabber, provide direct communications between users without an intermediate server. Bitcoin is a peer-to-peer payments system using the Bitcoin electronic currency. Freenet is a decentralized database that has been designed to make it easier to publish information anonymously and to make it difficult for authorities to suppress this information.

Other p2p systems have been developed where privacy is not the principal requirement. Voice over IP (VoIP) phone services, such as Viber, rely on peer-to-peer communication between the parties involved in the phone call or conference. SETI@home is a long-running project that processes data from radio telescopes on home PCs in order to search for indications of extraterrestrial life. In these systems, the advantage of the p2p model is that a central server is not a processing bottleneck.

Peer-to-peer systems have also been used by businesses to harness the power in their PC networks (McDougall 2000). Intel and Boeing have both implemented p2p systems for computationally intensive applications. Such systems take advantage of unused processing capacity on local computers. Instead of buying expensive high-performance hardware, engineering computations can be run overnight when desktop computers are unused. Businesses also make extensive use of commercial p2p systems, such as messaging and VoIP systems.

In principle, every node in a p2p network could be aware of every other node. Nodes could connect to and exchange data directly with any other node in the network. In practice, this is impossible unless the network has only a few members. Consequently, nodes are usually organized into "localities," with some nodes acting as bridges to other node localities. Figure 17.14 shows this decentralized p2p architecture.

In a decentralized architecture, the nodes in the network are not simply functional elements but are also communications switches that can route data and control signals from one node to another. For example, assume that Figure 17.14 represents a decentralized, document-management system. A consortium of researchers uses this system to share documents. Each member of the consortium maintains his or her own document store. However, when a document is retrieved, the node retrieving that document also makes it available to other nodes.

If someone needs a document that is stored somewhere on the network, they issue a search command, which is sent to nodes in their "locality." These nodes check whether they have the document and, if so, return it to the requestor. If they do not have it, they route the search to other nodes. Therefore if n1 issues a search for a

Figure 17.14
A decentralized p2p architecture

document that is stored at n10, this search is routed through nodes n3, n6, and n9 to n10. When the document is finally discovered, the node holding the document then sends it to the requesting node directly by making a peer-to-peer connection.

This decentralized architecture has the advantage of being highly redundant and hence both fault-tolerant and tolerant of nodes disconnecting from the network. However, the disadvantages are that many different nodes may process the same search, and there is also significant overhead in replicated peer communications.

An alternative p2p architectural model, which departs from a pure p2p architecture, is a semicentralized architecture where, within the network, one or more nodes act as servers to facilitate node communications. This reduces the amount of traffic between nodes. Figure 17.15 illustrates how this semicentralized architectural model differs from the completely decentralized model shown in Figure 17.14.

In a semicentralized architecture, the role of the server (sometimes called a superpeer) is to help establish contact between peers in the network or to coordinate the results of a computation. For example, if Figure 17.15 represents an instant messaging system, then network nodes communicate with the server (indicated by dashed lines) to find out what other nodes are available. Once these nodes are discovered, direct communications can be established and the connection to the server becomes unnecessary. Therefore, nodes n2, n3, n5, and n6 are in direct communication.

In a computational p2p system, where a processor-intensive computation is distributed across a large number of nodes, it is normal for some nodes to be superpeers. Their role is to distribute work to other nodes and to collate and check the results of the computation.

The peer-to-peer architectural model may be the best model for a distributed system in two circumstances:

1. Where the system is computationally-intensive and it is possible to separate the processing required into a large number of independent computations. For example, a peer-to-peer system that supports computational drug discovery distributes computations that look for potential cancer treatments by analyzing a huge number of molecules to see if they have the characteristics required to suppress the growth of cancers. Each molecule can be considered separately, so there is no need for the peers in the system to communicate.

Figure 17.15 A semicentralized p2p architecture

2. Where the system primarily involves the exchange of information between individual computers on a network and there is no need for this information to be centrally stored or managed. Examples of such applications include file-sharing systems that allow peers to exchange local files such as music and video files, and phone systems that support voice and video communications between computers.

Peer-to-peer architectures allow for the efficient use of capacity across a network. However, security concerns are the principal reason why these systems have not become more widely used, especially in business (Wallach 2003). The lack of centralized management means that attackers can set up malicious nodes that deliver spam and malware to legitimate p2p system users. Peer-to-peer communications involve opening your computer to direct interactions with other peers and this means that these systems could potentially access any of your resources. To counter this possibility, you need to organize your system so that these resources are protected. If this is done incorrectly, then your system is insecure and vulnerable to external corruption.

17.4 Software as a service

In the previous sections, I discussed client–server models and how functionality may be distributed between the client and the server. To implement a client–server system, you may have to install a program or an app on the client computer, which communicates with the server, implements client-side functionality, and manages the user interface. For example, a mail client, such as Outlook or Mac Mail, provides mail management features on your own computer. This avoids the problem of server overload in thin-client systems, where all of the processing is carried out at the server.

The problems of server overload can be significantly reduced by using web technologies such as AJAX (Holdener, 2008) and HTML5 (Sarris 2013). These technologies support efficient management of web page presentation and local computation by executing scripts that are part of the web page. This means that a browser can be configured and used as client, with significant local processing. The application software can be

thought of as a remote service, which can be accessed from any device that can run a standard browser. Widely used examples of SaaS include web-based mail systems, such as Yahoo and Gmail, and office applications, such as Google Docs and Office 365.

This idea of software as a service (SaaS) involves hosting the software remotely and providing access to it over the Internet. The key elements of SaaS are as follows:

1. Software is deployed on a server (or more commonly in the cloud) and is accessed through a web browser. It is not deployed on a local PC.
2. The software is owned and managed by a software provider rather than the organizations using the software.
3. Users may pay for the software according to how much use they make of it or through an annual or monthly subscription. Sometimes the software is free for anyone to use, but users must then agree to accept advertisements, which fund the software service.

The development of SaaS has accelerated over the past few years as cloud computing has become widely used. When a service is deployed in the cloud, the number of servers can quickly change to match the user demands for that service. There is no need for service providers to provision for peak loads; as a result, the costs for these providers have been dramatically reduced.

For software purchasers, the benefit of SaaS is that the costs of management of software are transferred to the provider. The provider is responsible for fixing bugs and installing software upgrades, dealing with changes to the operating system platform, and ensuring that hardware capacity can meet demand. Software license management costs are zero. If someone has several computers, there is no need to license software for all of these. If a software application is only used occasionally, the pay-per-use model may be cheaper than buying an application. The software may be accessed from mobile devices, such as smartphones, from anywhere in the world.

The main problem that inhibits the use of SaaS is data transfer with the remote service. Data transfer takes place at network speeds, and so transferring a large amount of data, such as video or high-quality images takes a lot of time. You may also have to pay the service provider according to the amount transferred. Other problems are lack of control over software evolution (the provider may change the software when it wishes) and problems with laws and regulations. Many countries have laws governing the storage, management, preservation, and accessibility of data, and moving data to a remote service may breach these laws.

The notion of software as a service and service-oriented architectures (SOA), discussed in Chapter 18, are related, but they are not the same:

1. Software as a service is a way of providing functionality on a remote server with client access through a web browser. The server maintains the user's data and state during an interaction session. Transactions are usually long transactions, for example, editing a document.

2. Service-oriented architecture is an approach to structuring a software system as a set of separate, stateless services. These services may be provided by multiple providers and may be distributed. Typically, transactions are short transactions where a service is called, does something, and then returns a result.

SaaS is a way of delivering application functionality to users, whereas SOA is an implementation technology for application systems. Systems that are implemented using SOA do not have to be accessed by users as web services. SaaS applications for business may be implemented using components rather than services. However, if SaaS is implemented using SOA, it becomes possible for applications to use service APIs to access the functionality of other applications. They can then be integrated into more complex systems. These systems are called mashups and are another approach to software reuse and rapid software development.

From a software development perspective, the process of service development has much in common with other types of software development. However, service construction is not usually driven by user requirements, but by the service provider's assumptions about what users need. Accordingly, the software needs to be able to evolve quickly after the provider gets feedback from users on their requirements. Agile development with incremental delivery is therefore an effective approach for software that is to be deployed as a service.

Some software that is implemented as a service, such as Google Docs for web users, offers a generic experience to all users. However, businesses may wish to have specific services that are tailored to their own requirements. If you are implementing SaaS for business, you may base your software service on a generic service that is tailored to the needs of each business customer. Three important factors have to be considered:

1. *Configurability* How do you configure the software for the specific requirements of each organization?

2. *Multi-tenancy* How do you present each user of the software with the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources?

3. *Scalability* How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?

The notion of product-line architectures, discussed in Chapter 16, is one way of configuring software for users who have overlapping but not identical requirements. You start with a generic system and adapt it according to the specific requirements of each user.

This does not work for SaaS, however, for it would mean deploying a different copy of the service for each organization that uses the software. Rather, you need to design configurability into the system and provide a configuration interface that allows users to specify their preferences. You then use these preferences to adjust the behavior of the software dynamically as it is used. Configuration facilities may allow for:

1. *Branding*, where users from each organization are presented with an interface that reflects their own organization.

Figure 17.16 Configuration of a software system offered as a service

2. *Business rules and workflows*, where each organization defines its own rules that govern the use of the service and its data.
3. *Database extensions*, where each organization defines how the generic service data model is extended to meet its specific needs.
4. *Access control*, where service customers create individual accounts for their staff and define the resources and functions that are accessible to each of their users.

Figure 17.16 illustrates this situation. This diagram shows five users of the application service, who work for three different customers of the service provider. Users interact with the service through a customer profile that defines the service configuration for their employer.

Multi-tenancy is a situation in which many different users access the same system and the system architecture is defined to allow the efficient sharing of system resources. However, it must appear to users that they each have sole use of the system. Multi-tenancy involves designing the system so that there is an absolute separation between system functionality and system data. All operations must therefore be stateless so that they can be shared. Data must either be provided by the client or should be available in a storage system or database that can be accessed from any system instance.

A particular problem in multi-tenant systems is data management. The simplest way to provide data management is for all customers to have their own database, which they may use and configure as they wish. However, this requires the service provider to maintain many different database instances (one per customer) and to make these databases available on demand.

As an alternative, the service provider can use a single database, with different users being virtually isolated within that database. This is illustrated in Figure 17.17, where you can see that database entries also have a "tenant identifier" that links these entries to specific users. By using database views, you can extract the entries for each service customer and so present users from that customer with a virtual, personal database. This process can be extended to meet specific customer needs using the configuration features discussed above.

Scalability is the ability of the system to cope with increasing numbers of users without reducing the overall quality of service that is delivered to any user. Generally,

Tenant	Key	Name	Address
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	BigCorp	2, Main St, Motown
435	X234	J. Bowie	56, Mill St, Starville
592	PP37	R. Burns	Alloway, Ayrshire

Figure 17.17 A multi-tenant database

when considering scalability in the context of SaaS, you are considering "scaling out" rather than "scaling up." Recall that scaling out means adding additional servers and so also increasing the number of transactions that can be processed in parallel. Scalability is a complex topic that I cannot cover in detail here, but following are some general guidelines for implementing scalable software:

1. Develop applications where each component is implemented as a simple stateless service that may be run on any server. In the course of a single transaction, a user may therefore interact with instances of the same service that are running on several different servers.
2. Design the system using asynchronous interaction so that the application does not have to wait for the result of an interaction (such as a read request). This allows the application to carry on doing useful work while it is waiting for the interaction to finish.
3. Manage resources, such as network and database connections, as a pool so that no single server is likely to run out of resources.
4. Design your database to allow fine-grain locking. That is, do not lock out whole records in the database when only part of a record is in use.
5. Use a cloud PaaS platform, such as Google App Engine (Sanderson 2012) or other PaaS platform for system implementation. These include mechanisms that will automatically scale out your system as the load increases.

The notion of software as a service is a major paradigm shift for distributed computing. We have already seen consumer software and professional applications, such as Photoshop, move to this model of delivery. Increasingly, businesses are replacing their own systems, such as CRM and inventory systems, with cloud-based SaaS systems from external providers such as Salesforce. Specialized software companies that implement business applications prefer to provide SaaS because it simplifies software update and management.

SaaS represents a new way to think about the engineering of enterprise systems. It has always been helpful to think of systems delivering services to users, but, until SaaS, this function has involved using different abstractions, such as objects, when implementing the system. Where there is a closer match between user and system abstractions, the resultant systems are easier to understand, maintain, and evolve.

KEY POINTS

- The benefits of distributed systems are that they can be scaled to cope with increasing demand, can continue to provide user services (even if some parts of the system fail), and they enable resources to be shared.
- Issues to be considered in the design of distributed systems include transparency, openness, scalability, security, quality of service, and failure management.
- Client–server systems are distributed systems in which the system is structured into layers, with the presentation layer implemented on a client computer. Servers provide data management, application, and database services.
- Client–server systems may have several tiers, with different layers of the system distributed to different computers.
- Architectural patterns for distributed systems include master–slave architectures, two-tier and multi-tier client–server architectures, distributed component architectures, and peer-to-peer architectures.
- Distributed component systems require middleware to handle component communications and to allow objects to be added to and removed from the system.
- Peer-to-peer architectures are decentralized architectures in which there are no distinguished clients and servers. Computations can be distributed over many systems in different organizations.
- Software as a service is a way of deploying applications as thin client–server systems, where the client is a web browser.

FURTHER READING

Peer-to-Peer: Harnessing the Power of Disruptive Technologies. Although this book does not have a lot of information on p2p architectures, it is an excellent introduction to p2p computing and discusses the organization and approach used in a number of p2p systems. (A. Oram (ed.), O'Reilly and Associates Inc., 2001).

"Turning Software into a Service." A good overview paper that discusses the principles of service-oriented computing. Unlike many papers on this topic, it does not conceal these principles behind a discussion of the standards involved. (M. Turner, D. Budgen, and P. Brereton, *IEEE Computer*, 36 (10), October 2003) http://dx.doi.org/10.1109/MC.2003.1236470

Distributed Systems, 5th ed. A comprehensive textbook that discusses all aspects of distributed systems design and implementation. It includes coverage of peer-to-peer systems and mobile systems. (G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. Addison-Wesley, 2011).

Engineering Software as a Service: An Agile Approach Using Cloud Computing. This book accompanies the authors' online course on the topic. A good practical book that is aimed at people new to this type of development. (A. Fox and D. Patterson, Strawberry Canyon LLC, 2014) http://www.saasbook.info

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/requirements-and-design/

EXERCISES

17.1. What do you understand by "scalability"? Discuss the differences between scaling up and scaling out and explain when these different approaches to scalability may be used.

17.2. Explain why distributed software systems are more complex than centralized software systems, where all of the system functionality is implemented on a single computer.

17.3. Using an example of a remote procedure call, explain how middleware coordinates the interaction of computers in a distributed system.

17.4. What are the different logical layers in an application with a distributed client–server architecture?

17.5. You have been asked to design a secure system that requires strong authentication and authorization. The system must be designed so that communications between parts of the system cannot be intercepted and read by an attacker. Suggest the most appropriate client–server architecture for this system and, giving the reasons for your answer, propose how functionality should be distributed between the client and the server systems.

17.6. Your customer wants to develop a system for stock information where dealers can access information about companies and evaluate various investment scenarios using a simulation system. Each dealer uses this simulation in a different way, according to his or her experience and the type of stocks in question. Suggest a client–server architecture for this system that shows where functionality is located. Justify the client–server system model that you have chosen.

17.7. Using a distributed component approach, propose an architecture for a national theater booking system. Users can check seat availability and book seats at a group of theaters. The system should support ticket returns so that people may return their tickets for last-minute resale to other customers.

17.8. What is the fundamental problem with a two-tier client–server approach? Define how a multi-tier client–server approach overcomes this.

17.9. List the benefits that a distributed component model has when used for implementing distributed systems.

17.10. Your company wishes to move from using desktop applications to accessing the same functionality remotely as services. Identify three risks that might arise and suggest how these risks may be reduced.

REFERENCES

Bernstein, P. A. 1996. "Middleware: A Model for Distributed System Services." *Comm. ACM* 39 (2): 86–97. doi:10.1145/230798.230809.

Coulouris, G., J. Dollimore, T. Kindberg, and G. Blair. 2011. *Distributed Systems: Concepts and Design, 5th ed.* Harlow, UK: Addison-Wesley.

Holdener, A. T. (2008). *Ajax: The Definitive Guide. Sebastopol, CA.: O' Reilly & Associates*.

McDougall, P. 2000. "The Power of Peer-to-Peer." *Information Week* (August 28, 2000). http://www.informationweek.com/801/peer.htm

Oram, A. 2001. "Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology." Sebastopol, CA: O'Reilly & Associates.

Orfali, R., D. Harkey, and J. Edwards. 1997. *Instant CORBA*. Chichester, UK: John Wiley & Sons.

Pope, A. 1997. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Harlow, UK: Addison-Wesley.

Sanderson, D. 2012. *Programming with Google App Engine*. Sebastopol, CA: O'Reilly Media Inc.

Sarris, S. 2013. *HTML5 Unleashed*. Indianapolis, IN: Sams Publishing.

Tanenbaum, A. S., and M. Van Steen. 2007. *Distributed Systems: Principles and Paradigms, 2nd ed.* Upper Saddle River, NJ: Prentice-Hall.

Wallach, D. S. 2003. "A Survey of Peer-to-Peer Security Issues." In *Software Security: Theories and Systems*, edited by M. Okada, B. C. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, 42–57. Heidelberg: Springer-Verlag. doi:10.1007/3-540-36532-X_4.

18

Service-oriented software engineering

Objectives

The objective of this chapter is to introduce service-oriented software engineering as a way of building distributed applications using web services. When you have read this chapter, you will:

- understand the basic notions of a web service, web service standards, and service-oriented architecture;
- understand the idea of RESTful services and the important differences between RESTful and SOAP-based services;
- understand the service engineering process that is intended to produce reusable web services;
- understand how workflow-based service composition can be used to create service-oriented software that supports business processes.

Contents

18.1 Service-oriented architecture
18.2 RESTful services
18.3 Service engineering
18.4 Service composition

The development of the Web in the 1990s revolutionized organizational information exchange. Client computers could gain access to information on remote servers outside their own organizations. However, access was solely through a web browser, and direct access to the information by other programs was not practical. This meant that opportunistic connections between servers, where, for example, a program could query a number of catalogs from different suppliers, were not possible.

To get around this problem, web services were developed that allowed programs to access and update resources available on the web. Using a web service, organizations that wish to make their information accessible to other programs can do so by defining and publishing a programmatic web service interface. This interface defines the data available and how it can be accessed and used.

More generally, a web service is a standard representation for some computational or information resource that can be used by other programs. These may be information resources, such as a parts catalog, computer resources, such as a specialized processor, or storage resources. For example, an archive service could be implemented that permanently and reliably stores organizational data that, by law, has to be maintained for many years.

A web service is an instance of a more general notion of a service, which Lovelock et al. (Lovelock et al., 1996) defined as:

> *an act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production.*[†]

Services are a natural development of software components where the component model is, in essence, a set of standards associated with web services. A web service can therefore be defined as:

> *A loosely coupled, reusable software component that encapsulates discrete functionality, which may be distributed and programmatically accessed. A web service is a service that is accessed using standard Internet and XML-based protocols.*

A critical distinction between a service and a software component, as defined in component-based software engineering, is that services should be independent and loosely coupled. That is, they should always operate in the same way, irrespective of their execution environment. They should not rely on external components that may have different functional and non-functional behavior. Therefore, web services do not have a "requires" interface that, in CBSE, defines the other system components that must be present. A web service interface is simply a "provides" interface that defines the service functionality and parameters.

Service-oriented systems are a way of developing distributed systems where the system components are stand-alone services, executing on geographically distributed computers. Services are platform and implementation-language independent. Software systems can be constructed by composing local services and external services from different providers, with seamless interaction between the services in the system.

[†]Lovelock, C., Vandermerwe, S. and Lewis, B. (1996). Services Marketing. Englewood Cliffs, NJ: Prentice Hall.

As I discussed in Chapter 17, the ideas of "software as a service" and "service-oriented systems" are not the same thing. Software as a service means offering software functionality to users remotely over the web, rather than through applications installed on a user's computer. Service-oriented systems are systems that are implemented using reusable service components and that are accessed by other programs, rather than directly by users. Software that is offered as a service may be implemented using a service-oriented system. However, you don't have to implement software in this way to offer it as a user service.

Adopting a service-oriented approach to software engineering has a number of important benefits:

1. Services can be offered by any service provider inside or outside of an organization. Assuming these services conform to certain standards, organizations can create applications by integrating services from a range of providers. For example, a manufacturing company can link directly to services provided by its suppliers.
2. The service provider makes information about the service public so that any authorized user can use the service. The service provider and the service user do not need to negotiate about what the service does before it can be incorporated in an application program.
3. Applications can delay the binding of services until they are deployed or until execution. Therefore, an application using a stock price service (say) could, in principle, dynamically change service providers while the system was executing. This means that applications can be reactive and adapt their operation to cope with changes to their execution environment.
4. Opportunistic construction of new services is possible. A service provider may recognize new services that can be created by linking existing services in innovative ways.
5. Service users can pay for services according to their use rather than their provision. Therefore, instead of buying an expensive component that is rarely used, the application writer can use an external service that will be paid for only when required.
6. Applications can be made smaller, which is particularly important for mobile devices with limited processing and memory capabilities. Some computationally intensive processing and exception handling can be offloaded to external services.

Service-oriented systems have loosely coupled architectures where service bindings may change during system execution. A different, but equivalent, version of the service may therefore be executed at different times. Some systems will be solely built using web services, and others will mix web services with locally developed components. To illustrate how applications that use a mixture of services and components may be organized, consider the following scenario:

An in-car information system provides drivers with information on weather, road traffic conditions, local information and so forth. This is linked to the car

Figure 18.1 A service-based, in-car information system

radio so that information is delivered as a signal on a specific radio channel. The car is equipped with GPS receiver to discover its position, and, based on that position, the system accesses a range of information services. Information may then be delivered in the driver's specified language.

Figure 18.1 illustrates a possible organization for such a system. The in-car software includes five modules. These handle communications with the driver, with a GPS receiver that reports the car's position, and with the car radio. The Transmitter and Receiver modules handle all communications with external services.

The car communicates with an external mobile information service that aggregates information from a range of other services, providing information on weather, traffic, and local facilities. Different providers in different places offer these services, and the in-car system accesses an external discovery service to find the services available in the local area. The mobile information service also uses the discovery service to bind to the appropriate weather, traffic, and facilities services. The aggregated information is then sent to the car through a service that translates that information into the driver's preferred language.

This example illustrates one of the key advantages of the service-oriented approach. When the system is programmed or deployed, you don't have to decide what service

provider should be used or what specific services should be accessed. As the car moves around, the in-car software uses the service discovery service to find the most useful local information service. Because of the use of a translation service, it can move across borders and make local information available to people who don't speak the local language.

I think that the service-oriented approach to software engineering is as important a development as object-oriented software engineering. Service-oriented systems are essential to the cloud and mobile systems. Newcomer and Lomow (Newcomer and Lomow 2005), in their book on SOA, summarize the potential of service-oriented approaches, which is now being realized:

> *Driven by the convergence of key technologies and the universal adoption of Web services, the service-oriented enterprise promises to significantly improve corporate agility, speed time-to-market for new products and services, reduce IT costs and improve operational efficiency.*[†]

Building applications based on services allows companies and other organizations to cooperate and make use of each other's business functions. Thus, systems that involve extensive information exchange across company boundaries, such as supply chain systems where one company orders goods from another, can easily be automated. Service-based applications may be constructed by linking services from various providers using either a standard programming language or a specialized workflow language, as discussed in Section 18.4.

Initial work on service provision and implementation was heavily influenced by the failure of the software industry to agree on component standards. It was therefore standards-driven, with all of the main industrial companies involved in standards development. This led to a whole range of standards (WS* standards) and the notion of service-oriented architectures. These were proposed as architectures for service-based systems, with all service communication being standards-based. However, the standards proposed were complex and had a significant execution overhead. This problem has led many companies to adopt an alternative architectural approach, based on so-called RESTful services. A RESTful approach is a simpler approach than a service-oriented architecture, but it is less suited to services that offer complex functionality. I discuss both of these architectural approaches in this chapter.

18.1 Service-oriented architecture

Service-oriented architecture (SOA) is an architectural style based on the idea that executable services can be included in applications. Services have well-defined, published interfaces, and applications can choose whether or not these are appropriate. An important idea underlying SOA is that the same service may be available from different providers and that applications could make a runtime decision of which service provider to use.

[†]Newcomer, E. and Lomow, G. (2005). Understanding SOA with Web Services. Boston: Addison-Wesley.

Figure 18.2 Service-oriented architecture

Figure 18.3 Web service standards

Figure 18.2 illustrates the structure of a service-oriented architecture. Service providers design and implement services and specify the interface to these services. They also publish information about these services in an accessible registry. Service requestors (sometimes called service clients) who wish to make use of a service discover the specification of that service and locate the service provider. They can then bind their application to that specific service and communicate with it, using standard service protocols.

The development and use of internationally agreed standards is fundamental to SOA. As a result, service-oriented architectures have not suffered from the incompatibilities that normally arise with technical innovations, where different suppliers maintain their proprietary version of the technology. Figure 18.3 shows the stack of key standards that have been established to support web services.

Web service protocols cover all aspects of service-oriented architectures, from the basic mechanisms for service information exchange (SOAP) to programming language standards (WS-BPEL). These standards are all based on XML, a human and machine-readable notation that allows the definition of structured data where text is tagged with a meaningful identifier. XML has a range of supporting technologies, such as XSD for schema definition, which are used to extend and manipulate XML descriptions. Erl (Erl 2004) provides a good summary of XML technologies and their role in web services.

Briefly, the fundamental standards for service-oriented architectures are:

1. *SOAP* This is a message interchange standard that supports communication between services. It defines the essential and optional components of messages

passed between services. Services in a service-oriented architecture are sometimes called SOAP-based services.

2. *WSDL* The Web Service Description Language (WSDL) is a standard for service interface definition. It sets out how the service operations (operation names, parameters, and their types) and service bindings should be defined.

3. *WS-BPEL* This is a standard for a workflow language that is used to define process programs involving several different services. I explain what process programs are in Section 18.3.

The UDDI (Universal Description, Discovery, and Integration) discovery standard defines the components of a service specification intended to help potential users discover the existence of a service. This standard was meant to allow companies to set up registries, with UDDI descriptions defining the services they offered. Some companies set up UDDI registries in the early years of the 21st century, but users preferred standard search engines to find services. All public UDDI registries have now closed.

The principal SOA standards are supported by a range of supporting standards that focus on more specialized aspects of SOA. There are many supporting standards because they are intended to support SOA in different types of enterprise application. Some examples of these standards include:

1. *WS-Reliable Messaging,* a standard for message exchange that ensures messages will be delivered once and once only.

2. *WS-Security,* a set of standards supporting web service security, including standards that specify the definition of security policies and standards that cover the use of digital signatures.

3. *WS-Addressing,* which defines how address information should be represented in a SOAP message.

4. *WS-Transactions,* which defines how transactions across distributed services should be coordinated.

Web service standards are a huge topic, and I don't have space to discuss them in detail here. I recommend Erl's books (Erl 2004, 2005) for an overview of these standards. Their detailed descriptions are also available as public documents on the Web (W3C 2013).

18.1.1 Service components in an SOA

Message exchange, as I explained in Section 17.1, is an important mechanism for coordinating actions in a distributed computing system. Services in a SOA communicate by exchanging messages, expressed in XML, and these messages are distributed using standard Internet transport protocols such as HTTP and TCP/IP.

A service defines what it needs from another service by setting out its requirements in a message, which is sent to that service. The receiving service parses the

Figure 18.4
Organization of a WSDL specification

message, carries out the computation, and, upon completion, sends a reply, as a message, to the requesting service. This service then parses the reply to extract the required information. Unlike software components, services do not use remote procedure or method calls to access functionality associated with other services.

When you intend to use a web service, you need to know where the service is located (its Uniform Resource Identifier—URI) and the details of its interface. These details are provided in a service description that is written in an XML-based language called WSDL (Web Service Description Language). The WSDL specification defines three aspects of a Web service: what the service does, how it communicates, and where to find it:

1. The "what" part of a WSDL document, called an interface, specifies what operations the service supports and defines the format of the messages sent and received by the service.
2. The "how" part of a WSDL document, called a binding, maps the abstract interface to a concrete set of protocols. The binding specifies the technical details of how to communicate with a Web service.
3. The "where" part of a WSDL document describes the location of a specific Web service implementation (its endpoint).

The WSDL conceptual model (Figure 18.4) shows the elements of a service description. Each element is expressed in XML and may be provided in separate files. These elements are:

1. An introductory part that usually defines the XML namespaces used and that may include a documentation section providing additional information about the service.
2. An optional description of the types used in the messages exchanged by the service.
3. A description of the service interface, that is, the operations that the service provides for other services or users.
4. A description of the input and output messages processed by the service.
5. A description of the binding used by the service, that is, the messaging protocol that will be used to send and receive messages. The default is SOAP, but other

```
Define some of the types used. Assume that the namespace prefixes 'ws' refers to
the namespace URI for XML schemas and the namespace prefix associated with this
definition is weathns.
<types>
   <xs: schema targetNameSpace = "http://.../weathns"
       xmlns: weathns = "http://.../weathns" >
       <xs:element name = "PlaceAndDate" type = "pdrec" />
       <xs:element name = "MaxMinTemp" type = "mmtrec" />
       <xs:element name = "InDataFault" type = "errmess" />

       <xs:complexType name = "pdrec"
       <xs:sequence>
            <xs:element name = "town" type = "xs:string"/>
            <xs:element name = "country" type = "xs:string"/>
            <xs:element name = "day" type = "xs:date" />
       </xs:complexType>

       Definitions of MaxMinType and InDataFault here

   </schema>

</types>

Now define the interface and its operations. In this case, there is only a single
operation to return maximum and minimum temperatures

<interface name = "weatherInfo" >
   <operation name = "getMaxMinTemps" pattern = "wsdlns: in-out">
   <input messageLabel = "In" element = "weathns: PlaceAndDate" />
   <output messageLabel = "Out" element = "weathns:MaxMinTemp" />
   <outfault messageLabel = "Out" element = "weathns:InDataFault" />
</operation>
</interface>
```

Figure 18.5 Part of a WSDL description for a web service

bindings may also be specified. The binding sets out how the input and output messages associated with the service should be packaged into a message, and specifies the communication protocols used. The binding may also specify how supporting information, such as security credentials or transaction identifiers, is included in messages to the service.

6. An endpoint specification that is the physical location of the service, expressed as a URI—the address of a resource that can be accessed over the Internet.

Figure 18.5 shows part of the interface for a simple service that, given a date and a place, specified as a town within a country, returns the maximum and minimum temperature recorded in that place on that date. The input message also specifies whether these temperatures are to be returned in degrees Celsius or degrees Fahrenheit.

XML-based service descriptions include definitions of XML namespaces. A namespace identifier may precede any identifier used in the XML description, making it possible to distinguish between identifiers with the same name that have been defined in different parts of an XML description. You don't have to understand the details of namespaces to

understand the examples here. You only need to know that names may be prefixed with a namespace identifier and that the namespace:name pair should be unique.

In Figure 18.5, the first part of the description shows part of the element and type definition that is used in the service specification. This defines the elements PlaceAndDate, MaxMinTemp, and InDataFault. I have only included the specification of PlaceAndDate, which you can think of as a record with three fields—town, country and date. A similar approach would be used to define MaxMinTemp and InDataFault.

The second part of the description shows how the service interface is defined. In this example, the service weatherInfo has a single operation, although there are no restrictions on the number of operations that may be defined. The weatherInfo operation has an associated in-out pattern meaning that it takes one input message and generates one output message. The WSDL 2.0 specification allows for a number of message exchange patterns such as in-only, in-out, out-only, in-optional-out, and out-in. The input and output messages, which refer to the definitions made earlier in the types section, are then defined.

A service interface that is defined in WSDL is simply a description of the service signature, that is, the operations and their parameters. It does not include any information about the semantics of the service or its non-functional characteristics, such as performance and dependability. If you plan to use the service, you have to work out what the service actually does and the meaning of the input and output messages. You have to experiment to discover the service's performance and dependability. While meaningful names and documentation help with understanding the service functionality, it is still possible to misunderstand what the service actually does.

XML-based service descriptions are long, detailed, and tedious to read. WSDL specifications are not normally written by hand, and most of the information in a specification is automatically generated.

18.2 RESTful services

The initial developments of web services and service-oriented software engineering were standards-based, with XML-based messages exchanged between services. This is a general approach that allows for the development of complex services, dynamic service binding, and control over quality of service and service dependability. However, as services were developed, it emerged that most of these were single-function services with relatively simple input and output interfaces. Service users were not really interested in dynamic binding and the use of multiple service providers. They rarely use web service standards for quality of service, reliability, and so forth.

The problem is that web services standards are "heavyweight" standards that are sometimes overly general and inefficient. Implementing these standards requires a considerable amount of processing to create, transmit, and interpret the associated XML messages. This slows down communications between services, and, for high-throughput systems, additional hardware may be required to deliver the quality of service required.

In response to this situation, an alternative "lightweight" approach to web service architecture has been developed. This approach is based on the REST architectural

Figure 18.6 Resources and actions

(a) General resource actions

(b) Web resources

style, where REST stands for Representational State Transfer (Fielding 2000). REST is an architectural style based on transferring representations of resources from a server to a client. It is the style that underlies the web as a whole and has been used as a much simpler method than SOAP/WSDL for implementing web service interfaces.

The fundamental element in a RESTful architecture is a resource. Essentially, a resource is simply a data element such as a catalog and a medical record, or a document, such as this book chapter. In general, resources may have multiple representations; that is, they can exist in different formats. For example, this book chapter has three representations. These are a MS Word representation, which is used for editing, a PDF representation, which is used for web display, and a InDesign representation, which is used for publishing. The underlying logical resource made up of text and images is the same in all of these representations.

In a RESTful architecture, everything is represented as a resource. Resources have a unique identifier, which is their URL. Resources are a bit like objects, with four fundamental polymorphic operations associated with them, as shown in Figure 18.6(a):

1. Create—bring the resource into existence.
2. Read—return a representation of the resource.
3. Update—change the value of the resource.
4. Delete—make the resource inaccessible.

The Web is an example of a system that has a RESTful architecture. Web pages are resources, and the unique identifier of a web page is its URL.

The web protocols http and https are based on four actions, namely, POST, GET, PUT, and DELETE. These map onto the basic resource operations, as I have shown in Figure 18.6(b):

1. POST is used to create a resource. It has associated data that defines the resource.
2. GET is used to read the value of a resource and return that to the requestor in the specified representation, such as XHTML, that can be rendered in a web browser.

3. PUT is used to update the value of a resource.
4. DELETE is used to delete the resource.

All services, in some way, operate on data. For example, the service described in Section 18.2 that returns the maximum and minimum temperatures for a location on a given data uses a weather information database. SOAP-based services execute actions on this database to return particular values from it. RESTful services (Richardson and Ruby 2007) access the data directly.

When a RESTful approach is used, the data is exposed and is accessed using its URL. RESTful services use http or https protocols, with the only allowed actions being POST, GET, PUT, and DELETE. Therefore, the weather data for each place in the database might be accessed using URLs such as:

http://weather-info-example.net/temperatures/boston
http://weather-info-example.net/temperatures/edinburgh

This would invoke the GET operation and return a list of maximum and minimum temperatures. To request the temperatures for a specific date, a URL query can be used:

http://weather-info-example.net/temperatures/edinburgh?date=20140226

URL queries can also be used to disambiguate the request, given that there may be several places in the world with the same name:

http://weather-info-example.net/temperatures/boston?date=20140226&country=USA&state="Mass"

An important difference between RESTful services and SOAP-based services is that RESTful services are not exclusively XML-based. So, when a resource is requested, created, or changed, the representation may be specified. This is important for RESTful services because representations such as JSON (Javascript Object Notation), as well as XML, may be used. These can be processed more efficiently than XML-based notations, thus reducing the overhead involved in a service call. Therefore, the above request for maximum and minimum temperatures for Boston may return the following information:

```
{
"place": "Boston",
"country  "USA",
"state": "Mass",
"date": "26 Feb 2014",
"units": "Fahrenheit",
"max temp": 41,
"min temp": 29
}
```

The response to a GET request in a RESTful service may include URLs. Therefore, if the response to a request is a set of resources, then the URL of each of

Figure 18.7 RESTful and SOAP-based APIs

these services may be included. The requesting service may then process the requests in its own way. Therefore, a request for weather information given a place name that is not unique may return the URLs of all of the places that match the request. For example:

http://weather-info-example.net/temperatures/edinburgh-scotland
http://weather-info-example.net/temperatures/edinburgh-australia
http://weather-info-example.net/temperatures/edinburgh-maryland

A fundamental design principle for RESTful services is that they should be stateless. That is, in an interaction session, the resource itself should not include any state information, such as the time of the last request. Instead, all necessary state information should be returned to the requestor. If state information is required in later requests, it should be returned to the server by the requestor.

RESTful services have become more widely used over the past few years because of the widespread use of mobile devices. These devices have limited processing capabilities, so the lower overhead of RESTful services allows better system performance. They are also easy to use with existing websites—implementing a RESTful API for a website is usually fairly straightforward.

However, there are problems with the RESTful approach:

1. When a service has a complex interface and is not a simple resource, it can be difficult to design a set of RESTful services to represent this interface.
2. There are no standards for RESTful interface description, so service users must rely on informal documentation to understand the interface.
3. When you use RESTful services, you have to implement your own infrastructure for monitoring and managing the quality of service and the service reliability. SOAP-based services have additional infrastructure support standards such as WS-Reliability and WS-Transactions.

Pautasso et al. (Pautasso, Zimmermann, and Leymann 2008) discuss when RESTful and SOAP-based should be used. However, it is often possible to provide both SOAP-based and RESTful interfaces to the same service or resource (Figure 18.7). This dual approach is now common for cloud services from providers such as Microsoft, Google, and Amazon. Service clients can then choose the service access method that is best suited to their applications.

Figure 18.8 The service engineering process

18.3 Service engineering

Service engineering is the process of developing services for reuse in service-oriented applications. It has much in common with component engineering. Service engineers have to ensure that the service represents a reusable abstraction that could be useful in different systems. They must design and develop generally useful functionality associated with that abstraction and ensure that the service is robust and reliable. They have to document the service so that it can be discovered and understood by potential users.

As shown in Figure 18.8, there are three logical stages in the service engineering process:

1. *Service candidate identification,* where you identify possible services that might be implemented and define the service requirements.
2. *Service design,* where you design the logical service interface and its implementation interfaces (SOAP-based and/or RESTful).
3. *Service implementation and deployment,* where you implement and test the service and make it available for use.

As I discussed in Chapter 16, the development of a reusable component may start with an existing component that has already been implemented and used in an application. The same is true for services—the starting point for this process will often be an existing service or a component that is to be converted to a service. In this situation, the design process involves generalizing the existing component so that application-specific features are removed. Implementation means adapting the component by adding service interfaces and implementing the required generalizations.

18.3.1 Service candidate identification

The basic idea of service-oriented computing is that services should support business processes. As every organization has a wide range of processes, many possible services may be implemented. Service candidate identification therefore involves

	Utility	Business	Coordination
Task	Currency converter Employee locator	Validate claim form Check credit rating	Process expense claim Pay external supplier
Entity	Document translator Web form to XML converter	Expenses form Student application form	

Figure 18.9 Service classification

understanding and analyzing the organization's business processes to decide which reusable services could be implemented to support these processes.

Erl (Erl 2005) suggests that there are three fundamental types of service:

1. *Utility services.* These services implement some general functionality that may be used by different business processes. An example of a utility service is a currency conversion service that can be accessed to compute the conversion of one currency (e.g., dollars) to another (e.g., euros).

2. *Business services.* These services are associated with a specific business function. An example of a business function in a university would be the registration of students for a course.

3. *Coordination or process services.* These services support a more general business process that usually involves different actors and activities. An example of a coordination service in a company is an ordering service that allows orders to be placed with suppliers, goods accepted, and payments made.

Erl also suggests that services can be thought of as task-oriented or entity-oriented. Task-oriented services are associated with some activity, whereas entity-oriented services are associated with a system resource. The resource is a business entity such as a job application form. Figure 18.9 shows examples of services that are task-oriented or entity-oriented. Utility or business services may be entity-oriented or task-oriented. Coordination services are always task-oriented.

Your goal in service candidate identification should be to identify services that are logically coherent, independent, and reusable. Erl's classification is helpful in this respect, as it suggests how to discover reusable services by looking at business entities as resources and business activities. However, identifying service candidates is sometimes difficult because you have to envisage how the services could be used. You have to think of possible candidates and then ask a series of questions about them to see if they are likely to be useful services. Possible questions that you might ask to identify potentially reusable services are:

1. For an entity-oriented service, is the service associated with a single logical resource that is used in different business processes? What operations are normally performed on that entity that must be supported? Do these fit with the RESTful service operations PUT, CREATE, POST, and DELETE?

2. For a task-oriented service, is the task one that is carried out by different people in the organization? Will they be willing to accept the inevitable standardization

that occurs when a single support service is provided? Can this fit into the RESTful model, or should it be redesigned as an entity-oriented service.

3. Is the service independent? That is, to what extent does it rely on the availability of other services?

4. Does the service have to maintain state? If state information is required, this must either be maintained in a database or passed as a parameter to the service. Using a database affects service reusability as there is a dependency between the service and the required database. In general, services where the state is passed to the service are easier to reuse, as no database binding is required.

5. Might this service be used by external clients? For example, an entity-oriented service associated with a catalog could be made available to both internal and external users.

6. Are different users of the service likely to have different non-functional requirements? If they do, then more than one version of a service should perhaps be implemented.

The answers to these questions help you select and refine abstractions that can be implemented as services. However, there is no formulaic way of deciding which are the best services. You need to use your experience and business knowledge to decide on what are the most appropriate services.

The output of the service selection process is a set of identified services and associated requirements for these services. The functional service requirements should define what the service should do. The non-functional requirements should define the security, performance, and availability requirements of the service.

To help you understand the process of service candidate identification and implementation, consider the following example:

A company, which sells computer equipment, has arranged special prices for approved configurations for some large customers. To facilitate automated ordering, the company wishes to produce a catalog service that will allow customers to select the equipment that they need. Unlike a consumer catalog, orders are not placed directly through a catalog interface. Instead, goods are ordered through the web-based procurement system of each company that accesses the catalog as a web service. The reason for this is that large companies usually have their own budgeting and approval procedures for orders that must be followed when an order is placed.

The catalog service is an example of an entity-oriented service, where the underlying resource is the catalog. The functional catalog service requirements are as follows:

1. A specific version of the catalog shall be provided for each user company. This shall include the approved configurations and equipment that may be ordered by

employees of the customer company and the equipment prices that have been agreed to with that company.

2. The catalog shall allow a customer employee to download a version of the catalog for offline browsing.
3. The catalog shall allow users to compare the specifications and prices of up to six catalog items.
4. The catalog shall provide browsing and search facilities for users.
5. Users of the catalog shall be able to discover the predicted delivery date for a given number of specific catalog items.
6. Users of the catalog shall be able to place "virtual orders" where the items required will be reserved for them for 48 hours. Virtual orders must be confirmed by a real order placed by a procurement system. The real order must be received within 48 hours of the virtual order.

In addition to these functional requirements, the catalog has a number of non-functional requirements:

1. Access to the catalog service shall be restricted to employees of accredited organizations.
2. The prices and configurations offered to each customer shall be confidential, and access to these shall only be provided to employees of that customer.
3. The catalog shall be available without disruption of service from 0700 GMT to 1100 GMT.
4. The catalog service shall be able to process up to 100 requests per second peak load.

There is no non-functional requirement related to the response time of the catalog service. This depends on the size of the catalog and the expected number of simultaneous users. As this is not a time-critical service, there is no need to specify the required performance at this stage.

18.3.2 Service interface design

Once you have identified candidate services, the next stage in the service engineering process is to design the service interfaces. This involves defining the operations associated with the service and their parameters. If SOAP-based services are used, you have to design the input and output messages. If RESTful services are used, you have to think about the resources required and how the standard operations should be used to implement the service operations.

The starting point for service interface design is abstract interface design. where you identify the entities and the operations associated with the service, their inputs and

Operation	Description
MakeCatalog	Creates a version of the catalog tailored for a specific customer. Includes an optional parameter to create a downloadable PDF version of the catalog.
Lookup	Displays all of the data associated with a specified catalog item.
Search	Takes a logical expression and searches the catalog according to that expression. It displays a list of all items that match the search expression.
Compare	Provides a comparison of up to six characteristics (e.g., price, dimensions, processor speed, etc.) of up to four catalog items.
CheckDelivery	Returns the predicted delivery date for an item if ordered that day.
MakeVirtualOrder	Reserves the number of items to be ordered by a customer and provides item information for the customer's own procurement system.

Figure 18.10 Catalog operations

outputs, and the exceptions associated with these operations. You then need to think about how this abstract interface is realized as SOAP-based or RESTful services.

If you choose a SOAP-based approach, you have to design the structure of the XML messages that are sent and received by the service. The operations and messages are the basis of an interface description written in WSDL. If you choose a RESTful approach, you have to design how the service operations map onto the RESTful operations.

Abstract interface design starts with the service requirements and defines the operation names and parameters. At this stage, you should also define the exceptions that may arise when a service operation is invoked. Figure 18.10 shows the catalog operations that implement the requirements. There is no need for these to be specified in detail; you add detail at the next stage of the design process.

Once you have established an informal description of what the service should do, the next stage is to add more detail of the service inputs and outputs. I have shown this for the catalog service in Figure 18.11, which extends the functional description in Figure 18.10.

Defining exceptions and how these exceptions can be communicated to service users is particularly important. Service engineers do not know how their services will be used. It is usually unwise to make assumptions that service users will have completely understood the service specification. Input messages may be incorrect, so you should define exceptions that report incorrect inputs to the service client. It is generally good practice in reusable component development to leave all exception handling to the user of the component. Service developers should not impose their views on how exceptions should be handled.

In some cases, a textual description of the operations and their inputs and outputs is all that is required. The detailed realization of the service is left as an implementation decision. Sometimes, however, you need to have a more detailed design, and a detailed interface description can be specified in a graphical notation such as the UML or in a readable description format such as JSON. Figure 18.12, which describes the inputs and outputs for the getDelivery operation, shows how you can use the UML to describe the interface in detail.

Operation	Inputs	Outputs	Exceptions
MakeCatalog	*mcIn* Company id PDF-flag	*mcOut* URL of the catalog for that company	*mcFault* Invalid company id
Lookup	*lookIn* Catalog URL Catalog number	*lookOut* URL of page with the item information	*lookFault* Invalid catalog number
Search	*searchIn* Catalog URL Search string	*searchOut* URL of web page with search results	*searchFault* Badly formed search string
Compare	*compIn* Catalog URL Entry attribute (up to 6) Catalog number (up to 4)	*compOut* URL of page showing comparison table	*compFault* Invalid company id Invalid catalog number Unknown attribute
CheckDelivery	*cdIn* Company id Catalog number Number of items required	*cdOut* Expected delivery date	*cdFault* Invalid company id No availability Zero items requested
MakeVirtualOrder	*voIn* Company id Catalog number Number of items required	*voOut* Catalog number Number of items required Predicted delivery date Unit price estimate Total price estimate	*voFault* Invalid company id Invalid catalog number Zero items requested

Figure 18.11 Catalog interface design

Figure 18.12 UML definition of input and output messages

Notice how I have added detail to the description by annotating the UML diagram with constraints. These details define the length of the strings representing the company and the catalog item, and specify that the number of items must be greater than zero and that delivery must be after the current date. The annotations also show which error codes are associated with each possible fault.

The catalog service is an example of a practical service, which illustrates that it is not always straightforward whether to choose a RESTful or a SOAP-based approach to service implementation. As an entity-based service, the catalog can be represented as a resource, which suggests that a RESTful model is the right one to use. However, operations on the catalog are not simple GET operations, and you need to maintain some state in an interaction session with the catalog. This suggests the use of a SOAP-based approach. Such dilemmas are common in service engineering, and usually local circumstances (e.g., availability of expertise) are a major factor in the decision of which approach to use.

To implement a set of RESTful services, you have to decide on the set of resources that will be used to represent the catalog and how the fundamental GET, POST, and PUT operations will operate on these resources. Some of these design decisions are straightforward:

1. There should be a resource representing a company-specific catalog. This should have a URL of the form <base catalog>/<company name> and should be created using a POST operation.

2. Each catalog item should have its own URL of the form <base catalog>/<company name>/<item identifier>.

3. You use the GET operation to retrieve items. Lookup is implemented by using the URL of an item in a catalog as the GET parameter. Search is implemented by using GET with the company catalog as the URL and the search string as a query parameter. This GET operation returns a list of URLs of the items matching the search.

However, the Compare, CheckDelivery, and MakeVirtualOrder operations are more complex:

1. The Compare operation can be implemented as a sequence of GET operations to retrieve the individual items, followed by a POST operation to create the comparison table and a final GET operation to return this to the user.

2. The CheckDelivery and MakeVirtualOrder operations require an additional resource, representing a virtual order. A POST operation is used to create this resource with the number of items required. The company id is used to automatically fill in the order form, and the delivery date is calculated. The resource can then be retrieved using a GET operation.

You need to think carefully about how exceptions are mapped onto the standard http response codes such as a 404 code, meaning that a URL cannot be retrieved. I don't have space to go into this issue here, but it adds a further level of complexity to the service interface design.

Legacy system services

Legacy systems are old software systems that are used by an organization. It may not be cost-effective to rewrite or replace these systems, and many organizations would like to use them in conjunction with more modern systems. One of the most important uses of services is to implement "wrappers" for legacy systems that provide access to a system's functions and data. These systems can then be accessed over the web and integrated with other applications.

http://software-engineering-book.com/web/legacy-services

For SOAP-based services, the realization process, in this case, is simpler as the logical interface design can be translated automatically into WSDL. Most programming environments that support service-oriented development (e.g., the ECLIPSE environment) include tools that can translate a logical interface description into its corresponding WSDL representation.

18.3.3 Service implementation and deployment

Once you have identified candidate services and designed their interfaces, the final stage of the service engineering process is service implementation. This implementation may involve programming the service using a language such as Java or C#. Both of these languages include libraries with extensive support for developing SOAP-based and RESTful services.

Alternatively, you can implement services by creating service interfaces to existing components or legacy systems. Software assets that have already proved to be useful can therefore be made available for reuse. In the case of legacy systems, it may mean that the system functionality can be accessed by new applications. You can also develop new services by defining compositions of existing services, as I explain in Section 18.4.

Once a service has been implemented, it then has to be tested before it is deployed. This involves examining and partitioning the service inputs (as explained in Chapter 8), creating input messages that reflect these input combinations, and then checking that the outputs are expected. You should always try to generate exceptions during the test to check that the service can cope with invalid inputs. For SOAP-based services, testing tools are available that allow services to be examined and tested, and that generate tests from a WSDL specification. However, these tools can only test the conformity of the service interface to the WSDL. They cannot test the service's functional behavior.

Service deployment, the final stage of the process, involves making the service available for use on a web server. Most server software makes this operation straightforward. You install the file containing the executable service in a specific directory. It then automatically becomes available for use.

If the service is intended to be available within a large organization or as a publicly available service, you then have to provide documentation for external service users. Potential users can then decide if the service is likely to meet their needs and

if they can trust you, as a service provider, to deliver the service reliably and securely. Information that you may include in a service description might be:

1. Information about your business, contact details, and so on. This is important for trust reasons. External users of a service have to be confident that it will not behave maliciously. Information about the service provider allows users to check their credentials with business information agencies.
2. An informal description of the functionality provided by the service. This helps potential users to decide if the service is what they want.
3. A description of how to use the service. For simple services, this can be an informal textual description that explains the input and output parameters. For more complex SOAP-based services, the WSDL description may be published.
4. Subscription information that allows users to register for information about updates to the service.

A general difficulty with service specifications is that the functional behavior of the service is usually specified informally, as a natural language description. Natural language descriptions are easy to read, but they are subject to misinterpretation. To address this problem, there has been extensive research on using ontologies and ontology languages for specifying service semantics by marking up the service with ontology information (W3C 2012). However, ontology-based specification is complex and not widely understood. Consequently, it has not been widely used.

18.4 Service composition

The underlying principle of service-oriented software engineering is that you compose and configure services to create new, composite services. These may be integrated with a user interface implemented in a browser to create a web application, or they may be used as components in some other service composition. The services involved in the composition may be specially developed for the application, business services developed within a company, or services from an external provider. Both RESTful and SOAP-based services can be composed to create services with extended functionality.

Many companies have converted their enterprise applications into service-oriented systems, where the basic application building block is a service rather than a component. This allows for widespread reuse within the company. We are now seeing the emergence of interorganizational applications between trusted suppliers, who use each other's services. The final realization of the long-term vision of service-oriented systems will rely on the development of a "services market," where services are bought from trusted external suppliers.

Service composition may be used to integrate separate business processes to provide an integrated process offering more extensive functionality. Say an airline wishes to develop a travel aggregation service that provides a complete vacation package for

Figure 18.13 Vacation package workflow

travelers. In addition to booking their flights, travelers can also book hotels in their preferred location, arrange car rental or book a taxi from the airport, browse a travel guide, and make reservations to visit local attractions. To create this application, the airline composes its own booking service with services offered by a hotel booking agency, rental car and taxi companies, and reservation services offered by owners of local attractions. The end result is a single service that integrates the services from different providers.

You can think of this process as a sequence of separate steps, as shown in Figure 18.13. Information is passed from one step to the next. For example, the rental car company is informed of the time that the flight is scheduled to arrive. The sequence of steps is called a workflow—a set of activities ordered in time, with each activity carrying out some part of the work. A workflow is a model of a business process; that is, it sets out the steps involved in reaching a particular goal that is important for a business. In this case, the business process is the vacation booking service, offered by the airline.

Workflow is a simple idea, and the above scenario of booking a vacation seems to be straightforward. In practice, service composition is usually more complex than this simple model implies. You have to consider the possibility of service failure and include exception management to handle these failures. You also have to take into account nonstandard demands made by users of the application. For example, say a traveler was disabled and required a wheelchair to be rented and delivered to the airport. This would require extra services to be implemented and composed, with additional steps added to the workflow.

When designing a travel aggregation service, you must be able to cope with situations where the normal execution of one of the services results in an incompatibility with some other service execution. For example, say a flight is booked to leave on June 1 and to return on June 7. The workflow then proceeds to the hotel booking stage. However, the resort is hosting a major convention until June 2, so no hotel rooms are available. The hotel booking service reports this lack of availability. This is not a failure; lack of availability is a common situation.

You therefore have to "undo" the flight booking and pass the information about lack of availability back to the user. He or she then has to decide whether to change the dates or the resort. In workflow terminology, this is called a compensation action. Compensation actions are used to undo actions that have already been completed but that must be changed as a result of later workflow activities.

The process of designing new services by reusing existing services is a process of software design with reuse (Figure 18.13). Design with reuse inevitably involves requirements compromises. The "ideal" requirements for the system have to be modified to reflect the services that are actually available, whose costs fall within budget and whose quality of service is acceptable.

Figure 18.14 Service construction by composition

I have shown the six key stages in the process of system construction by composition in Figure 18.14:

1. *Formulate outline workflow* In this initial stage of service design, you use the requirements for the composite service as a basis for creating an "ideal" service design. You should create a fairly abstract design at this stage, with the intention of adding details once you know more about available services.

2. *Discover services* During this stage of the process, you look for existing services to include in the composition. Most service reuse is within enterprises, so this may involve searching local service catalogs. Alternatively, you may search the services offered by trusted service providers, such as Oracle and Microsoft.

3. *Select possible services* From the set of possible service candidates that you have discovered, you then select possible services that can implement workflow activities. Your selection criteria will obviously include the functionality of the services offered. They may also include the cost of the services and the quality of service (responsiveness, availability, etc.) offered.

4. *Refine workflow* On the basis of information about the services that you have selected, you then refine the workflow. This involves adding detail to the abstract description and perhaps adding or removing workflow activities. You may then repeat the service discovery and selection stages. Once a stable set of services has been chosen and the final workflow design established, you move on to the next stage in the process.

5. *Create workflow program* During this stage, the abstract workflow design is transformed to an executable program and the service interface is defined. You can implement workflow programs using a programming language, such as Java or C#, or by using a workflow language, such as BPMN (explained below). This stage may also involve the creation of web-based user interfaces to allow the new service to be accessed from a web browser.

6. *Test completed service or application* The process of testing the completed, composite service is more complex than component testing in situations where external services are used. I discuss testing issues in Section 18.4.2.

This process assumes that existing services are available for composition. If you rely on external information that is not available through a service interface, you may have to implement these services yourself. This usually involves a "screen

scraping" process where your program extracts information from the HTML text of web pages that are sent to a browser for rendering.

18.4.1 Workflow design and implementation

Workflow design involves analyzing existing or planned business processes to understand the tasks involved and how these tasks exchange information. You then define the new business process in a workflow design notation. This sets out the stages involved in enacting the process and the information that is passed between the different process stages. However, existing processes may be informal and dependent on the skills and ability of the people involved. There may be no "normal" way of working or process definition. In such cases, you have to use your knowledge of the current process to design a workflow that achieves the same goals.

Workflows represent business process models. They are graphical models that are written using UML activity diagrams or BPMN, the Business Process Modeling Notation (White and Miers 2008; OMG 2011). I use BPMN for the examples in this chapter. If you use SOAP-based services, it is possible to convert BPMN workflows automatically into WS-BPEL, an XML-based workflow language. This is conformant with other web service standards such as SOAP and WSDL. RESTful services may be composed within a program in a standard programming language such as Java. Alternatively, a composition language used for service mashups may be used (Rosenberg et al. 2008).

Figure 18.15 is an example of a simple BPMN model of part of the vacation package scenario, shown in Figure 18.14. The model shows a simplified workflow for hotel booking and assumes the existence of a **Hotels** service with associated operations called **GetRequirements**, **CheckAvailability**, **ReserveRooms**, **NoAvailability**, **ConfirmReservation**, and **CancelReservation**. The process involves getting requirements from the customer, checking room availability, and then, if rooms are available, making a booking for the required dates.

This model introduces some of the core concepts of BPMN that are used to create workflow models:

1. Rectangles with rounded corners represent activities. An activity can be executed by a human or by an automated service.

2. Circles represent discrete events. An event is something that happens during a business process. A simple circle is used to represent a starting event and a darker circle to represent an end event. A double circle (not shown) is used to represent an intermediate event. Events can be clock events, thus allowing workflows to be executed periodically or timed out.

3. A diamond is used to represent a gateway. A gateway is a stage in the process where some choice is made. For example, in Figure 18.15, a choice is made on the basis of whether or not rooms are available.

4. A solid arrow shows the sequence of activities; a dashed arrow represents message flow between activities. In Figure 18.15, these messages are passed between the hotel booking service and the customer.

Figure 18.15 A fragment of a hotel booking workflow

These key features are enough to describe most workflows. However, BPMN includes many additional features that I don't have space to describe here. These add information to a business process description that allows it to be automatically translated into an executable service.

Figure 18.15 shows a process that is enacted in a single organization, the company that provides a booking service. However, the key benefit of a service-oriented approach is that it supports interorganizational computing. This means that a computation involves processes and services in different companies. This process is represented in BPMN by developing separate workflows for each of the organizations involved with interactions between them.

To illustrate multiple workflow processes, I use a different example, drawn from high-performance computing, where hardware is offered as a service. Services are created to provide access to high-performance computers to a geographically distributed user community. In this example, a vector-processing computer (a machine that can carry out parallel computations on arrays of values) is offered as a service (VectorProcService) by a research laboratory. This is accessed through another service called SetupComputation. These services and their interactions are shown in Figure 18.16.

In this example, the workflow for the SetupComputation service asks for access to a vector processor and, if a processor is available, establishes the computation required and downloads data to the processing service. Once the computation is complete, the results are stored on the local computer. The workflow for VectorProcService includes the following steps:

Check if a processor is available

Allocate resources for the computation

Initialize the system

Figure 18.16 Interacting workflows

Carry out the computation

Return the results to the client service

In BPMN terms, the workflow for each organization is represented in a separate pool. It is shown graphically by enclosing the workflow for each participant in the process in a rectangle, with the name written vertically on the left edge. The workflows in each pool are coordinated by exchanging messages. In situations where different parts of an organization are involved in a workflow, pools are divided into named "lanes." Each lane shows the activities in that part of the organization.

Once a business process model has been designed, it has to be refined depending on the services that have been discovered. As I suggested in the discussion of Figure 18.14, the model may go through a number of iterations until a design that allows the maximum possible reuse of available services has been created.

Once the final design is available, you can then develop the final service-oriented system. This involves implementing services that are not available for reuse and converting the workflow model into an executable program. As services are implementation-language independent, new services can be written in any language. The workflow model may be automatically processed to create an executable WS-BPEL model if SOAP-based services are used. Alternatively, if RESTful services are used, the workflow may be manually programmed, with the model acting as a program specification.

18.4.2 Testing service compositions

Testing is important in all system development processes as it demonstrates that a system meets its functional and non-functional requirements and detects defects that

have been introduced during the development process. Many testing techniques, such as program inspections and coverage testing, rely on analysis of the software source code. However, if you use services from an external provider, you will not have access to the source code of the service implementations. You cannot therefore use "white box" testing techniques that rely on the source code of the system.

As well as problems of understanding the implementation of the service, testers may also face further difficulties when testing service compositions:

1. External services are under the control of the service provider rather than the user of the service. The service provider may withdraw these services at any time or may make changes to them, which invalidates any previous application testing. These problems are handled in software components by maintaining different versions of the component, but service versions are not normally supported.
2. If services are dynamically bound, an application may not always use the same service each time that it is executed. Therefore, tests may be successful when an application is bound to a particular service, but it cannot be guaranteed that that service will be used during an actual execution of the system. This problem has been one reason why dynamic binding has not been widely used.
3. The non-functional behavior of a service is not simply dependent on how it is used by the application that is being tested. A service may perform well during testing because it is not operating under a heavy load. In practice, the observed service behavior may be different because of the demands made by other service users.
4. The payment model for services could make service testing very expensive. There are different possible payment models: Some services may be freely available, some may be paid for by subscription, and others may be paid for on a per-use basis. If services are free, then the service provider will not wish them to be loaded by applications being tested; if a subscription is required, then a service user may be reluctant to enter into a subscription agreement before testing the service. Similarly, if the usage is based on payment for each use, service users may find the cost of testing to be prohibitive.
5. I have discussed the notion of compensation actions that are invoked when an exception occurs and previous commitments that have been made (such as a flight reservation) have to be revoked. There is a problem in testing such actions as they may depend on the failure of other services. Simulating the failure of these services during the testing process is usually difficult.

These problems are particularly acute when external services are used. They are less serious when services are used within the same company or where cooperating companies trust services offered by their partners. In such cases, source code may be available to guide the testing process, and payment for services is unlikely to be a problem. Resolving these testing problems and producing guidelines, tools, and techniques for testing service-oriented applications remains an important research issue.

KEY POINTS

- Service-oriented architecture is an approach to software engineering where reusable, standardized services are the basic building blocks for application systems.
- Services may be implemented within a service-oriented architecture using a set of XML-based web service standards. These include standards for service communication, interface definition, and service enactment in workflows.
- Alternatively, a RESTful architecture may be used, which is based on resources and standard operations on these resources. A RESTful approach uses the http and https protocols for service communication and maps operations on the standard http verbs POST, GET, PUT, and DELETE.
- Services may be classified as utility services that provide a general-purpose functionality, business services that implement part of a business process, or coordination services that coordinate the execution of other services.
- The service engineering process involves identifying candidate services for implementation, defining the service interface, and implementing, testing, and deploying the service.
- The development of software using services is based on the idea that programs are created by composing and configuring services to create new composite services and systems.
- Graphical workflow languages, such as BPMN, may be used to describe a business process and the services used in that process. These languages can describe interactions between the organizations that are involved.

FURTHER READING

There is an immense amount of tutorial material on the web covering all aspects of web services. However, I found the book by Thomas Erl to be the best overview and description of services and service standards. Erl includes some discussion of software engineering issues in service-oriented computing. He has also written more recent books on RESTful services.

Service-Oriented Architecture: Concepts, Technology and Design. Erl has written a number of books on service-oriented systems covering both SOA and RESTful architectures. In this book, Erl discusses SOA and web service standards but mostly concentrates on discussing how a service-oriented approach may be used at all stages of the software process. (T. Erl, Prentice-Hall, 2005).

"Service-oriented architecture." This is a good, readable introduction to SOA. (Various authors, 2008) http://msdn.microsoft.com/en-us/library/bb833022.aspx

"RESTful Web Services: The Basics." A good introductory tutorial on the RESTful approach and RESTful services. (A. Rodriguez, 2008). https://www.ibm.com/developerworks/webservices/library/ws-restful/

Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL, and RESTful Web Services. This is a more advanced text for developers who wish to use web services in enterprise applications. It describes a number of common problems and abstract web service solutions to these problems. (R. Daigneau, Addison-Wesley, 2012).

"Web Services Tutorial." This is an extensive tutorial on all aspects of service-oriented architecture, web services, and web service standards, written by people involved in the development of these standards. Very useful if you need a detailed understanding of the standards. (W3C schools, 1999–2014) http://www.w3schools.com/webservices/default.asp

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-reuse/

EXERCISES

18.1. Why is it important to define exceptions in service engineering?

18.2. Standards are fundamental to service-oriented architectures, and it was believed that standards conformance was essential for successful adoption of a service-based approach. However, RESTful services, which are increasingly widely used, are not standards-based. Discuss why you think this change has occurred and whether or not you think that the lack of standards will inhibit the development and takeup of RESTful services.

18.3. Extend Figure 18.5 to include WSDL definitions for `MaxMinType` and `InDataFault`. The temperatures should be represented as integers, with an additional field indicating whether the temperature is in degrees Fahrenheit or degrees Celsius. `InDataFault` should be a simple type consisting of an error code.

18.4. Suggest how the SimpleInterestCalculator service could be implemented as a RESTful service.

18.5. What is a workflow? List out the key stages in the process of system construction by composition.

18.6. Design possible input and output messages for the services shown in Figure 18.13. You may specify these in the UML or in XML.

18.7. Giving reasons for your answer, suggest two important types of application where you would *not* recommend the use of service-oriented architecture.

18.8. Explain what is meant by a "compensation action" and, using an example, show why these actions may have to be included in workflows.

18.9. For the example of the vacation package reservation service, design a workflow that will book ground transportation for a group of passengers arriving at an airport. They should be given the option of booking either a taxi or a hire car. You may assume that the taxi and rental car companies offer web services to make a reservation.

18.10. Using an example, explain in detail why the thorough testing of services that include compensation actions is difficult.

REFERENCES

Erl, T. 2004. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Upper Saddle River, NJ: Prentice-Hall.

———. 2005. *Service-Oriented Architecture: Concepts, Technology and Design*. Upper Saddle River, NJ: Prentice-Hall.

Fielding, R. 2000. "Representational State Transfer." *Architectural Styles and the Design of Network-Based Software Architecture*. https://www.ics.uci.edu/~fielding/pubs/. . ./fielding_dissertation.pdf

Lovelock, C, S Vandermerwe, and B Lewis. 1996. *Services Marketing*. Englewood Cliffs, NJ.: Prentice-Hall.

Newcomer, E., and G. Lomow. 2005. *Understanding SOA with Web Services*. Boston: Addison-Wesley.

OMG. 2011. "Documents Associated with Business Process Model and Notation (BPMN) Version 2.0." http://www.omg.org/spec/BPMN/2.0/

Pautasso, C., O. Zimmermann, and F. Leymann. 2008. "RESTful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision." In *Proc. WWW 2008*, 805–14. Beijing, China. doi:10.1145/1367497.1367606.

Richardson, L., and S. Ruby. 2007. *RESTful Web Services*. Sebastopol, CA: O'Reilly Media Inc.

Rosenberg, F., F. Curbera, M. Duftler, and R. Khalaf. 2008. "Composing RESTful Services and Collaborative Workflows: A Lightweight Approach." *IEEE Internet Computing* 12 (5): 24–31. doi:10.1109/MIC.2008.98.

W3C. 2012. "OWL 2 Web Ontology Language." http://www.w3.org/TR/owl2-overview/

———. 2013. "Web of Services." http://www.w3.org/standards/webofservices/

White, S. A., and D. Miers. 2008. *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Lighthouse Point, FL. USA: Future Strategies Inc.

19

Systems engineering

Objectives

The objectives of this chapter are to explain why software engineers should understand systems engineering and to introduce the most important systems engineering processes. When you have read this chapter, you will:

- know what is meant by a sociotechnical system and understand why human, social, and organizational issues affect the requirements and design of software systems;
- understand the idea of conceptual design and why it is an essential first stage in the systems engineering process;
- know what is meant by system procurement and understand why different system procurement processes are used for different types of system;
- know about the key systems engineering development processes and their relationships.

Contents

19.1 Sociotechnical systems
19.2 Conceptual design
19.3 System procurement
19.4 System development
19.5 System operation and evolution

A computer only becomes useful when it includes both software and hardware. Without hardware, a software system is an abstraction—simply a representation of some human knowledge and ideas. Without software, a hardware system is a set of inert electronic devices. However, if you put them together to form a computer system, you create a machine that can carry out complex computations and deliver the results of these computations to its environment.

This illustrates one of the fundamental characteristics of a system: It is more than the sum of its parts. Systems have properties that only become apparent when their components are integrated and operate together. Furthermore, systems are developed to support human activities—work, entertainment, communication, protection of people and the environment, and so on. They interact with people, and their design is influenced by human and organizational concerns. Hardware, human, social, and organizational factors have to be taken into account when developing all professional software systems.

Systems that include software fall into two categories:

1. *Technical computer-based systems* are systems that include hardware and software components but not procedures and processes. Examples of technical systems include televisions, mobile phones, and other equipment with embedded software. Applications for PCs, computer games, and mobile devices are also technical systems. Individuals and organizations use technical systems for a particular purpose, but knowledge of this purpose is not part of the technical system. For example, the word processor I am using (Microsoft Word) is not aware that it is being used to write a book.

2 *Sociotechnical systems:* include one or more technical systems but, crucially, also people, who understand the purpose of the system, within the system itself. Sociotechnical systems have defined operational processes, and people (the operators) are inherent parts of the system. They are governed by organizational policies and rules and may be affected by external constraints such as national laws and regulatory policies. For example, this book was created through a sociotechnical publishing system that includes various processes (creation, editing, layout, etc.) and technical systems (Microsoft Word and Excel, Adobe Illustrator, InDesign, etc.).

Systems engineering (White et al. 1993; Stevens et al. 1998; Thayer 2002) is the activity of designing entire systems, taking into account the characteristics of hardware, software, and human elements of these systems. Systems engineering includes everything to do with procuring, specifying, developing, deploying, operating, and maintaining both technical and sociotechnical systems. Systems engineers have to consider the capabilities of hardware and software as well as the system's interactions with users and its environment. They must think about the system's services, the constraints under which the system must be built and operated, and the ways in which the system is used.

In this chapter, my focus is on the engineering of large and complex software-intensive systems. These are "enterprise systems," that is, systems that are used to support the goals of a large organization. Enterprise systems are used by government and the military services as well as large companies and other public bodies.

Figure 19.1 Stages of systems engineering

They are sociotechnical systems that are influenced by the ways that the organization works and by national and international rules and regulations. They may be made up of a number of separate systems and are distributed systems with large-scale databases. They have a long lifetime and are critical for the operation of the enterprise.

I believe that it is important for software engineers to know about systems engineering and to be active participants in systems engineering processes for two reasons:

1. Software is now the dominant element in all enterprise systems, yet many senior decision makers in organizations have a limited understanding of software. Software engineers have to play a more active part in high-level systems decision making if the system software is to be dependable and developed on time and to budget.

2. As a software engineer, it helps if you have a broader awareness of how software interacts with other hardware and software systems, and the human, social, and organizational factors that affect the ways in which software is used. This knowledge helps you understand the limits of software and to design better software systems.

There are four overlapping stages (Figure 19.1) in the lifetime of large, complex systems:

1. *Conceptual design* This initial systems engineering activity develops the concept of the type of system that is required. It sets out, in nontechnical language, the purpose of the system, why it is needed, and the high-level features that users might expect to see in the system. It may also describe broad constraints, such as the need for interoperability with other systems. These limit the freedom of systems engineers in designing and developing the system.

2. *Procurement or acquisition* During this stage, the conceptual design is further developed so that information is available to make decisions about the contract for the system development. This may involve making decisions about the distribution of

functionality across hardware, software, and operational processes. You also make decisions about which hardware and software has to be acquired, which suppliers should develop the system, and the terms and conditions of the supply contract.

3. *Development* During this stage, the system is developed. Development processes include requirements definition, system design, hardware and software engineering, system integration, and testing. Operational processes are defined, and the training courses for system users are designed.

4. *Operation* At this stage, the system is deployed, users are trained, and the system is brought into use. The planned operational processes usually then have to change to reflect the real working environment where the system is used. Over time, the system evolves as new requirements are identified. Eventually, the system declines in value, and it is decommissioned and replaced.

Figure 19.1 shows the interactions between these stages. The conceptual design activity is a basis for the system procurement and development but is also used to provide information to users about the system. Development and procurement overlap and further procurement during development, and operation may be needed as new equipment and software become available. Once the system is operational, requirements changes are inevitable; implementing these changes requires further development and, perhaps, software and hardware procurement.

Decisions made at any one of these stages may have a profound influence on the other stages. Design options may be restricted by procurement decisions on the scope of the system and on its hardware and software. Human errors made during the specification, design, and development stages may mean that faults are introduced into the system. A decision to limit testing for budget reasons may mean that faults are not discovered before a system is put into use. During operation, errors in configuring the system for deployment may lead to problems in using the system. Decisions made during the original procurement may be forgotten when system changes are proposed. This may lead to unforeseen consequences arising from the implementation of the changes.

An important difference between systems and software engineering is the involvement of a range of professionals throughout the lifetime of the system. These include engineers who may be involved in hardware and software design, system end-users, managers who are concerned with organizational issues, and experts in the system's application domain. For example, engineering the insulin pump system introduced in Chapter 1 requires experts in electronics, mechanical engineering, software, and medicine.

For very large systems, an even wider range of expertise may be required. Figure 19.2 illustrates the technical disciplines that may be involved in the procurement and development of a new system for air traffic management. Architects and civil engineers are involved because new air traffic management systems usually have to be installed in a new building. Electrical and mechanical engineers are involved to specify and maintain the power and air conditioning. Electronic engineers are concerned with computers, radars, and other equipment. Ergonomists

Figure 19.2 Professional disciplines involved in ATC systems engineering

design the controller workstations and software engineers, and user interface designers are responsible for the software in the system.

The involvement of a range of professional disciplines is essential because of the different types of components in complex systems. However, differences and misunderstandings between disciplines can lead to inappropriate design decisions. These poor decisions can delay the system's development or make it less suitable for its intended purpose. There are three reasons why there may be misunderstandings or other differences between engineers with different backgrounds:

1. Different professional disciplines often use the same words, but these words do not always mean the same thing. Consequently, misunderstandings are common in discussions between engineers from different backgrounds. If these are not discovered and resolved during system development, they can lead to errors in delivered systems. For example, an electronic engineer may know a bit about C programming but may not understand that a method in Java is like a function in C.

2. Each discipline makes assumptions about what other disciplines can or cannot do. These assumptions are often based on an inadequate understanding of what is possible. For example, an electronic engineer may decide that all signal processing (a computationally intensive task) should be done by software to simplify the hardware design. However, this may mean significantly greater software effort to ensure that the system processor can cope with the amount of computation that is resolved.

3. Disciplines try to protect their professional boundaries and may argue for certain design decisions because these decisions will call for their professional expertise. Therefore, a software engineer may argue for a software-based door locking system in a building, although a mechanical, key-based system may be more reliable.

My experience is that interdisciplinary working can be successful only if enough time is available for these issues to be discussed and resolved. This requires regular face-to-face discussions and a flexible approach from everyone involved in the systems engineering process.

19.1 Sociotechnical systems

The term *system* is universally used. We talk about computer systems, operating systems, payment systems, the education system, the system of government, and so on. These are all obviously quite different uses of the word "system," although they share the essential characteristic that, somehow, the system is more than simply the sum of its parts.

Abstract systems, such as the system of government, are outside the scope of this book. I focus here on systems that include computers and software and that have some specific purpose such as to enable communication, support navigation, or maintain medical records. A useful working definition of these types of system is as follows:

A system is a purposeful collection of interrelated components of different kinds that work together to deliver a set of services to the system owner and its users.

This general definition can cover a very wide range of systems. For example, a simple system, such as a laser pointer, delivers an indication service. It may include a few hardware components with a tiny control program in read-only memory (ROM). By contrast, an air traffic control system includes thousands of hardware and software components as well as human users who make decisions based on information from that computer system. It delivers a range of services, including providing information to pilots, maintaining safe separation of planes, utilizing airspace, and so on.

In all complex systems, the properties and behavior of the system components are inextricably intermingled. The successful functioning of each system component depends on the functioning of other components. Software can only operate if the processor is operational. The processor can only carry out computations if the software system defining these computations has been successfully installed.

Large-scale systems are often "systems of systems." That is, they are made up of several separate systems. For example, a police command and control system may include a geographical information system to provide details of the location of incidents. The same geographical information system may be used in systems for transport logistics and emergency command and control. Engineering systems of systems is an increasingly important topic in software engineering that I cover in Chapter 20.

Large-scale systems are, with a few exceptions, sociotechnical systems, which I explained in Chapter 10. That is, they do not just include software and hardware but also people, processes, and organizational policies. Sociotechnical systems are enterprise systems that are intended to help deliver a business purpose. This purpose might be to increase sales, reduce material used in manufacturing, collect taxes, maintain a safe airspace, and so on. Because they are embedded in an organizational environment, the procurement, development, and use of these systems are influenced by the organization's policies and procedures, as well as by its working culture. The users of the system are people who are influenced by the way the organization is managed and by their interactions with other people inside and outside of the organization.

The close relationships between sociotechnical systems and the organizations that use these systems means that it is often difficult to establish system boundaries.

Figure 19.3 Layered structure of sociotechnical systems

Different people within the organization will see the boundaries of the system in different ways. This is significant because establishing what is and what is not in the scope of the system is important when defining the system requirements.

Figure 19.3 illustrates this problem. The diagram shows a sociotechnical system as a set of layers, where each layer contributes, in some way, to the functioning of the system. At the core is a software-intensive technical system and its operational processes (shaded in Figure 19.3). Most people would agree that these are both parts of the system. However, the system's behavior is influenced by a range of sociotechnical factors outside of the core. Should the system boundary simply be drawn around the core, or should it include other organizational levels?

Whether or not these broader sociotechnical considerations should be considered to be part of a system depends on the organization and its policies and rules. If organizational rules and policies can be changed, then some people might argue they should be part of the system. However, it is more difficult to change organizational culture and even more challenging to change strategy and goals. Only governments can change laws to accommodate a system. Moreover, different stakeholders may have different opinions on where the system boundaries should be drawn. There are no simple answers to these questions, but they have to be discussed and negotiated during the system design process.

Generally, large sociotechnical systems are used in organizations. When you are designing and developing sociotechnical systems, you need to understand, as far as possible, the organizational environment in which they will be used. If you don't, the systems may not meet business needs. Users and their managers may reject the system or fail to use it to its full potential.

Figure 19.4 shows the key elements in an organization that may affect the requirements, design, and operation of a sociotechnical system. A new system may lead to changes in some or all of these elements:

1. *Process changes* A new system may mean that people have to change the way that they work. If so, training will certainly be required. If changes are significant, or if they involve people losing their jobs, there is a danger that the users will resist the introduction of the system.

Figure 19.4
Organizational elements

2. *Job changes* New systems may deskill the users in an environment or cause them to change the way they work. If so, users may actively resist the introduction of the system into the organization. Professional staff, such as doctors or teachers, may resist system designs that require them to change their normal way of working. The people involved may feel that their professional expertise is being eroded and that their status in the organization is being reduced by the system.

3. *Organizational policies* The proposed system may not be completely consistent with organizational policies (e.g., on privacy). This may require system changes, policy changes, or process changes to bring the system and policies into line.

4. *Organizational politics* The system may change the political power structure in an organization. For example, if an organization is dependent on a complex system, those who control access to that system have a great deal of political power. Alternatively, if an organization reorganizes itself into a different structure, this may affect the requirements and use of the system.

Sociotechnical systems are complex systems, which means that it is practically impossible to have a complete understanding, in advance, of their behavior. This complexity leads to three important characteristics of sociotechnical systems:

1. They have emergent properties that are properties of the system as a whole, rather than associated with individual parts of the system. Emergent properties depend on both the system components and the relationships between them. Some of these relationships only come into existence when the system is integrated from its components, so the emergent properties can only be evaluated at that time. Security and dependability are examples of important emergent system properties.

2. They are nondeterministic, so that when presented with a specific input, they may not always produce the same output. The system's behavior depends on the human operators, and people do not always react in the same way. Furthermore, use of the system may create new relationships between the system components and hence change its emergent behavior.

3. The system's success criteria are subjective rather than objective. The extent to which the system supports organizational objectives does not just depend on the system itself. It also depends on the stability of these objectives, the relationships

Property	Description
Reliability	System reliability depends on component reliability, but unexpected interactions can cause new types of failure and therefore affect the reliability of the system.
Repairability	This property reflects how easy it is to fix a problem with the system once it has been discovered. It depends on being able to diagnose the problem, access the components that are faulty, and modify or replace these components.
Security	The security of the system (its ability to resist attack) is a complex property that cannot be easily measured. Attacks may be devised that were not anticipated by the system designers and so may defeat built-in safeguards.
Usability	This property reflects how easy it is to use the system. It depends on the technical system components, its operators, and its operating environment.
Volume	The volume of a system (the total space occupied) depends on how the component assemblies are arranged and connected.

Figure 19.5 Examples of emergent properties

and conflicts between organizational objectives, and how people in the organization interpret these objectives. New management may reinterpret the organizational objectives that a system was designed to support so that a "successful" system may then be seen as no longer fit for its intended purpose.

Sociotechnical considerations are often critical in determining whether or not a system has successfully met its objectives. Unfortunately, taking these into account is very difficult for engineers who have little experience of social or cultural studies. To help understand the effects of systems on organizations, various sociotechnical systems methodologies have been proposed. My paper on sociotechnical systems design discusses the advantages and disadvantages of these sociotechnical design methodologies (Baxter and Sommerville 2011).

19.1.1 Emergent properties

The complex relationships between the components in a system mean that a system is more than simply the sum of its parts. It has properties that are properties of the system as a whole. These "emergent properties" (Checkland 1981) cannot be attributed to any specific part of the system. Rather, they only emerge once the system components have been integrated. Some emergent properties, such as weight, can be derived directly from the subsystem properties. More often, however, they emerge from a combination of subsystem properties and subsystem relationships. The system property cannot be calculated directly from the properties of the individual system components. Examples of emergent properties are shown in Figure 19.5.

There are two types of emergent properties:

1. *Functional emergent properties,* when the purpose of a system only emerges after its components are integrated. For example, a bicycle has the functional property of being a transportation device once it has been assembled from its components.

2. *Non-functional emergent properties,* which relate to the behavior of the system in its operational environment. Reliability, performance, safety, and security are examples of these properties. These system characteristics are critical for computer-based systems, as failure to achieve a minimum defined level in these properties usually makes the system unusable. Some users may not need some of the system functions, so the system may be acceptable without them. However, a system that is unreliable or too slow is likely to be rejected by all its users.

Emergent properties, such as reliability, depend on both the properties of individual components and their interactions or relationships. For example, the reliability of a sociotechnical system is influenced by three things:

1. *Hardware reliability* What is the probability of hardware components failing, and how long does it take to repair a failed component?
2. *Software reliability* How likely is it that a software component will produce an incorrect output? Software failure is unlike hardware failure in that software does not wear out. Failures are often transient. The system carries on working after an incorrect result has been produced.
3. *Operator reliability* How likely is it that the operator of a system will make an error and provide an incorrect input? How likely is it that the software will fail to detect this error and propagate the mistake?

Hardware, software, and operator reliability are not independent but affect each other in unpredictable ways. Figure 19.6 shows how failures at one level can be propagated to other levels in the system. Say a hardware component in a system starts to go wrong. Hardware failure can sometimes generate spurious signals that are outside the range of inputs expected by the software. The software can then behave unpredictably and produce unexpected outputs. These may confuse and consequently cause stress in the system operator.

We know that people are more likely to make mistakes when they feel stressed. So a hardware failure may be the trigger for operator errors. These mistakes can, in turn, lead to unexpected software behavior, resulting in additional demands on the processor. This could overload the hardware, causing more failures and so on. Thus, an initial, relatively minor, failure, can rapidly develop into a serious problem that could lead to a complete shutdown of the system.

The reliability of a system depends on the context in which that system is used. However, the system's environment cannot be completely specified, and it is often impossible for the system designers to limit the environment for operational systems. Different systems operating within an environment may react to problems in unpredictable ways, thus affecting the reliability of all of these systems.

For example, say a system is designed to operate at normal room temperature. To allow for variations and exceptional conditions, the electronic components of a system are designed to operate within a certain range of temperatures, say, from 0 degrees to 40 degrees Celsius. Outside this temperature range, the components will

Figure 19.6 Failure propagation

behave in an unpredictable way. Now assume that this system is installed close to an air conditioner. If this air conditioner fails and vents hot gas over the electronics, then the system may overheat. The components, and hence the whole system may then fail.

If this system had been installed elsewhere in that environment, this problem would not have occurred. When the air conditioner worked properly, there were no problems. However, because of the physical closeness of these machines, an unanticipated relationship existed between them that led to system failure.

Like reliability, emergent properties such as performance or usability are hard to assess but can be measured after the system is operational. Properties such as safety and security, however, are not directly measurable. Here, you are not simply concerned with attributes that relate to the behavior of the system but also with unwanted or unacceptable behavior.

A secure system is one that does not allow unauthorized access to its data. Unfortunately, it is clearly impossible to predict all possible modes of access and explicitly forbid them. Therefore, it may only be possible to assess these "shall not" properties after the system is operational. That is, you only know that a system is insecure when someone manages to penetrate the system.

19.1.2 Non-determinism

A deterministic system is one that is absolutely predictable. If we ignore issues of concurrency, software systems that run on reliable hardware are deterministic. When they are presented with a sequence of inputs they will always produce the same sequence of outputs. Of course, there is no such thing as completely reliable hardware, but hardware is usually reliable enough to think of hardware systems as deterministic.

People, on the other hand, are non-deterministic. When presented with exactly the same input (say a request to complete a task), their responses will depend on their emotional and physical state, the person making the request, other people in the environment, and whatever else they are doing. Sometimes they will be happy to do the work, and, at other times, they will refuse; sometimes they will perform a task well, and sometimes they will do it badly.

Sociotechnical systems are nondeterministic partly because they include people and partly because changes to the hardware, software, and data in these systems are

so frequent. The interactions between these changes are complex, and so the behavior of the system is unpredictable. Users do not know when and why changes have been made, so they see the system as nondeterministic.

For example, say a system is presented with a set of 20 test inputs. It processes these inputs and the results are recorded. At some later time, the same 20 test inputs are processed, and the results are compared to the previous stored results. Five of them are different. Does this mean that there have been five failures? Or are the differences simply reasonable variations in the system's behavior? You can only find this out by looking at the results in more depth and making judgments about the way the system has handled each input.

Non-determinism is often seen as a bad thing, and it is felt that designers should try to avoid nondeterministic behavior wherever possible. In fact, in sociotechnical systems, non-determinism has important benefits. It means that the behavior of a system is not fixed for all time but can change depending on the system's environment. For example, operators may observe that a system is showing signs of failure. Instead of using the system normally, they can change their behavior to diagnose and recover from the detected problems.

19.1.3 Success criteria

Generally, complex sociotechnical systems are developed to tackle "wicked problems" (Rittel and Webber 1973). A wicked problem is a problem that is so complex and that involves so many related entities that there is no definitive problem specification. Different stakeholders see the problem in different ways, and no one has a full understanding of the problem as a whole. The true nature of the problem may only emerge as a solution is developed.

An extreme example of a wicked problem is emergency planning to deal with the aftermath of an earthquake. No one can accurately predict where the epicenter of an earthquake will be, what time it will occur, or what effect it will have on the local environment. It is impossible to specify in detail how to deal with the problem. System designers have to make assumptions, but understanding what is required emerges only after the earthquake has happened.

This makes it difficult to define the success criteria for a system. How do you decide if a new system contributes to the business goals of the company that paid for the system? The judgment of success is not usually made against the original reasons for procuring and developing the system. Rather, it is based on whether or not the system is effective at the time it is deployed. As the business environment can change very quickly, the business goals may have changed significantly during the development of the system.

The situation is even more complex when there are multiple conflicting goals that are interpreted differently by different stakeholders. For instance, the system on which the Mentcare system is based was designed to support two separate business goals:

1. To improve the quality of care for sufferers from mental illness.
2. To improve the cost-effectiveness of treatments by providing managers with detailed reports of care provided and the costs of that care.

Unfortunately, these proved to be conflicting goals because the information that was needed to satisfy the reporting goal meant that doctors and nurses had to provide additional information, over and above the health records that they normally maintained. This reduced the quality of care for patients as it meant that clinical staff had less time to talk with them. From a doctor's perspective, this system was not an improvement on the previous manual system, but from a manager's perspective, it was.

Thus, any success criteria that are established in the early stages of the systems engineering process have to be regularly reconsidered during system development and use. You cannot evaluate these criteria objectively as they depend on the system's effect on its environment and its users. A system may apparently meet its requirements as originally specified but be practically useless because of changes in the environment where it is used.

19.2 Conceptual design

Once an idea for a system has been suggested, conceptual design is the very first thing that you do in the systems engineering process. In the conceptual design phase, you take that initial idea, investigate its feasibility, and develop it to create an overall vision of a system that could be developed. You then have to describe the envisaged system so that nonexperts, such as system users, senior company decision makers, or politicians, can understand what you are proposing.

There is an obvious overlap between conceptual design and requirements engineering. As part of the conceptual design process, you have to imagine how the proposed system will be used. This may involve discussions with potential users and other stakeholders, focus groups, and observations of how existing systems are used. The goal of these activities is to understand how users work, what is important to them, and what practical constraints on the system there might be.

The importance of establishing a vision of a proposed system is rarely mentioned in the software design and requirements literature. However, this vision has been part of the systems engineering process for military systems for many years. Fairley et al. (Fairley, Thayer, and Bjorke 1994) discuss the idea of concept analysis and the documentation of the results of concept analysis in a "Concept of Operations" (ConOps) document. This idea of developing a ConOps document is now widely used for large-scale systems, and you can find many examples of ConOps documents on the web.

Unfortunately, as is so often the case with military and government systems, good ideas can become mired in bureaucracy and inflexible standards. This is exactly what happened with ConOps, and a ConOps document standard was proposed (IEEE, 2007). As Mostashari et al. say (Mostashari et al. 2012), this tends to lead to long and unreadable documents, which do not serve their intended purpose. They propose a more agile approach to the development of a ConOps document with a shorter and more flexible document as the output of the process.

Figure 19.7 Conceptual design activities

I don't like the term *Concept of Operations* partly because of its military connotations and partly because I think that a conceptual design document is not just about system operation. It should also present the system engineer's understanding of why the system is being developed, an explanation of why the design proposals are appropriate, and, sometimes, an initial organization for the system. As Fairley says, "It should be organized to tell a story," that is, written so that people without a technical background can understand the proposals that are being made.

Figure 19.7 shows activities that may be part of the conceptual design process. Conceptual design should always be a team process that involves people from different backgrounds. I was part of the conceptual design team for the digital learning environment, introduced in Chapter 1. For the digital learning environment, the design team included teachers, education researchers, software engineers, system administrators, and system managers.

Concept formulation is the first stage of the process where you try to refine an initial statement of needs and work out what type of system would be best to meet the needs of system stakeholders. Initially, we were tasked with proposing an intranet for information sharing across schools that was easier to use than the current system. However, after discussions with teachers, we discovered that this was not really what was required. The existing system was awkward to use, but people had found workarounds. What was really required was a flexible digital learning environment that could be adapted by adding subject and age-specific tools and content that are freely available on the Internet.

We discovered this because the concept formulation activity overlapped with the activity of problem understanding. To understand a problem, you need to discuss with users and other stakeholders how they do their work. You need to find out what is important to them, what are the barriers that stop them from doing what they want to do, and their ideas of what changes are required. You need to be open-minded (it is their problem, not yours) and to be prepared to change your ideas when the reality does not match your initial vision.

In the system proposal development stage, the conceptual design team set out their ideas for alternative systems and these are the basis for a feasibility study to decide which of the ideas are worth further development. In a feasibility study, you should look at comparable systems that have been developed elsewhere and technological issues (e.g., use of mobile devices) that may affect use of the system. Then you need to assess whether or not the system could be implemented using current hardware and software technologies.

I have found that an additional useful activity is to develop an outline structure or architecture for the system. This activity is helpful both for making a feasibility assessment and for providing a basis for more detailed requirements engineering and architectural design. Furthermore, as the majority of systems are now assembled from existing systems and components, an initial architecture means that the key parts of the system have been identified and can be procured separately. This approach is often better than procuring a system as a monolithic unit from a single supplier.

For the digital learning environment, we decided on a layered service architecture (shown in Figure 1.8). All components in the system should be considered to be replaceable services. In this way, users can replace a standard service with their preferred alternative and so adapt the system to the ages and interests of the students learning with the system.

All of these activities generate information that is used to develop the system vision document. This is a critical document that senior decision makers use to decide whether or not further development of the system should go ahead. It is also used to develop further documents such as a risk analysis and budget estimate, which are also important inputs to the decision-making process.

Managers use the system vision document to understand the system; a procurement team uses it to define a tender document; and requirements engineers use it as a basis for refining the system requirements. Because these different people need different levels of detail, I suggest that the document should be structured into two parts:

1. A short summary for senior decision makers that presents the key points of the problem and the proposed system. It should be written so that readers can immediately see how the system will be used and the benefits that it will provide.

2. A number of appendices that develop the ideas in more detail and that can be used in the system procurement and requirements engineering activities.

It is challenging to write a summary of the system vision inasmuch as the readers are busy people who are unlikely to have a technical background. I have found that using user stories is very effective, providing a tangible vision of system use that nontechnical people can relate to. Stories should be short and personalized and should be a feasible description of the use of the system, as shown in Figure 19.8. There is another example of a user story from the same system in Chapter 4 (Figure 4.9).

Digital art

Jill is an S2 pupil at a secondary school in Dundee. She has a smartphone of her own, and the family has a shared Samsung tablet and a Dell laptop computer. At school, Jill signs on to the school computer and is presented with a personalized Glow+ environment, which includes a range of services, some chosen by her teachers and some she has chosen herself from the Glow app library.

She is working on a Celtic art project, and she uses Google to research a range of art sites. She sketches out some designs on paper and then uses the camera on her phone to photograph what she has done; she uploads this using the school wifi to her personal Glow+ space. Her homework is to complete the design and write a short commentary on her ideas.

At home, she uses the family tablet to sign on to Glow+, and she then uses an artwork app to process her photograph and to extend the work, add color, and so on. She finishes this part of the work, and to complete it she moves to her home laptop to type up her commentary. She uploads the finished work to Glow+ and sends a message to her art teacher that it is available for review. Her teacher looks at the project in a free period before Jill's next art class using a school tablet, and, in class, she discusses the work with Jill.

After the discussion, the teacher and Jill decide that the work should be shared, and so they publish it to the school web pages that show examples of students' work. In addition, the work is included in Jill's e-portfolio—her record of schoolwork from age 3 to 18.

Figure 19.8 A user story used in a system vision document

User stories are effective because, as already noted, readers can relate to them; in addition, they can show the capabilities of the proposed system in an easily accessible way. Of course, these are only part of a system vision, and the summary must also include a high-level description of the basic assumptions made and the ways in which the system will deliver value to the organization.

19.3 System procurement

System procurement or system acquisition is a process whose outcome is a decision to buy one or more systems from system suppliers. At this stage, decisions are made on the scope of a system that is to be purchased, system budgets and timescales, and high-level system requirements. Using this information, further decisions are then made on whether to procure a system, the type of system required, and the supplier or suppliers of the system. The drivers for these decisions are:

1. *The replacement of other organizational systems* If the organization has a mixture of systems that cannot work together or that are expensive to maintain, then procuring a replacement system, with additional capabilities, may lead to significant business benefits.

2. *The need to comply with external regulations* Increasingly, businesses are regulated and have to demonstrate compliance with externally defined regulations (e.g., Sarbanes–Oxley accounting regulations in the United States). Compliance may require the replacement of noncompliant systems or the provision of new systems specifically to monitor compliance.

3. *External competition* If a business needs to compete more effectively or maintain a competitive position, managers may decide to buy new systems to improve business efficiency or effectiveness. For military systems, the need to improve capability in the face of new threats is an important reason for procuring new systems.

4. *Business reorganization* Businesses and other organizations frequently restructure with the intention of improving efficiency and/or customer service. Reorganizations lead to changes in business processes that require new systems support.

5. *Available budget* The budget that is available is an obvious factor in determining the scope of new systems that can be procured.

In addition, new government systems are often procured to reflect political changes and political policies. For example, politicians may decide to buy new surveillance systems, which they claim will counter terrorism. Buying such systems shows voters that they are taking action.

Large complex systems are usually engineered using a mixture of off-the-shelf and specially built components. They are often integrated with existing legacy systems and organizational databases. When legacy systems and off-the-shelf systems are used, new custom software may be needed to integrate these components. The new software manages the component interfaces so that these components can interoperate. The need to develop this "glueware" is one reason why the savings from using off-the-shelf components are sometimes not as great as anticipated.

Three types of systems or system components may have to be procured:

1. Off-the-shelf applications that may be used without change and that need only minimal configuration for use.

2. Configurable application or ERP systems that have to be modified or adapted for use either by modifying the code or by using inbuilt configuration features, such as process definitions and rules.

3. Custom systems that have to be specially designed and implemented for use.

Each of these components tends to follow a different procurement process. Figure 19.9 illustrates the main features of the procurement process for these types of system. Key issues that affect procurement processes are:

1. Organizations often have an approved and recommended set of application software that has been checked by the IT department. It is usually possible to buy or acquire open-source software from this set directly without the need for detailed justification. For example, in the iLearn system, we recommended that Wordpress should be made available for student and staff blogs. If microphones are needed, off-the-shelf hardware can be bought. There are no detailed requirements, and the users adapt to the features of the chosen application.

2. Off-the-shelf components do not usually match requirements exactly, unless the requirements have been written with these components in mind. Therefore, choosing

Figure 19.9 System procurement processes

a system means that you have to find the closest match between the system requirements and the facilities offered by off-the-shelf systems. ERP and other large-scale application systems usually fall into this category. You may then have to modify the requirements to fit in with the system assumptions. This can have knock-on effects on other subsystems. You also usually have an extensive configuration process to tailor and adapt the application or ERP system to the buyer's working environment.

3. When a system is to be built specially, the specification of requirements is part of the contract for the system being acquired. It is therefore a legal as well as a technical document. The requirements document is critical, and procurement processes of this type usually take a considerable amount of time.

4. For public sector systems in particular, there are detailed rules and regulations that affect the procurement of systems. For example, in the European Union, all public sector systems over a certain price must be open to tender by any supplier in Europe. This requires detailed tender documents to be drawn up and the tender to be advertised across Europe for a fixed period of time. Not only does this rule slow down the procurement process, it also tends to inhibit agile development. It forces the system buyer to develop requirements so that all companies have enough information to bid for the system contract.

5. For application systems that require change or for custom systems, there is usually a contract negotiation period when the customer and supplier negotiate the terms and conditions for development of the system. Once a system has been

selected, you may negotiate with the supplier on costs, license conditions, possible changes to the system, and other contractual issues. For custom systems, negotiations are likely to involve payment schedules, reporting, acceptance criteria, requirements change requests, and costs of system changes. During this process, requirements changes may be agreed that will reduce the overall costs and avoid some development problems.

Complex sociotechnical systems are rarely developed "in house" by the buyer of the system. Rather, external systems companies are invited to bid for the systems engineering contract. The customer's business is not systems engineering, so its employees do not have the skills needed to develop the systems themselves. For complex hardware/software systems, it may be necessary to use a group of suppliers, each with a different type of expertise.

For large systems, such as an air traffic management system, a group of suppliers may form a consortium to bid for a contract. The consortium should include all of the capabilities required for this type of system. For an ATC system, this would include computer hardware suppliers, software companies, peripheral suppliers, and suppliers of specialist equipment such as radar systems.

Customers do not usually wish to negotiate with multiple suppliers, so the contract is usually awarded to a principal contractor, who coordinates the project. The principal contractor coordinates the development of different subsystems by subcontractors. The subcontractors design and build parts of the system to a specification that is negotiated with the principal contractor and the customer. Once completed, the principal contractor integrates these components and delivers them to the customer.

Decisions made at the procurement stage of the systems engineering process are critical for later stages in that process. Poor procurement decisions often lead to problems such as late delivery of a system and development of systems that are unsuited to their operational environment. If the wrong system or the wrong supplier is chosen, then the technical processes of system and software engineering become more complex.

For example, I studied a system "failure" where a decision was made to choose an ERP system because this would "standardize" operations across the organization. These operations were very diverse, and it turned out there were good reasons for this. Standardization was practically impossible. The ERP system could not be adapted to cope with this diversity. It was ultimately abandoned after incurring costs of around £10 million.

Decisions and choices made during system procurement have a profound effect on the security and dependability of a system. For example, if a decision is made to procure an off-the-shelf system, then the organization has to accept that they have no influence over the security and dependability requirements of this system. System security depends on decisions made by system vendors. In addition, off-the-shelf systems may have known security weaknesses or may require complex configuration. Configuration errors, where entry points to the system are not properly secured, are a significant source of security problems.

On the other hand, a decision to procure a custom system means that a lot of effort must be devoted to understanding and defining security and dependability requirements. If a company has limited experience in this area, this is quite a difficult thing to do. If the

required level of dependability as well as acceptable system performance is to be achieved, then the development time may have to be extended and the budget increased.

Many bad procurement decisions stem from political rather than technical causes. Senior management may wish to have more control and so demand that a single system is used across an organization. Suppliers may be chosen because they have a long-standing relationship with a company rather than because they offer the best technology. Managers may wish to maintain compatibility with existing systems because they feel threatened by new technologies. As I discuss in Chapter 20, people who do not understand the required system are often responsible for procurement decisions. Engineering issues do not necessarily play a major part in their decision-making process.

19.4 System development

System development is a complex process in which the elements that are part of the system are developed or purchased and then integrated to create the final system. The system requirements are the bridge between the conceptual design and the development processes. During conceptual design, business and high-level functional and non-functional system requirements are defined. You can think of this as the start of development, hence the overlapping processes shown in Figure 19.1. Once contracts for the system elements have been agreed, more detailed requirements engineering takes place.

Figure 19.10 is a model of the systems development process. Systems engineering processes usually follow a "waterfall" process model similar to the one that I discussed in Chapter 2. Although the waterfall model is inappropriate for most types of software development, higher-level systems engineering processes are plan-driven processes that still follow this model.

Plan-driven processes are used in systems engineering because different elements of the system are independently developed. Different contractors are working concurrently on separate subsystems. Therefore, the interfaces to these elements have to be designed before development begins. For systems that include hardware and other equipment, changes during development can be very expensive or, sometimes, practically impossible. It is essential therefore, that the system requirements are fully understood before hardware development or building work begins.

One of the most confusing aspects of systems engineering is that companies use different terminology for each stage of the process. Sometimes, requirements engineering is part of the development process, and sometimes it is a separate activity. However, after conceptual design, there are seven fundamental development activities:

1. *Requirements engineering* is the process of refining, analyzing, and documenting the high-level and business requirements identified in the conceptual design. I have covered the most important requirements engineering activities in Chapter 4.

2. *Architectural design* overlaps significantly with the requirements engineering process. The process involves establishing the overall architecture of the system,

Figure 19.10 The systems development process

identifying the different system components, and understanding the relationships between them.

3. *Requirements partitioning* is concerned with deciding which subsystems (identified in the system architecture) are responsible for implementing the system requirements. Requirements may have to be allocated to hardware, software, or operational processes and prioritized for implementation. Ideally, you should allocate requirements to individual subsystems so that the implementation of a critical requirement does not need subsystem collaboration. However, this is not always possible. At this stage you also decide on the operational processes and on how these are used in the requirements implementation.

4. *Subsystem engineering* involves developing the software components of the system, configuring off-the-shelf hardware and software, designing, if necessary, special-purpose hardware, defining the operational processes for the system, and re-designing essential business processes.

5. *System integration* is the process of putting together system elements to create a new system. Only then do the emergent system properties become apparent.

6. *System testing* is an extended activity where the whole system is tested and problems are exposed. The subsystem engineering and system integration phases are reentered to repair these problems, tune the performance of the system, and implement new requirements. System testing may involve both testing by the system developer and acceptance/user testing by the organization that has procured the system.

7. *System deployment* is the process of making the system available to its users, transferring data from existing systems, and establishing communications with other systems in the environment. The process culminates with a "go live," after which users start to use the system to support their work.

Although the overall process is plan-driven, the processes of requirements development and system design are inextricably linked. The requirements and the high-level

Figure 19.11 Requirements and design spiral

design are developed concurrently. Constraints posed by existing systems may limit design choices, and these choices may be specified in the requirements. You may have to do some initial design to structure and organize the requirements engineering process. As the design process continues, you may discover problems with existing requirements and new requirements may emerge. Consequently, you can think of these linked processes as a spiral, as shown in Figure 19.11.

The spiral reflects the reality that requirements affect design decisions and vice versa, and so it makes sense to interleave these processes. Starting in the center, each round of the spiral may add detail to the requirements and the design. As subsystems are identified in the architecture, decisions are made on the responsibilities of these subsystems for providing the system requirements. Some rounds of the spiral may focus on requirements, others on design. Sometimes new knowledge collected during the requirements and design process means that the problem statement itself has to be changed.

For almost all systems, many possible designs meet the requirements. These cover a range of solutions that combine hardware, software, and human operations. The solution that you choose for further development may be the most appropriate technical solution that meets the requirements. However, wider organizational and political considerations may influence the choice of solution. For example, a government client may prefer to use national rather than foreign suppliers for its system, even if national products are technically inferior.

These influences usually take effect in the review and assessment phase of the spiral model where designs and requirements may be accepted or rejected. The process ends when a review decides that the requirements and high-level design are sufficiently detailed for subsystems to be specified and designed.

Subsystem engineering involves designing and building the system's hardware and software components. For some types of systems, such as spacecraft, all hardware and software components may be designed and built during the development process. However, in most systems, some components are bought rather than developed. It is usually much cheaper to buy existing products than to develop special-purpose components. However, if you buy large off-the-shelf systems, such as ERP systems, there is a significant cost in configuring these systems for use in their operational environment.

Subsystems are usually developed in parallel. When problems that cut across subsystem boundaries are encountered, a system modification request must be made. Where systems involve extensive hardware engineering, making modifications after manufacturing has started is usually very expensive. Often "workarounds" that compensate for the problem must be found. These workarounds usually involve software changes to implement new requirements.

During systems integration, you take the independently developed subsystems and put them together to make up a complete system. This integration can be achieved using a "big bang" approach, where all the subsystems are integrated at the same time. However, for technical and managerial reasons, an incremental integration process where subsystems are integrated one at a time is the best approach:

1. It is usually impossible to schedule the development of all the subsystems so that they are all finished at the same time.

2. Incremental integration reduces the cost of error location. If many subsystems are simultaneously integrated, an error that arises during testing may be in any of these subsystems. When a single subsystem is integrated with an already working system, errors that occur are probably in the newly integrated subsystem or in the interactions between the existing subsystems and the new subsystem.

As an increasing number of systems are built by integrating off-the-shelf hardware and software application systems, the distinction between implementation and integration is becoming blurred. In some cases, there is no need to develop new hardware or software. Essentially, systems integration is the implementation phase of the system.

During and after the integration process, the system is tested. This testing should focus on testing the interfaces between components and the behavior of the system as a whole. Inevitably, testing also reveals problems with individual subsystems that have to be repaired. Testing takes a long time, and a common problem in system development is that the testing team may run out of either budget or time. This problem can lead to the delivery of error-prone systems that need be repaired after they have been deployed.

Subsystem faults that are a consequence of invalid assumptions about other subsystems are often exposed during system integration. This may lead to disputes between the contractors responsible for implementing different subsystems. When problems are discovered in subsystem interaction, the contractors may argue about which subsystem is faulty. Negotiations on how to solve the problems can take weeks or months.

The final stage of the system development process is system delivery and deployment. The software is installed on the hardware and is readied for operation. This may

involve more system configuration to reflect the local environment where it is used, the transfer of data from existing systems, and the preparation of user documentation and training. At this stage, you may also have to reconfigure other systems in the environment to ensure that the new system interoperates with them.

Although system deployment is straightforward in principle, it is often more difficult than anticipated. The user environment may be different from that anticipated by the system developers. Adapting the system to make it work in an unexpected environment can be difficult. The existing system data may require extensive clean-up, and parts of it may involve more effort than expected. The interfaces to other systems may not be properly documented. You may find that the planned operational processes have to be changed because they are not compatible with the operational processes for other systems. User training is often difficult to arrange, with the consequence that, initially at least, users are unable to access the capabilities of the system. System deployment can therefore take much longer and cost much more than anticipated.

19.5 System operation and evolution

Operational processes are the processes that are involved in using the system as intended by its designers. For example, operators of an air traffic control system follow specific processes when aircraft enter and leave airspace, when they have to change height or speed, when an emergency occurs, and so on. For new systems, these operational processes have to be defined and documented during the system development process. Operators may have to be trained and other work processes adapted to make effective use of the new system. Undetected problems may arise at this stage because the system specification may contain errors or omissions. While the system may perform to specification, its functions may not meet the real operational needs. Consequently, the operators may not use the system as its designers intended.

Although the designers of operational processes may have based their process designs on extensive user studies, there is always a period of "domestication" (Stewart and Williams 2005) when users adapt to the new system and work out practical processes of how to use it. While user interface design is important, studies have shown that, given time, users can adapt to complex interfaces. As they become experienced, they prefer ways of using the system quickly rather than easily. This means that when designing systems, you should not simply cater for inexperienced users but you should design the user interface to be adaptable for experienced users.

Some people think that system operators are a source of problems in a system and that we should move toward automated systems where operator involvement is minimized. In my opinion, there are two problems with this approach:

1. It is likely to increase the technical complexity of the system because it has to be designed to cope with all anticipated failure modes. This increases the costs and

time required to build the system. Provision also has to be made to bring in people to deal with unanticipated failures.

2. People are adaptable and can cope with problems and unexpected situations. Thus, you do not have to anticipate everything that could possibly go wrong when you are specifying and designing the system.

People have a unique capability of being able to respond effectively to the unexpected, even when they have never had direct experience of these unexpected events or system states. Therefore, when things go wrong, the system operators can often recover the situation by finding workarounds and using the system in nonstandard ways. Operators also use their local knowledge to adapt and improve processes. Normally, the actual operational processes are different from those anticipated by the system designers.

Consequently, you should design operational processes to be flexible and adaptable. The operational processes should not be too constraining; they should not require operations to be done in a particular order; and the system software should not rely on a specific process being followed. Operators usually improve the process because they know what does and does not work in a real situation.

A problem that may only emerge after the system goes into operation is the operation of the new system alongside existing systems. There may be physical problems of incompatibility, or it may be difficult to transfer data from one system to another. More subtle problems might arise because different systems have different user interfaces. Introducing a new system may increase the operator error rate, as the operators use user interface commands for the wrong system.

19.5.1 System evolution

Large, complex systems usually have a long lifetime. Complex hardware/software systems may remain in use for more than 20 years, even though both the original hardware and software technologies used are obsolete. There are several reasons for this longevity, as shown in Figure 19.12.

Over their lifetime, large complex systems change and evolve to correct errors in the original system requirements and to implement new requirements that have emerged. The system's computers are likely to be replaced with new, faster machines. The organization that uses the system may reorganize itself and hence use the system in a different way. The external environment of the system may change, forcing changes to the system. Hence, evolution is a process that runs alongside normal system operational processes. System evolution involves reentering the development process to make changes and extensions to the system's hardware, software, and operational processes.

System evolution, like software evolution (discussed in Chapter 9), is inherently costly for several reasons:

1. Proposed changes have to be analyzed very carefully from a business and a technical perspective. Changes have to contribute to the goals of the system and should not simply be technically motivated.

Factor	Rationale
Investment cost	The costs of a systems engineering project may be tens or even hundreds of millions of dollars. These costs can only be justified if the system can deliver value to an organization for many years.
Loss of expertise	As businesses change and restructure to focus on their core activities, they often lose engineering expertise. This may mean that they lack the ability to specify the requirements for a new system.
Replacement cost	The cost of replacing a large system is very high. Replacing an existing system can be justified only if this leads to significant cost savings over the existing system.
Return on investment	If a fixed budget is available for systems engineering, spending on new systems in some other area of the business may lead to a higher return on investment than replacing an existing system.
Risks of change	Systems are an inherent part of business operations, and the risks of replacing existing systems with new systems cannot be justified. The danger with a new system is that things can go wrong in the hardware, software, and operational processes. The potential costs of these problems for the business may be so high that they cannot take the risk of system replacement.
System dependencies	Systems are interdependent and replacing one of these systems may lead to extensive changes in other systems.

Figure 19.12 Factors that influence system lifetimes

2. Because subsystems are never completely independent, changes to one subsystem may have side-effects that adversely affect the performance or behavior of other subsystems. Consequent changes to these subsystems may therefore be needed.
3. The reasons for original design decisions are often unrecorded. Those responsible for the system evolution have to work out why particular design decisions were made.
4. As systems age, their structure becomes corrupted by change, so the costs of making further changes increases.

Systems that have been in use for many years are often reliant on obsolete hardware and software technology. These "legacy systems" (discussed in Chapter 9) are sociotechnical computer-based systems that have been developed using technology that is now obsolete. However, they don't just include legacy hardware and software. They also rely on legacy processes and procedures—old ways of doing things that are difficult to change because they rely on legacy software. Changes to one part of the system inevitably involve changes to other components.

Changes made to a system during system evolution are often a source of problems and vulnerabilities. If the people implementing the changes are different from those who developed the system, they may be unaware that a design decision was taken for dependability and security reasons. Therefore, they may change the system and lose some safeguards that were deliberately implemented when the system was built. Furthermore, as testing is so expensive, complete retesting may be impossible after every system change. Consequently, testing may not discover the adverse side-effects of changes that introduce or expose faults in other system components.

KEY POINTS

- Systems engineering is concerned with all aspects of specifying, buying, designing, and testing complex sociotechnical systems.
- Sociotechnical systems include computer hardware, software, and people, and are situated within an organization. They are designed to support organizational or business goals and objectives.
- The emergent properties of a system are characteristics of the system as a whole rather than of its component parts. They include properties such as performance, reliability, usability, safety, and security.
- The fundamental systems engineering processes are conceptual systems design, system procurement, system development, and system operation.
- Conceptual systems design is a key activity where high-level system requirements and a vision of the operational system is developed.
- System procurement covers all of the activities involved in deciding what system to buy and who should supply that system. Different procurement processes are used for off-the-shelf application systems, configurable COTS systems, and custom systems.
- System development processes include requirements specification, design, construction, integration, and testing.
- When a system is put into use, the operational processes and the system itself inevitably change to reflect changes to the business requirements and the system's environment.

FURTHER READING

"Airport 95: Automated Baggage System." An excellent, readable case study of what can go wrong with a systems engineering project and how software tends to get the blame for wider systems failures. (*ACM Software Engineering Notes*, 21, March 1996). http://doi.acm.org/10.1145/227531.227544

"Fundamentals of Systems Engineering." This is the introductory chapter in NASA's systems engineering handbook. It presents an overview of the systems engineering process for space systems. Although these are mostly technical systems, there are sociotechnical issues to be considered. Dependability is obviously critically important. (In *NASA Systems Engineering Handbook*, NASA-SP 2007-6105, 2007). http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20080008301_2008008500.pdf

The LSCITS Socio-technical Systems Handbook. This handbook introduces sociotechnical systems in an accessible way and provides access to more detailed papers on sociotechnical topics. (Various authors, 2012). http://archive.cs.st-andrews.ac.uk/STSE-Handbook

Architecting systems: Concepts, Principles and Practice. This is a refreshingly different book on systems engineering that does not have the hardware focus of many "traditional" systems engineering books.

The author, who is an experienced systems engineer, draws on examples from a wide range of systems and recognizes the importance of sociotechnical as well as technical issues. (H. Sillitto, College Publications, 2014).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/systems-engineering/

EXERCISES

19.1. Give two examples of government functions that are supported by complex sociotechnical systems and explain why, in the foreseeable future, these functions cannot be completely automated.

19.2. Explain briefly why the involvement of a range of professional disciplines is essential in systems engineering.

19.3. Complex sociotechnical systems lead to three important characteristics. What are they? Explain each in brief.

19.4. What is a "wicked problem"? Explain why the development of a national medical records system should be considered a "wicked problem."

19.5. A multimedia virtual museum system offering virtual experiences of ancient Greece is to be developed for a consortium of European museums. The system should provide users with the facility to view 3-D models of ancient Greece through a standard web browser and should also support an immersive virtual reality experience. Develop a conceptual design for such a system, highlighting its key characteristics and essential high-level requirements.

19.6. Explain why you need to be flexible and adapt system requirements when procuring large off-the-shelf software systems, such as ERP systems. Search the web for discussions of the failures of such systems and explain, from a sociotechnical perspective, why these failures occurred. A possible starting point is: http://blog.360cloudsolutions.com/blog/bid/94028/Top-Six-ERP-Implementation-Failures

19.7. Why is system integration a particularly critical part of the systems development process? Suggest three sociotechnical issues that may cause difficulties in the system integration process.

19.8. Why is system evolution inherently costly?

19.9. What are the arguments for and against considering system engineering as a profession in its own right, like electrical engineering or software engineering?

19.10. You are an engineer involved in the development of a financial system. During installation, you discover that this system will make a significant number of people redundant. The people in the environment deny you access to essential information to complete the system installation. To what extent should you, as a systems engineer, become involved in this situation? Is it your professional responsibility to complete the installation as contracted? Should you simply abandon the work until the procuring organization has sorted out the problem?

REFERENCES

Baxter, G., and I. Sommerville. 2011. "Socio-Technical Systems: From Design Methods to Systems Engineering." *Interacting with Computers* 23 (1): 4–17. doi:10.1016/j.intcom.2010.07.003.

Checkland, P. 1981. *Systems Thinking, Systems Practice*. Chichester, UK: John Wiley & Sons.

Fairley, R. E., R. H. Thayer, and P. Bjorke. 1994. "The Concept of Operations: The Bridge from Operational Requirements to Technical Specifications." In *1st Int. Conf. on Requirements Engineering,* 40–7. Colorado Springs, CO. doi:10.1109/ICRE.1994.292405.

IEEE. 2007. "IEEE Guide for Information Technology. System Definition—Concept of Operations (ConOps) Document." *Electronics*. Vol. 1998. doi:10.1109/IEEESTD.1998.89424. http://ieeexplore.ieee.org/servlet/opac?punumber=6166

Mostashari, A., S. A. McComb, D. M. Kennedy, R. Cloutier, and P. Korfiatis. 2012. "Developing a Stakeholder-Assisted Agile CONOPS Development Process." *Systems Engineering* 15 (1): 1–13. doi:10.1002/sys.20190.

Rittel, H., and M. Webber. 1973. "Dilemmas in a General Theory of Planning." *Policy Sciences* 4: 155–169. doi:10.1007/BF01405730.

Stevens, R., P. Brook, K. Jackson, and S. Arnold. 1998. *Systems Engineering: Coping with Complexity*. London: Prentice-Hall.

Stewart, J., and R. Williams. 2005. "The Wrong Trousers? Beyond the Design Fallacy: Social Learning and the User." In *User Involvement in Innovation Processes. Strategies and Limitations from a Socio-Technical Perspective,* edited by H. Rohrache, 39–71. Berlin: Profil-Verlag.

Thayer, R. H. 2002. "Software System Engineering: A Tutorial." *IEEE Computer* 35 (4): 68–73. doi:10.1109/MC.2002.993773.

White, S., M. Alford, J. Holtzman, S. Kuehl, B. McCay, D. Oliver, D. Owens, C. Tully, and A. Willey. 1993. "Systems Engineering of Computer-Based Systems." *IEEE Computer* 26 (11): 54–65. doi:10.1109/ECBS.1994.331687.

20

Systems of systems

Objectives

The objectives of this chapter are to introduce the idea of a system of systems and to discuss the challenges of building complex systems of software systems. When you have read this chapter, you will:

- understand what is meant by a system of systems and how it differs from an individual system;
- understand systems of systems classification and the differences between different types of systems of systems;
- understand why conventional methods of software engineering that are based on reductionism are inadequate for developing systems of systems;
- have been introduced to the systems of systems engineering process and architectural patterns for systems of systems.

Contents

20.1 System complexity
20.2 Systems of systems classification
20.3 Reductionism and complex systems
20.4 Systems of systems engineering
20.5 Systems of systems architecture

We need software engineering because we create large and complex software systems. The discipline emerged in the 1960s because the first attempts to build large software systems mostly went wrong. Creating software was much more expensive than expected, took longer than planned, and the software itself was often unreliable. To address these problems, we have developed a range of software engineering techniques and technologies, which have been remarkably successful. We can now build systems that are much larger, more complex, much more reliable, and more effective than the software systems of the 1970s.

However, we have not "solved" the problems of large system engineering. Software project failures are still common. For example, there have been serious problems and delays in the implementation of government health care systems in both the United States and the UK. The root cause of these problems is, as it was in the 1960s, that we are trying to build systems that are larger and more complex than before. We are attempting to build these "mega-systems" using methods and technology that were never designed for this purpose. As I discuss later in the chapter, I believe that current software engineering technology cannot scale up to cope with the complexity that is inherent in many of the systems now being proposed.

The increase in size of software systems since the introduction of software engineering has been remarkable. Today's large systems may be a hundred or even a thousand times larger than the "large" systems of the 1960s. Northrop and her colleagues (Northrop et al. 2006) suggested in 2006 that we would shortly see the development of systems with a billion lines of code. Almost 10 years after this prediction, I suspect such systems are already in use.

Of course, we do not start with nothing and then write a billion lines of code. As I discussed in Chapter 15, the real success story of software engineering has been software reuse. It is only because we have developed ways of reusing software across applications and systems that large-scale development is possible. Very large-scale systems now and in the future will be built by integrating existing systems from different providers to create systems of systems (SoS).

What do we mean when we talk about a system of systems? As Hitchens says (Hitchins 2009), from a general systems perspective, there is no difference between a system and a system of systems. Both have emergent properties and can be composed from subsystems. However, from a software engineering perspective, I think there is a useful distinction between these terms. This distinction is sociotechnical rather than technical:

> *A system of systems is a system that contains two or more independently managed elements.*

This means that there is no single manager for all of the parts of the system of systems and that different parts of a system are subject to different management and control policies and rules. As we shall see, distributed management and control has a profound effect on the overall complexity of the system.

This definition of systems of systems says nothing about the size of systems of systems. A relatively small system that includes services from different providers is

a system of systems. Some of the problems of SoS engineering apply to such small systems, but the real challenges emerge when the constituent systems are themselves large-scale systems.

Much of the work in the area of systems of systems has come from the defense community. As the capability of software systems increased in the late 20th century, it became possible to coordinate and control previously independent military systems, such as naval and ground-based air and ship defense systems. The system might include tens or hundreds of separate elements, with software systems keeping track of these elements and providing controllers with information that allows them to be deployed most effectively.

This type of system of systems is outside the scope of a software engineering book. Instead, I focus here on systems of systems where the system elements are software systems rather than hardware such as aircraft, military vehicles, or radars. Systems of software systems are created by integrating separate software systems, and, at the time of writing, most software SoS include a relatively small number of separate systems. Each constituent system is usually a complex system in its own right. However, it is predicted that, over the next few years, the size of software SoS is likely to grow significantly as more and more systems are integrated to make use of the capabilities that they offer.

Examples of systems of systems of software systems are:

1. A cloud management system that handles local private cloud management and management of servers on public clouds such as Amazon and Microsoft.
2. An online banking system that handles loan requests and that connects to a credit reference system provided by credit reference agencies to check the credit of applicants.
3. An emergency information system that integrates information from police, ambulance, fire, and coast guard services about the assets available to deal with civil emergencies such as flooding and large-scale accidents.
4. The digital learning environment (iLearn) that I introduced in Chapter 1. This system provides a range of learning support by integrating separate software systems such as Microsoft Office 365, virtual learning environments such as Moodle, simulation modeling tools, and content such as newspaper archives.

Maier (Maier 1998) identified five essential characteristics of systems of systems:

1. *Operational independence of elements* Parts of the system are not simply components but can operate as useful systems in their own right. The systems within the SoS evolve independently.
2. *Managerial independence of elements* Parts of the system are "owned" and managed by different organizations or by different parts of a larger organization. Therefore different rules and policies apply to the management and evolution of

these systems. As I have suggested, this is the key factor that distinguishes a system of systems from a system.

3. *Evolutionary development* SoS are not developed in a single project but evolve over time from their constituent systems.

4. *Emergence* SoS have emergent characteristics that only become apparent after the SoS has been created. Of course, as I have discussed in Chapter 19, emergence is a characteristic of all systems, but it is particularly important in SoS.

5. *Geographical distribution of elements* The elements of a SoS are often geographically distributed across different organizations. This is important technically because it means that an externally-managed network is an integral part of the SoS. It is also important managerially as it increases the difficulties of communication between those involved in making system management decisions and adds to the difficulties of maintaining system security.[†]

I would like to add two further characteristics to Maier's list that are particularly relevant to systems of software systems:

1. *Data intensive* A software SoS typically relies on and manages a very large volume of data. In terms of size, this may be tens or even hundreds of times larger than the code of the constituent systems itself.

2. *Heterogeneity* The different systems in a software SoS are unlikely to have been developed using the same programming languages and design methods. This is a consequence of the very rapid pace of evolution of software technologies. Companies frequently update their development methods and tools as new, improved versions become available. In a 20-year lifetime of a large SoS, technologies may change four or five times.

As I discuss in Section 20.1, these characteristics mean that SoS can be much more complex than systems with a single owner and manager. I believe that our current software engineering methods and techniques cannot scale to cope with this complexity. Consequently, problems with the very large and complex systems that we are now developing are inevitable. We need a completely new set of abstractions, methods, and technologies for software systems of systems engineering.

This need has been recognized independently by a number of different authorities. In the UK, a report published in 2004 (Royal Academy of Engineering 2004) led to the establishment of a national research and training initiative in large-scale complex IT systems (Sommerville et al. 2012). In the United States, the Software Engineering Institute reported on Ultra-Large Scale Systems in 2006 (Northrop et al. 2006). From the systems engineering community, Stevens (Stevens 2010) discusses the problems of constructing "mega-systems" in transport, health care, and defense.

[†]Maier, M. W. 1998. "Architecting Principles for Systems-of-Systems." Systems Engineering 1 (4): 267–284. doi:10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D.

Figure 20.1 Simple and complex systems

System (a)

System (b)

20.1 System complexity

I suggested in the introduction that the engineering problems that arise when constructing systems of software systems are due to the inherent complexity of these systems. In this section, I explain the basis of system complexity and discuss the different types of complexity that arise in software SoS.

All systems are composed of parts (elements) with relationships between these elements of the system. For example, the parts of a program may be objects, and the parts of each object may be constants, variables, and methods. Examples of relationships include "calls" (method A calls method B), "inherits-from" (object X inherits the methods and attributes of object Y), and "part of" (method A is part of object X).

The complexity of any system depends on the number and types of relationships between system elements. Figure 20.1 shows examples of two systems. System (a) is a relatively simple system with only a small number of relationships between its elements. By contrast, System (b), with the same number of elements, is a more complex system because it has many more element–element relationships.

The type of relationship also influences the overall complexity of a system. Static relationships are relationships that are planned and analyzable from static depictions of the system. Therefore, the "uses" relationship in a software system is a static relationship. From either the software source code or a UML model of a system, you can work out how any one software component uses other components.

Dynamic relationships are relationships that exist in an executing system. The "calls" relationship is a dynamic relationship because, in any system with if-statements, you cannot tell whether or not one method will call another method. It depends on the runtime inputs to the system. Dynamic relationships are more complex to analyze as you need to know the system inputs and data used as well as the source code of the system.

As well as system complexity, we also have to consider the complexity of the processes used to develop and maintain the system once it has gone into use. Figure 20.2 illustrates these processes and their relationship with the developed system.

Figure 20.2 Production and management processes

As systems grow in size, they need more complex production and management processes. Complex processes are themselves complex systems. They are difficult to understand and may have undesirable emergent properties. They are more time consuming than simpler processes, and they require more documentation and coordination between the people and the organizations involved in the system development. The complexity of the production process is one of the main reasons why projects go wrong, with software delivered late and overbudget. Therefore, large systems are always at risk of cost and time overruns.

Complexity is important for software engineering because it is the main influence on the understandability and the changeability of a system. The more complex a system, the more difficult it is to understand and analyze. Given that complexity is a function of the number of relationships between elements of a system, it is inevitable that large systems are more complex than small systems. As complexity increases, there are more and more relationships between elements of the system and an increased likelihood that changing one part of a system will have undesirable effects elsewhere.

Several different types of complexity are relevant to sociotechnical systems:

1. The *technical complexity* of the system is derived from the relationships between the different components of the system itself.

2. The *managerial complexity* of the system is derived from the complexity of the relationships between the system and its managers (i.e., what can managers change in the system) and the relationships between the managers of different parts of the system.

3. The *governance complexity* of a system depends on the relationships between the laws, regulations, and policies that affect the system and the relationships between the decision-making processes in the organizations responsible for the system. As different parts of the system may be in different organizations and in different countries, different laws, rules, and policies may apply to each system within the SoS.

Governance and managerial complexity are related, but they are not the same thing. Managerial complexity is an operational issue—what can and can't actually be done with the system. Governance complexity is associated with the higher level of decision-making processes in organizations that affect the system. These decision-making processes are constrained by national and international laws and regulations.

For example, say a company decides to allow its staff to access its systems using their own mobile devices rather than company-issued laptops. The decision to allow this is a governance decision because it changes the policy of the company. As a result of this decision, management of the system becomes more complex as managers have to ensure that the mobile devices are configured properly so that company data is secure. The technical complexity of the system also increases as there is no longer a single implementation platform. Software may have to be modified to work on laptops, tablets and phones.

As well as technical complexity, the characteristics of systems of systems may also lead to significantly increased managerial and governance complexity. Figure 20.3 summarizes how the different SoS characteristics primarily contribute to different types of complexity:

1. *Operational independence* The constituent systems in the SoS are subject to different policies and rules (governance complexity) and ways of managing the system (managerial complexity).

2. *Managerial independence* The constituent systems in the SoS are managed by different people in different ways. They have to coordinate to ensure that management changes are consistent (managerial complexity). Special software may be needed to support consistent management and evolution (technical complexity).

3. Evolutionary development contributes to the technical complexity of a SoS because different parts of the system are likely to be built using different technologies.

4. Emergence is a consequence of complexity. The more complex a system, the more likely it is that it will have undesirable emergent properties. These properties increase the technical complexity of the system as software has to be developed or changed to compensate for them.

5. Geographical distribution increases the technical, managerial, and governance complexity in a SoS. Technical complexity is increased because software is required to coordinate and synchronize remote systems; managerial complexity is increased because it is more difficult for managers in different countries to coordinate their actions; governance complexity is increased because different

SoS characteristic	Technical complexity	Managerial complexity	Governance complexity
Operational independence		X	X
Managerial independence	X	X	
Evolutionary development	X		
Emergence	X		
Geographical distribution	X	X	X
Data-intensive	X		X
Heterogeneity	X		

Figure 20.3 SoS characteristics and system complexity

parts of the systems may be located in different jurisdictions and so are subject to different laws and regulations.

6. Data-intensive systems are technically complex because of the relationships between the data items. The technical complexity is also likely to be increased to cope with data errors and incompleteness. Governance complexity may be increased because of different laws governing the use of data.

7. The heterogeneity of a system contributes to its technical complexity because of the difficulties of ensuring that different technologies used in different parts of the system are compatible.

Large-scale systems of systems are now unimaginably complex entities that cannot be understood or analyzed as a whole. As I discuss in Section 20.3, the large number of interactions between the parts and the dynamic nature of these interactions means that conventional engineering approaches do not work well for complex systems. It is complexity that is the root cause of problems in projects to develop large software-intensive systems, not poor management or technical failings.

20.2 Systems of systems classification

Earlier, I suggested that the distinguishing feature of a system of systems was that two or more of its elements were independently managed. Different people with different priorities have the authority to take day-to-day operational decisions about changes to the system. As their work is not necessarily aligned, conflicts can arise that require a significant amount of time and effort to resolve. Systems of systems, therefore, always have some degree of managerial complexity.

However, this broad definition of SoS covers a very wide range of system types. It includes systems that are owned by a single organization but are managed by different

parts of that organization. It also includes systems whose constituent systems are owned and managed by different organizations that may, at times, compete with each other. Maier (Maier 1998) devised a classification scheme for SoS based on their governance and management complexity:

1. *Directed systems.* Directed SoS are owned by a single organization and are developed by integrating systems that are also owned by that organization. The system elements may be independently managed by parts of the organization. However, there is an ultimate governing body within the organization that can set priorities for system management. It can resolve disputes between the managers of different elements of the system. Directed systems therefore have some managerial complexity but no governance complexity. A military command-and-control system that integrates information from airborne and ground-based systems is an example of a directed SoS.

2. *Collaborative systems.* Collaborative SoS are systems with no central authority to set management priorities and resolve disputes. Typically, elements of the system are owned and governed by different organizations. However, all of the organizations involved recognize the mutual benefits of joint governance of the system. They therefore usually set up a voluntary governance body that makes decisions about the system. Collaborative systems have both managerial complexity and a limited degree of governance complexity. An integrated public transport information system is an example of a collaborative system of systems. Bus, rail, and air transport providers agree to link their systems to provide passengers with up-to-date information.

3. *Virtual systems.* Virtual systems have no central governance, and the participants may not agree on the overall purpose of the system. Participant systems may enter or leave the SoS. Interoperability is not guaranteed but depends on published interfaces that may change. These systems have a very high degree of both managerial and governance complexity. An example of a virtual SoS is an automated high-speed algorithmic trading system. These systems from different companies automatically buy and sell stock from each other, with trades taking place in fractions of a second.

Unfortunately, I think that the names that Maier has used do not really reflect the distinctions between these different types of systems. As Maier himself says, there is always some collaboration in the management of the system elements. So, "collaborative systems" is not really a good name. The term *directed systems* implies top-down authority. However, even within a single organization, the need to maintain good working relationships between the people involved means that governance is agreed to rather than imposed.

In "virtual" SoS, there may be no formal mechanisms for collaboration, but the system has some mutual benefit for all participants. Therefore, they are likely to collaborate informally to ensure that the system can continue to operate. Furthermore, Maier's use of the term *virtual* could be confusing because "virtual" has now come to mean "implemented by software," as in virtual machines and virtual reality.

Figure 20.4 SoS collaboration

Figure 20.4 illustrates the collaboration in these different types of system. Rather than use Maier's names, I have used what I hope are more descriptive terms:

1. *Organizational systems of systems* are SoS where the governance and management of the system lies within the same organization or company. These correspond to Maier's "directed SoS." Collaboration between system owners is managed by the organization. The SoS may be geographically distributed, with different parts of the system subject to different national laws and regulations. In Figure 20.4, Systems 1, 2, and 3 are independently managed, but the governance of these systems is centralized.

2. *Federated systems* are SoS where the governance of the SoS depends on a voluntary participative body in which all of the system owners are represented. In Figure 20.4, this is shown by the owners of Systems 1, 2, and 3 participating in a single governance body. The system owners agree to collaborate and believe that decisions made by the governance body are binding. They implement these decisions in their individual management policies, although implementations may differ because of national laws, regulations, and culture.

3. *System of system coalitions* are SoS with no formal governance mechanisms but where the organizations involved informally collaborate and manage their own systems to maintain the system as a whole. For example, if one system provides a data feed to others, the managers of that system will not change the format of the data without notice. Figure 20.4 shows that there is no governance at the organizational level but that informal collaboration exists at the management level.

This governance-based classification scheme provides a means of identifying the governance requirements for a SoS. By classifying a system according to this model, you can check if the appropriate governance structures exist and if these are the ones you really need. Setting up these structures across organizations is a political process and inevitably takes a long time. It is therefore helpful to understand the governance

problem early in the process and take actions to ensure that appropriate governance is in place. It may be the case that you need to adopt a governance model that moves a system from one class to another. Moving the governance model to the left in Figure 20.4 usually reduces complexity.

As I have suggested, the school digital learning environment (iLearn) is a system of systems. As well as the digital learning system itself, it is connected to school administration systems and to network management systems. These network management systems are used for Internet filtering, which stops students from accessing undesirable material on the Internet.

iLearn is a relatively simple technical system, but it has a high level of governance complexity. This complexity arises because of the way that education is funded and managed. In many countries pre-university education is funded and organized at a local level rather than at a national level. States, cities, or counties are responsible for schools in their area and have autonomy in deciding school funding and policies. Each local authority maintains its own school administration system and network management system.

In Scotland, there are 32 local authorities with responsibility for education in their area. School administration is outsourced to one of three providers and iLearn must connect to their systems. However, each local authority has its own network management policies with separate network management systems involved.

The development of a digital learning system is a national initiative, but to create a digital learning environment, it has to be integrated with network management and school administration systems. It is therefore a system of systems with administration and network management systems, as well as the systems within iLearn such as Office 365 and Wordpress. There is no common governance process across authorities, so, according to the classification scheme, this is a coalition of systems. In practice, this means that it cannot be guaranteed that students in different places can access the same tools and content, because of different Internet filtering policies.

When we produced the conceptual model for the system, we made a strong recommendation that common policies should be established across local authorities on administrative information provision and Internet filtering. In essence, we suggested that the system should be a federated system rather than a coalition of systems. This suggestion requires a new governance body to be established to agree on common policies and standards for the system.

20.3 Reductionism and complex systems

I have already suggested that our current software engineering methods and technologies cannot cope with the complexity that is inherent in modern systems of systems. Of course, this idea is not new: Progress in all engineering disciplines has always been driven by challenging and difficult problems. New methods and tools are developed in response to failures and difficulties with existing approaches.

In software engineering, we have seen the incredibly rapid development of the discipline to help manage the increasing size and complexity of software systems. This effort has been very successful indeed. We can now build systems that are orders of magnitude larger and more complex than those of the 1960s and 1970s.

As with other engineering disciplines, the approach that has been the basis of complexity management in software engineering is called *reductionism*. Reductionism is a philosophical position based on the assumptions that any system is made up of parts or subsystems. It assumes that the behavior and properties of the system as a whole can be understood and predicted by understanding the individual parts and the relationships between these parts. Therefore, to design a system, the parts making up that system are identified, constructed separately, and then assembled into the complete system. Systems can be thought of as hierarchies, with the important relationships between parent and child nodes in the hierarchy.

Reductionism has been and continues to be the fundamental underpinning approach to all kinds of engineering. We can identify common abstractions across the same types of system and design and build these separately. They can then be integrated to create the required system. For example, the abstractions in an automobile might be a body shell, a drive train, an engine, a fuel system, and so on. There are a relatively small number of relationships between these abstractions, so it is possible to specify interfaces and design and build each part of the system separately.

The same reductionist approach has been the basis of software engineering for almost 50 years. Top-down design, where you start with a very high-level model of a system and break this down to its components is a reductionist approach. This is the basis of all software design methods, such as object-oriented design. Programming languages include abstractions, such as procedures and objects that directly reflect reductionist system decomposition.

Agile methods, although they may appear quite different from top-down systems design, are also reductionist. They rely on being able to decompose a system into parts, implement these parts separately, and then integrate these to create the system. The only real difference between agile methods and top-down design is that the system is decomposed into components incrementally rather than all at once.

Reductionist methods are most successful when there are relatively few relationships or interactions between the parts of a system and it is possible to model these relationships in a scientific way. This is generally true for mechanical and electrical systems where there are physical linkages between the system components. It is less true for electronic systems and certainly not the case for software systems, where there may be many more static and dynamic relationships between system components.

The distinctions between software and hardware components was recognized in the 1970s. Design methods emphasized the importance of limiting and controlling the relationships between the parts of a system. These methods suggested that components should be tightly integrated with loose coupling between these components. Tight integration meant that most of the relationships were internal to a component, and loose coupling meant that there were relatively few component–component

Figure 20.5 Reductionist assumptions and complex system reality

relationships. The need for tight integration (data and operations) and loose coupling was the driver for the development of object-oriented software engineering.

Unfortunately, controlling the number and types of relationship is practically impossible in large systems, especially systems of systems. Reductionism does not work well when there are many relationships in a system and when these relationships are difficult to understand and analyze. Therefore, any type of large system development is likely to run into difficulties.

The reasons for these potential difficulties are that the fundamental assumptions inherent to reductionism are inapplicable for large and complex systems (Sommerville et al. 2012). These assumptions are shown in Figure 20.5 and apply in three areas:

1. *System ownership and control* Reductionism assumes that there is a controlling authority for a system that can resolve disputes and make high-level technical decisions that will apply across the system. As we have seen, because there are multiple bodies involved in their governance, this is simply not true for systems of systems.

2. *Rational decision making* Reductionism assumes that interactions between components can be objectively assessed by, for example, mathematical modeling. These assessments are the driver for system decision making. Therefore, if one particular design of a vehicle, say, offers the best fuel economy without a reduction in power, then a reductionist approach assumes that this will be the design chosen.

3. *Defined system boundaries* Reductionism assumes that the boundaries of a system can be agreed to and defined. This is often straightforward: There may be a physical shell defining the system as in a car, a bridge has to cross a given stretch of water, and so on. Complex systems are often developed to address wicked problems (Rittel and Webber 1973). For such problems, deciding on what is part of the system and what is outside it is usually a subjective judgment, with frequent disagreements between the stakeholders involved.

These reductionist assumptions break down for all complex systems, but when these systems are software-intensive, the difficulties are compounded:

1. Relationships in software systems are not governed by physical laws. We cannot produce mathematical models of software systems that will predict their behavior and attributes. We therefore have no scientific basis for decision making. Political factors are usually the driver of decision making for large and complex software systems.
2. Software has no physical limitations; hence there are no limits on where the boundaries of a system should be drawn. Different stakeholders will argue for the boundaries to be placed in such a way that is best for them. Furthermore, it is much easier to change software requirements than hardware requirements. The boundaries and the scope of a system are likely to change during its development.
3. Linking software systems from different owners is relatively easy; hence we are more likely to try and create a SoS where there is no single governing body. The management and evolution of the different systems involved cannot be completely controlled.

For these reasons, I believe that the problems and difficulties that are commonplace in large software systems engineering are inevitable. Failures of large government projects such as the health automation projects in the UK and the United States are a consequence of complexity rather than technical or project management failures.

Reductionist approaches such as object-oriented development have been very successful in improving our ability to engineer many types of software system. They will continue to be useful and effective in developing small and medium-sized systems whose complexity can be controlled and which may be parts of a software SoS. However, because of the fundamental assumptions underlying reductionism, "improving" these methods will not lead to an improvement in our ability to engineer complex systems of systems. Rather, we need new abstractions, methods, and tools that recognize the technical, human, social, and political complexities of SoS engineering. I believe that these new methods will be probabilistic and statistical and that tools will rely on system simulation to support decision making. Developing these new approaches is a major challenge for software and systems engineering in the 21st century.

20.4 Systems of systems engineering

Systems of systems engineering is the process of integrating existing systems to create new functionality and capabilities. Systems of systems are not designed in a top-down way. Rather, they are created when an organization recognizes that they can add value to existing systems by integrating these into a SoS. For example, a city government might wish to reduce air pollution at particular hot-spots in the city. To do so, it might integrate its traffic management system with a national real-time pollution monitoring systems. This then allows for the traffic management system to alter its strategy to reduce pollution by changing traffic light sequences, speed limits and so on.

Figure 20.6 An SoS engineering process

The problems of software SoS engineering have much in common with the problems of integrating large-scale application systems that I discussed in Chapter 15 (Boehm and Abts 1999). To recap, these were:

1. Lack of control over system functionality and performance.
2. Differing and incompatible assumptions made by the developers of the different systems.
3. Different evolution strategies and timetables for the different systems.
4. Lack of support from system owners when problems arise.

Much of the effort in building systems of software systems comes from addressing these problems. It involves deciding on the system architecture, developing software interfaces that reconcile differences between the participating systems, and making the system resilient to unforeseen changes that may occur.

Software systems of systems are large and complex entities, and the processes used for their development vary widely depending on the type of systems involved, the application domain, and the needs of the organizations involved in developing the SoS. However, as shown in Figure 20.6, five general activities are involved in SoS development processes:

1. *Conceptual design* I introduced the idea of conceptual design in Chapter 19, which covers systems engineering. Conceptual design is the activity of creating a high-level vision for a system, defining essential requirements, and identifying constraints on the overall system. In SoS engineering, an important input to the conceptual design process is knowledge of the existing systems that may participate in the SoS.
2. *System selection* During this activity, a set of systems for inclusion in the SoS is chosen. This process is comparable to the process of choosing application

systems for reuse, covered in Chapter 15. You need to assess and evaluate existing systems to choose the capabilities that you need. When you are selecting application systems, the selection criteria are largely commercial; that is, which systems offer the most suitable functionality at a price you are prepared to pay?

However, political imperatives and issues of system governance and management are often the key factors that influence what systems are included in a SoS. For example, some systems may be excluded from consideration because an organization does not wish to collaborate with a competitor. In other cases, organizations that are contributing to a federation of systems may have systems in place and insist that these are used, even though they are not necessarily the best systems.

3. *Architectural design* In parallel with system selection, an overall architecture for the SoS has to be developed. Architectural design is a major topic in its own right that I cover in Section 20.5.
4. *Interface development* The different systems involved in a SoS usually have incompatible interfaces. Therefore, a major part of the software engineering effort in developing a SoS is to develop interfaces so that constituent systems can interoperate. This may also involve the development of a unified user interface so that SoS operators do not have to deal with multiple user interfaces as they use the different systems in the SoS.
5. *Integration and deployment* This stage involves making the different systems involved in the SoS work together and interoperate through the developed interfaces. System deployment means putting the system into place in the organizations concerned and making it operational.

In parallel with these technical activities, there needs to be a high-level activity concerned with establishing policies for the governance of the system of systems and defining management guidelines to implement these policies. Where there are several organizations involved, this process can be prolonged and difficult. It may involve organizations changing their own policies and processes. It is therefore important to start governance discussions at an early stage in the SoS development process.

20.4.1 Interface development

The constituent systems in a SoS are usually developed independently for some specific purpose. Their user interface is tailored to that original purpose. These systems may or may not have application programming interfaces (APIs) that allow other systems to interface directly to them. Therefore, when these systems are integrated into a SoS, software interfaces have to be developed, which allows the constituent systems in the SoS to interoperate.

In general, the aim in SoS development is for systems to be able to communicate directly with each other without user intervention. If these systems already offer a service-based interface, as discussed in Chapter 18, then this communication can be implemented using this approach. Interface development involves describing how to

Figure 20.7 Systems with service interfaces

use the interfaces to access the functionality of each system. The systems involved can communicate directly with each other. System coalitions, where all of the systems involved are peers, are likely to use this type of direct interaction as it does not require prearranged agreements on system communication protocols.

More commonly, however, the constituent systems in a SoS either have their own specialized API or only allow their functionality to be accessed through their user interfaces. You therefore have to develop software that reconciles the differences between these interfaces. It is best to implement these interfaces as service-based interfaces, as shown in Figure 20.7 (Sillitto 2010).

To develop service-based interfaces, you have to examine the functionality of existing systems and define a set of services to reflect that functionality. The interface then provides these services. The services are implemented either by calls to the underlying system API or by mimicking user interaction with the system. One of the systems in the SoS is usually a principal or coordinating system that manages the interactions between the constituent systems. The principal system acts as a service broker, directing service calls between the different systems in the SoS. Each system therefore does not need to know which other system is providing a called service.

User interfaces for each system in a SoS are likely to be different. The principal system must have some overall user interfaces that handle user authentication and provide access to the features of the underlying system. However, it is usually expensive and time consuming to implement a unified user interface to replace the individual interfaces of the underlying systems.

A unified user interface (UI) makes it easier for new users to learn to use the SoS and reduces the likelihood of user error. However, whether or not unified UI development is cost-effective depends on a number of factors:

1. *The interaction assumptions of the systems in the SoS* Some systems may have a process-driven model of interaction where the system controls the interface and prompts the user for inputs. Others may give control to the user, so that the user chooses the sequence of interactions with the system. It is practically impossible to unify different interaction models.

2. *The mode of use of the SoS* In many cases, SoS are used in such a way that most of the interactions of users at a site are with one of the constituent systems. They use other systems only when additional information is required. For example, air traffic controllers may normally use a radar system for flight information and only access a flight plan database when additional information is required. A unified interface is a bad idea in these situations because it would slow down interaction with the most commonly used system. However, if the operators interact with all of the constituent systems, then a unified UI may be the best way forward.

3. *The "openness" of the SoS* If the SoS is open, so that new systems may be added to it when it is in use, then unified UI development is impractical. It is impossible to anticipate what the UI of new systems will be. Openness also applies to the organizations using the SoS. If new organizations can become involved, then they may have existing equipment and their own preferences for user interaction. They may therefore prefer not to have a unified UI.

In practice, the limiting factor in UI unification is likely to be the budget and time available for UI development. UI development is one of the most expensive systems engineering activities. In many cases, there is simply not enough project budget available to pay for the creation of a unified SoS user interface.

20.4.2 Integration and deployment

System integration and deployment are usually separate activities. A system is integrated from its components by an integration and testing team, validated, and then released for deployment. The components are managed so that changes are controlled and the integration team can be confident that the required version is included in the system. However, for SoS, such an approach may not be possible. Some of the component systems may already be deployed and in use, and the integration team cannot control changes to these systems.

For SoS, therefore, it makes sense to consider integration and deployment to be part of the same process. This approach reflects one of the design guidelines that I discuss in the following section, which is that an incomplete system of systems should be usable and provide useful functionality. The integration process should begin with systems that are already deployed, with new systems added to the SoS to provide coherent additions to the functionality of the overall system.

It often makes sense to plan the deployment of the SoS to reflect this, so that SoS deployment takes place in a number of stages. For example, Figure 20.8 illustrates a three-stage deployment process for the iLearn digital learning environment:

1. The initial deployment provides authentication, basic learning functionality, and integration with school administration systems.

2. Stage 2 of the deployment adds an integrated storage system and a set of more specialized tools to support subject-specific learning. These tools might include

Figure 20.8 Release sequence For the iLearn SoS

archives for history, simulation systems for science, and programming environments for computing.

3. Stage 3 adds features for user configuration and the ability of users to add new systems to the iLearn environment. This stage allows different versions of the system to be created for different age groups, further specialized tools, and alternatives to the standard tools to be included.

As in any large systems engineering project, the most time-consuming and expensive part of system integration is system testing. Testing systems of systems is difficult and expensive for three reasons:

1. There may not be a detailed requirements specification that can be used as a basis for system testing. It may not be cost-effective to develop a SoS requirements document because the details of the system functionality are defined by the systems that are included.
2. The constituent systems may change in the course of the testing process, so tests may not be repeatable.
3. If problems are discovered, it may not be possible to fix the problems by requiring one or more of the constituent systems to be changed. Rather, some intermediate software may have to be introduced to solve the problem.

To help address some of these problems, I believe that SoS testing should take on board some of the testing techniques developed in agile methods:

1. Agile methods do not rely on having a complete system specification for system acceptance testing. Rather, stakeholders are closely engaged with the testing process

and have the authority to decide when the overall system is acceptable. For SoS, a range of stakeholders should be involved in the testing process if possible, and they can comment on whether or not the system is ready for deployment.

2. Agile methods make extensive use of automated testing. This makes it much easier to rerun tests to discover if unexpected system changes have caused problems for the SoS as a whole.

Depending on the type of system, you may have to plan the installation of equipment and user training as part of the deployment process. If the system is being installed in a new environment, equipment installation is straightforward. However, if it is intended to replace an existing system, there may be problems in installing new equipment if it is not compatible with the equipment that is in use. There may not be the physical space for the new equipment to be installed alongside the working system. There may be insufficient electrical power, or users may not have time to be involved because they are busy using the current system. These nontechnical issues can delay the deployment process and slow down the adoption and use of the SoS.

20.5 Systems of systems architecture

Perhaps the most crucial activity of the systems of systems engineering process is architectural design. Architectural design involves selecting the systems to be included in the SoS, assessing how these systems will interoperate, and designing mechanisms that facilitate interaction. Key decisions on data management, redundancy, and communications are made. In essence, the SoS architect is responsible for realizing the vision set out in the conceptual design of the system. For organizational and federated systems, in particular, decisions made at this stage are crucial to the performance, resilience, and maintainability of the system of systems.

Maier (Maier 1998) discusses four general principles for the architecting of complex systems of systems:

1. Design systems so that they can deliver value if they are incomplete. Where a system is composed of several other systems, it should not just be useful if all of its components are working properly. Rather, there should be several "stable intermediate forms" so that a partial system works and can do useful things.

2. Be realistic about what can be controlled. The best performance from a SoS may be achieved when an individual or group exerts control over the overall system and its constituents. If there is no control, then delivering value from the SoS is difficult. However, attempts to overcontrol the SoS are likely to lead to resistance from the individual system owners and consequent delays in system deployment and evolution.

3. Focus on the system interfaces. To build a successful system of systems, you have to design interfaces so that the system elements can interoperate. It is

important that these interfaces are not too restrictive so that the system elements can evolve and continue to be useful participants in the SoS.

4. Provide collaboration incentives. When the system elements are independently owned and managed, it is important each system owner have incentives to continue to participate in the system. These may be financial incentives (pay per use or reduced operational costs), access incentives (you share your data and I'll share mine), or community incentives (participate in a SoS and you get a say in the community).

Sillitto (Sillitto 2010) has added to these principles and suggests additional important design guidelines. These include the following:

1. Design a SoS as node and web architecture. Nodes are sociotechnical systems that include data, software, hardware, infrastructure (technical components), and organizational policies, people, processes, and training (sociotechnical). The web is not just the communications infrastructure between nodes, but it also provides a mechanism for informal and formal social communications between the people managing and running the systems at each node.
2. Specify behavior as services exchanged between nodes. The development of service-oriented architectures now provides a standard mechanism for system operability. If a system does not already provide a service interface, then this interface should be implemented as part of the SoS development process.
3. Understand and manage system vulnerabilities. In any SoS, there will be unexpected failures and undesirable behavior. It is critically important to try to understand vulnerabilities and design the system to be resilient to such failures.

The key message that emerges from both Maier's and Sillitto's work is that SoS architects have to take a broad perspective. They need to look at the system as a whole, taking into account both technical and sociotechnical considerations. Sometimes the best solution to a problem is not more software but changes to the rules and policies that govern the operation of the system.

Architectural frameworks such as MODAF (MOD 2008) and TOGAF (TOGAF is a registered trademark of The Open Group 2011) have been suggested as a means of supporting the architectural design of systems of systems. Architectural frameworks were originally developed to support enterprise systems architectures, which are portfolios of separate systems. Enterprise systems may be organizational systems of systems, or they may have a simpler management structure so that the system portfolio can be managed as a whole. Architectural frameworks are intended for the development of organizational systems of systems where there is a single governance authority for the entire SoS.

An architectural framework recognizes that a single model of an architecture does not present all of the information needed for architectural and business analysis. Rather, frameworks propose a number of architectural views that should be created and maintained to describe and document enterprise systems. Frameworks have much in common and tend to reflect the language and history of the organizations

Figure 20.9 The TOGAF architecture development method (TOGAF® Version 9.1, © 1999–2011. The Open Group.)

involved. For example, MODAF and DODAF are comparable frameworks from the UK Ministry of Defence (MOD) and the U.S. Department of Defense (DOD).

The TOGAF framework has been developed by the Open Group as an open standard and is intended to support the design of a business architecture, a data architecture, an application architecture, and a technology architecture for an enterprise. At its heart is the Architecture Development Method (ADM), which consists of a number of discrete phases. These are shown in Figure 20.9, taken from the TOGAF reference documentation (Open Group 2011).

All architectural frameworks involve the production and management of a large set of architectural models. Each of the activities shown in Figure 20.8 leads to the production of system models. However, this is problematic for two reasons:

1. Initial model development takes a long time and involves extensive negotiations between system stakeholders. This slows the development of the overall system.
2. It is time-consuming and expensive to maintain model consistency as changes are made to the organization and the constituent systems in a SoS.

Architecture frameworks are fundamentally reductionist, and they largely ignore sociotechnical and political issues. While they do recognize that problems are difficult to define and are open-ended, they assume a degree of control and governance

Figure 20.10 Systems as data feeds

that is impossible to achieve in many systems of systems. They are a useful checklist to remind architects of things to think about in the architectural design process. However, I think that the overhead involved in model management and the reductionist approach taken by frameworks limits their usefulness in SoS architectural design.

20.5.1 Architectural patterns for systems of systems

I have described architectural patterns for different types of system in Chapters 6, 17, and 21. In short, an architectural pattern is a stylized architecture that can be recognized across a range of different systems. Architectural patterns are a useful way of stimulating discussions about the most appropriate architecture for a system and for documenting and explaining the architectures used. This section covers a number of "typical" patterns in systems of software systems. As with all architectural patterns, real systems are usually based on more than one of these patterns.

The notion of architectural patterns for systems of systems is still at an early stage of development. Kawalsky (Kawalsky et al. 2013) discusses the value of architectural patterns in understanding and supporting SoS design, with a focus on patterns for command and control systems. I find that patterns are effective in illustrating SoS organization, without the need for detailed domain knowledge.

Systems as data-feeds

In this architectural pattern (Figure 20.10), there is a principal system that requires data of different types. This data is available from other systems, and the principal system queries these systems to get the data required. Generally, the systems that provide data do not interact with each other. This pattern is often observed in organizational or federated systems where some governance mechanisms are in place.

For example, to license a vehicle in the UK, you need to have both valid insurance and a roadworthiness certificate. When you interact with the vehicle licensing system, it interacts with two other systems to check that these documents are valid. These systems are:

1. An *"insured vehicles" system*, which is a federated system run by car insurance companies that maintains information about all current car insurance policies.

Figure 20.11 Systems as data feeds with a unifying interface

2. An "*MOT certificate" system*, which is used to record all roadworthiness certificates issued by testing agencies licensed by the government.

The "systems as data feeds" architecture is an appropriate architecture to use when it is possible to identify entities in a unique way and create relatively simple queries about these entities. In the licensing system, vehicles can be uniquely identified by their registration number. In other systems, it may be possible to identify entities such as pollution monitors by their GPS coordinates.

A variant of the "systems as data feeds" architecture arises when a number of systems provide data that are similar but not identical. Therefore, the architecture has to include an intermediate layer as shown in Figure 20.11. The role of this intermediate layer is to translate the general query from the principal system into the specific query required by the individual information system.

For example, the iLearn environment interacts with school administration systems from three different providers. All of these systems provide the same information about students (names, personal information, etc.) but have different interfaces. The databases have different organizations, and the format of the data returned differs from one system to another. The unifying interface here detects where the user of the system is based and, using this regional information, knows which administrative system should be accessed. It then converts a standard query into the appropriate query for that system.

Problems that can arise in systems that use this pattern are primarily interface problems when the data feeds are unavailable or are slow to respond. It is important to ensure that timeouts are included in the system so that a failure of a data feed does not compromise the response time of the system as a whole. Governance mechanisms should be in place to ensure that the format of provided data is not changed without the agreement of all system owners.

Systems in a container

Systems in a container are systems of systems where one of the systems acts as a virtual container and provides a set of common services such as an authentication and a storage service. Conceptually, other systems are then placed into this container

Figure 20.12 Systems in a container

to make their functionality accessible to system users. Figure 20.12 illustrates a container system with three common services and six included systems. The systems that are included may be selected from an approved list of systems and need not be aware that they are included in the container. This pattern of SoS is most often observed in federated systems or system coalitions.

The iLearn environment is a system in a container. There are common services that support authentication, storage of user data, and system configuration. Other functionality comes from choosing existing systems such as a newspaper archive or a virtual learning environment and integrating these into the container.

Of course, you don't place systems into a real container to implement these systems of systems. Rather, for each approved system, there is a separate interface that allows it to be integrated with the common services. This interface manages the translation of the common services provided by the container and the requirements of the integrated system. It may also be possible to include systems that are not approved. However, these will not have access to the common services provided by the container.

Figure 20.13 illustrates this integration. This graphic is a simplified version of iLearn that provides three common services:

1. An authentication service that provides a single sign-in to all approved systems. Users do not have to maintain separate credentials for these systems.
2. A storage service for user data. This service can be seamlessly transferred to and from approved systems.
3. A configuration service that is used to include or remove systems from the container.

This example shows a version of iLearn for Physics. As well as an office productivity system (Office 365) and a VLE (Moodle), this system includes simulation and data analysis systems. Other systems—YouTube and a science encyclopedia—are also part of this system. However, these are not "approved," and so no container interface is available. Users must log on to these systems separately and organize their own data transfers.

Figure 20.13 The DLE as a container system

There are two problems with this type of SoS architecture:

1. A separate interface must be developed for each approved system so that common services can be used with these systems. This means that only a relatively small number of approved systems can be supported.
2. The owners of the container system have no influence on the functionality and behavior of the included systems. Systems may stop working, or they may be withdrawn at any time.

However, the main benefit of this architecture is that it allows for incremental development. An early version of the container system can be based on "unapproved" systems. Interfaces to these can be developed in later versions so that they are more closely integrated with the container services.

Trading systems

Trading systems are systems of systems where there is no single principal system but processing may take place in any of the constituent systems. The systems involved trade information among themselves. There may be one-to-one or one-to-many interactions between these systems. Each system publishes its own interface, but there may not be any interface standards that are followed by all systems. This system is shown in Figure 20.14. Trading systems may be federated systems or system coalitions.

An example of a trading SoS is a system of systems for algorithmic trading of stocks and shares. Brokers all have their own separate systems that can automatically buy and sell stock from other systems. They set prices and negotiate individually with these systems. Another example of a trading system is a travel aggregator that shows price comparisons and allows travel to be booked directly by a user.

Figure 20.14 A trading system of systems

Trading systems may be developed for any type of marketplace, with the information exchanged being information about the goods being traded and their prices. Although trading systems are systems in their own right and could conceivably be used for individual trading, they are most useful in an automated trading context where the systems negotiate directly with each other.

The major problem with this type of system is that there is no governance mechanism, so any of the systems involved may change at any time. Because these changes may contradict the assumptions made by other systems, trading cannot continue. Sometimes the owners of the systems in the coalition wish to be able to continue trading with other systems and so may make informal arrangements to ensure that changes to one system do not make trading impossible. In other cases, such as a travel aggregator, an airline may deliberately change its system so that it is unavailable and so force bookings to be made directly with it.

KEY POINTS

- Systems of systems are systems where two or more of the constituent systems are independently managed and governed.
- Three types of complexity are important for systems of systems—technical complexity, managerial complexity, and governance complexity.
- System governance can be used as the basis for a classification scheme for SoS. This leads to three classes of SoS, namely, organizational systems, federated systems, and system coalitions.
- Reductionism as an engineering method breaks down because of the inherent complexity of systems of systems. Reductionism assumes clear system boundaries, rational decision making, and well-defined problems. None of these are true for systems of systems.
- The key stages of the SoS development process are conceptual design, system selection, architectural design, interface development, and integration and deployment. Governance and management policies must be designed in parallel with these activities.

- Architectural patterns for systems of systems are a means of describing and discussing typical architectures for SoS. Important patterns are systems as data feeds, systems in a container, and trading systems.

FURTHER READING

"Architecting Principles for Systems of Systems." A now-classic paper on systems of systems that introduces a classification scheme for SoS, discusses its value, and proposes a number of architectural principles for SoS design. (M. Maier, *Systems Engineering*, 1 (4), 1998).

Ultra-large Scale Systems: The Software Challenge of the Future This book, produced for the U.S. Department of Defense in 2006, introduces the notion of ultra-large-scale systems, which are systems of systems with hundreds of nodes. It discusses the issues and challenges in developing such systems. (L. Northrop et al., Software Engineering Institute, 2006). http://www.sei.cmu.edu/library/assets/ULS_Book20062.pdf

"Large-scale Complex IT Systems." This paper discusses the problems of large-scale complex IT systems that are systems of systems and expands on the ideas here on the breakdown of reductionism. It proposes a number of research challenges in the area of SoS. (I. Sommerville et al., Communications of the ACM, 55 (7), July 2012). http://dx.doi.org/ 10.1145/2209249.2209268

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/systems-engineering/

EXERCISES

20.1. Explain why managerial and operational independence are the key distinguishing characteristics of systems of systems when compared to other large, complex systems.

20.2. Briefly explain any four essential characteristics of systems of systems.

20.3. The classification of SoS presented in Section 20.2 suggests a governance-based classification scheme. Giving reasons for your answer, identify the classifications for the following systems of systems:

(a) A health care system that provides unified access to all patient health records from hospitals, clinics, and primary care.
(b) The World Wide Web
(c) A government system that provides access to a range of welfare services such as pensions, disability benefits, and unemployment benefits.

Are there any problems with the suggested classification for any of these systems?

20.4. Explain what is meant by reductionism and why it is effective as a basis for many kinds of engineering.

20.5. Define systems of systems engineering. List the problems of software SoS engineering that are also common to problems of integrating large-scale application systems.

20.6. How beneficial is a unified user interface in the interface design of SoS? What are the factors on which the cost-effectiveness of a unified user interface is dependent?

20.7. Sillitto suggests that communications between nodes in a SoS are not just technical but should also include informal sociotechnical communications between the people involved in the system. Using the iLearn SoS as an example, suggest where these informal communications may be important to improve the effectiveness of the system.

20.8. Suggest the closest-fit architectural pattern for the systems of systems introduced in Exercise 20.3.

20.9. The trading system pattern assumes that there is no central authority involved. However, in areas such as equity trading, trading systems must follow regulatory rules. Suggest how this pattern might be modified to allow a regulator to check that these rules have been followed. This should not involve all trades going through a central node.

20.10. You work for a software company that has developed a system that provides information about consumers and that is used within a SoS by a number of other retail businesses. They pay you for the services used. Discuss the ethics of changing the system interfaces without notice to coerce users into paying higher charges. Consider this question from the point of view of the company's employees, customers, and shareholders.

REFERENCES

Boehm, B., and C. Abts. 1999. "COTS Integration: Plug and Pray?" *Computer* 32 (1): 135–138. doi:10.1109/2.738311.

Hitchins, D. 2009. "System of Systems—The Ultimate Tautology." http://www.hitchins.net/profs-stuff/profs-blog/system-of-systems---the.html

Kawalsky, R., D. Joannou, Y. Tian, and A. Fayoumi. 2013. "Using Architecture Patterns to Architect and Analyze Systems of Systems." In *Conference on Systems Engineering Research (CSER 13)*, 283–292. doi:10.1016/j.procs.2013.01.030.

Maier, M. W. 1998. "Architecting Principles for Systems-of-Systems." *Systems Engineering* 1 (4): 267–284. doi:10.1002/(SICI)1520–6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D.

MOD, UK. 2008. "MOD Architecture Framework." https://www.gov.uk/mod-architecture-framework

Northrop, Linda, R. P. Gabriel, M. Klein, and D. Schmidt. 2006. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Pittsburgh: Software Engineering Institute. http://www.sei.cmu.edu/library/assets/ULS_Book20062.pdf

Open Group. 2011. "Open Group Standard TOGAF Version 9.1." http://pubs.opengroup.org/architecture/togaf91-doc/arch/

Rittel, H., and M. Webber. 1973. "Dilemmas in a General Theory of Planning." *Policy Sciences* 4: 155–169. doi:10.1007/BF01405730.

Royal Academy of Engineering. 2004. "Challenges of Complex IT Projects." London. http://www.bcs.org/upload/pdf/complexity.pdf

Sillitto, H. 2010. "Design Principles for Ultra-Large-Scale Systems." In *Proceedings of the 20th International Council for Systems Engineering International Symposium*. Chicago.

Sommerville, I., D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatkowska, J. McDermid, and R. Paige. 2012. "Large-Scale Complex IT Systems." *Comm. ACM* 55 (7): 71–77. doi:10.1145/2209249.2209268.

Stevens, R. 2010. *Engineering Mega-Systems: The Challenge of Systems Engineering in the Information Age*. Boca Raton, FL: CRC Press.

21

Real-time software engineering

Objectives

The objective of this chapter is to introduce some of the characteristic features of embedded real-time software engineering. When you have read this chapter, you will:

- understand the concept of embedded software, which is used to control systems that react to external events in their environment;
- have been introduced to a design process for real-time systems, where the software systems are organized as a set of cooperating processes;
- understand three architectural patterns that are commonly used in embedded real-time systems design;
- understand the organization of real-time operating systems and the role that they play in an embedded, real-time system.

Contents

21.1 Embedded systems design
21.2 Architectural patterns for real-time software
21.3 Timing analysis
21.4 Real-time operating systems

Computers are used to control a wide range of systems from simple domestic machines, through games controllers, to entire manufacturing plants. These computers interact directly with hardware devices. Their software must react to events generated by the hardware and often issue control signals in response to these events. These signals result in an action, such as the initiation of a phone call, the movement of a character on the screen, the opening of a valve, or the display of the system status. The software in these systems is embedded in system hardware, often in read-only memory. It responds, in real time, to events from the system's environment. By real time, I mean that the software system has a deadline for responding to external events. If this deadline is missed, then the overall hardware–software system will not operate correctly.

Embedded software is very important economically because almost every electrical device now includes software. There are therefore many more embedded software systems than other types of software systems. Ebert and Jones (Ebert and Jones 2009) estimated that there were about 30 embedded microprocessor systems per person in developed countries. This figure was increasing between 10% and 20% per year. This suggests that, by 2020, there will be more than 100 embedded systems per person.

Responsiveness in real time is the critical difference between embedded systems and other software systems, such as information systems, web-based systems, or personal software systems, whose main purpose is data processing. For non–real-time systems, the correctness of a system can be defined by specifying how system inputs map to corresponding outputs that should be produced by the system. In response to an input, a corresponding output should be generated by the system and, often, some data should be stored. For example, if you choose a create command in a patient information system, then the correct system response is to create a new patient record in a database and to confirm that this has been done. Within reasonable limits, it does not matter how long this takes.

However, in a real-time system, the correctness depends both on the response to an input and the time taken to generate that response. If the system takes too long to respond, then the required response may be ineffective. For example, if embedded software controlling a car's braking system is too slow, then an accident may occur because it is impossible to stop the car in time.

Therefore, time is fundamental in the definition of a real-time software system:

> *A real-time software system is a system whose correct operation depends on both the results produced by the system and the time at which these results are produced. A "soft real-time system" is a system whose operation is degraded if results are not produced according to the specified timing requirements. If results are not produced according to the timing specification in a "hard real-time system," this is considered to be a system failure.*

Timely response is an important factor in all embedded systems, but not all embedded systems require a very fast response. For example, the insulin pump software that I have used as an example in several chapters of this book is an embedded system. However, while the system needs to check the glucose level at periodic intervals, it does not need to

respond very quickly to external events. The wilderness weather station software is also an embedded system, but, again, it does not require a fast response to external events.

As well as the need for real-time response, there are other important differences between embedded systems and other types of software system:

1. Embedded systems generally run continuously and do not terminate. They start when the hardware is switched on, and execute until the hardware is switched off. Techniques for reliable software engineering, as discussed in Chapter 11, may therefore have to be used to ensure continuous operation. The real-time system may include update mechanisms that support dynamic reconfiguration so that the system can be updated while it is in service.

2. Interactions with the system's environment are unpredictable. In interactive systems, the pace of the interaction is controlled by the system. By limiting user options, the events and commands to be processed are known in advance. By contrast, real-time embedded systems must be able to respond to expected and unexpected events at any time. This leads to a design for real-time systems based on concurrency, with several processes executing in parallel.

3. Physical limitations may affect the design of a system. Examples of limitations include restrictions on the power available to the system and the physical space taken up by the hardware. These limitations may generate requirements for the embedded software, such as the need to conserve power and so prolong battery life. Size and weight limitations may mean that the software has to take over some hardware functions because of the need to limit the number of chips used in the system.

4. Direct hardware interaction may be necessary. In interactive systems and information systems, a layer of software (the device drivers) hides the hardware from the operating system. This is possible because you can only connect a few types of device to these systems, such as keyboards, mice, and displays. By contrast, embedded systems may have to interact with a wide range of hardware devices that do not have separate device drivers.

5. Issues of safety and reliability may dominate the system design. Many embedded systems control devices whose failure may have high human or economic costs. Therefore, dependability is critical, and the system design has to ensure safety-critical behavior at all times. This often leads to a conservative approach to design where tried and tested techniques are used instead of newer techniques that may introduce new failure modes.

Real-time embedded systems can be thought of as reactive systems; that is, they must react to events in their environment (Berry 1989; Lee 2002). Response times are often governed by the laws of physics rather than chosen for human convenience. This is in contrast to other types of software where the system controls the speed of the interaction. For example, the word processor that I am using to write this book can check spelling and grammar, and there are no practical limits on the time taken to do so.

21.1 Embedded system design

During the design process for embedded software, software designers have to consider in detail the design and performance of the system hardware. Part of the system design process may involve deciding which system capabilities are to be implemented in software and which in hardware. For many real-time systems that are embedded in consumer products, such as the systems in cell phones, the costs and power consumption of the hardware are critical. Specific processors designed to support embedded systems may be used. For some systems, special-purpose hardware may have to be designed and built.

A top-down software design process, in which the design starts with an abstract model that is decomposed and developed in a series of stages, is impractical for most real-time systems. Low-level decisions on hardware, support software, and system timing must be considered early in the process. These limit the flexibility of system designers. Additional software functionality, such as battery and power management, may have to be included in the system.

Given that embedded systems are reactive systems that react to events in their environment, the most general approach to embedded, real-time software design is based on a stimulus-response model. A stimulus is an event occurring in the software system's environment that causes the system to react in some way; a response is a signal or message that the software sends to its environment.

You can define the behavior of a real-time system by listing the stimuli received by the system, the associated responses, and the time at which the response must be produced. For example, Figure 21.1 shows possible stimuli and system responses for a burglar alarm system (discussed in Section 21.2.1).

Stimuli fall into two classes:

1. *Periodic stimuli* These occur at predictable time intervals. For example, the system may examine a sensor every 50 milliseconds and take action (respond) depending on that sensor value (the stimulus).

2. *Aperiodic stimuli* These occur irregularly and unpredictably and are usually signaled, using the computer's interrupt mechanism. An example of such a stimulus would be an interrupt indicating that an I/O transfer was complete and that data was available in a buffer.

Stimuli come from sensors in the system's environment, and responses are sent to actuators, as shown in Figure 21.2. These actuators control equipment, such as a pump, which then makes changes to the system's environment. The actuators themselves may also generate stimuli. The stimuli from actuators often indicate that some problem with the actuator has occurred, which must be handled by the system.

A general design guideline for real-time systems is to have separate control processes for each type of sensor and actuator (Figure 21.3). For each type of sensor, there may be a sensor management process that handles data collection from these sensors. Data-processing processes compute the required responses for the stimuli received by the system. Actuator control processes are associated with each actuator

Stimulus	Response
Clear alarms	Switch off all active alarms; switch off all lights that have been switched on.
Console panic button positive	Initiate alarm; turn on lights around console; call police.
Power supply failure	Call service technician.
Sensor failure	Call service technician.
Single sensor positive	Initiate alarm; turn on lights around site of positive sensor.
Two or more sensors positive	Initiate alarm; turn on lights around sites of positive sensors; call police with location of suspected break-in.
Voltage drop of between 10% and 20%	Switch to battery backup; run power supply test.
Voltage drop of more than 20%	Switch to battery backup; initiate alarm; call police, run power supply test.

Figure 21.1 Stimuli and responses for a burglar alarm system

and manage the operation of that actuator. This model allows data to be collected quickly from the sensor (before it is overwritten by the next input) and enables processing and the associated actuator response to be carried out later.

A real-time system has to respond to stimuli that occur at different times. You therefore have to organize the system architecture so that, as soon as a stimulus is received, control is transferred to the correct handler. This is impractical in sequential programs. Consequently, real-time software systems are normally designed as a set of concurrent, cooperating processes. To support the management of these processes, the execution platform on which the real-time system executes may include a real-time operating system (discussed in Section 21.4). The functions provided by this operating system are accessed through the runtime support system for the real-time programming language that is used.

There is no standard embedded system design process. Rather, different processes are used that depend on the type of system, available hardware, and the organization that is developing the system. The following activities may be included in a real-time software design process:

1. *Platform selection* In this activity, you choose an execution platform for the system, that is, the hardware and the real-time operating system to be used. Factors that influence these choices include the timing constraints on the system, limitations on power available, the experience of the development team, and the price target for the delivered system.

2. *Stimuli/response identification* This involves identifying the stimuli that the system must process and the associated response or responses for each stimulus.

Figure 21.2 A general model of an embedded real-time system

3. *Timing analysis* For each stimulus and associated response, you identify the timing constraints that apply to both stimulus and response processing. These constraints are used to establish the deadlines for the processes in the system.

4. *Process design* Process design involves aggregating the stimulus and response processing into a number of concurrent processes. A good starting point for designing the process architecture is the architectural patterns that I describe in Section 20.2. You then optimize the process architecture to reflect the specific requirements that you have to implement.

5. *Algorithm design* For each stimulus and response, you design algorithms to carry out the required computations. Algorithm designs may have to be developed relatively early in the design process to indicate the amount of processing required and the time needed to complete that processing. This is especially important for computationally intensive tasks, such as signal processing.

6. *Data design* You specify the information that is exchanged by processes and the events that coordinate information exchange, and design data structures to manage this information exchange. Several concurrent processes may share these data structures.

7. *Process scheduling* You design a scheduling system that will ensure that processes are started in time to meet their deadlines.

The specific activities and the activity sequence in a real-time system design process depend on the type of system being developed, its novelty, and its environment.

Figure 21.3 Sensor and actuator processes

Figure 21.4 Producer/consumer processes sharing a circular buffer

In some cases, for new systems, you may be able to follow a fairly abstract approach where you start with the stimuli and associated processing, and decide on the hardware and execution platforms late in the process. In other cases, the choice of hardware and operating system is made before the software design starts. You then have to design the software to take account of the constraints imposed by the system hardware.

Processes in a real-time system have to be coordinated and share information. Process coordination mechanisms ensure mutual exclusion to shared resources. When one process is modifying a shared resource, other processes should not be able to change that resource. Mechanisms for ensuring mutual exclusion include semaphores, monitors, and critical regions. These process synchronization mechanisms are described in most operating system books (Silberschaltz, Galvin, and Gagne 2013; Stallings 2014).

When designing the information exchange between processes, you have to take into account that these processes may be running at different speeds. One process is producing information, and the other process is consuming that information. If the producer is running faster than the consumer, new information could overwrite a previously read information item before the consumer process has read the original information. If the consumer process is running faster than the producer process, the same item could be read twice.

To avoid this problem, you should implement information exchange using a shared buffer and use mutual exclusion mechanisms to control access to that buffer. This means that information can't be overwritten before it has been read and that information cannot be read twice. Figure 21.4 illustrates the organization of a shared buffer. This is usually implemented as a circular queue, using a list data structure. Mismatches in speed between the producer and consumer processes can be accommodated without having to delay process execution.

The producer process always enters data in the buffer location at the end of the queue (represented as v10 in Figure 21.4). The consumer process always retrieves information from the head of the queue (represented as v1 in Figure 21.4). After the consumer process has retrieved the information, the tail of the queue is adjusted to point at the next item (v2). After the producer process has added information, the tail of the queue is adjusted to point at the next free slot in the queue.

Obviously, it is important to ensure that the producer and consumer process do not attempt to access the same item at the same time (i.e., when Head = Tail). If they do, the value of the item is unpredictable. The system also has to ensure that the producer process does not add items to a full buffer and that the consumer process does not try to take items from an empty buffer.

To do this, you implement the circular buffer as a process with Get and Put operations to access the buffer. The Put operation is called by the producer process and the Get operation by the consumer process. Synchronization primitives, such as semaphores or critical regions, are used to ensure that the operation of Get and Put are synchronized, so that they don't access the same location simultaneously. If the buffer is full, the Put process has to wait until a slot is free; if the buffer is empty, the Get process has to wait until an entry has been made.

Once you have chosen the execution platform for the system, designed a process architecture, and decided on a scheduling policy, you have to check that the system will meet its timing requirements. You can perform this check through static analysis of the system using knowledge of the timing behavior of components, or through simulation. This analysis may reveal that the system will not perform adequately. The process architecture, the scheduling policy, the execution platform, or all of these may then have to be redesigned to improve the performance of the system.

Timing constraints or other requirements may sometimes mean that it is best to implement some system functions, such as signal processing, in hardware. Modern hardware components, such as FPGAs (field-programmable gate arrays), are flexible and can be adapted to different functions. Hardware components deliver much better performance than the equivalent software. System processing bottlenecks can be identified and replaced by hardware, thus avoiding expensive software optimization.

21.1.1 Real-time system modeling

The events that a real-time system must react to often cause the system to move from one state to another. For this reason, state models, which I introduced in Chapter 5, are used to describe real-time systems. A state model of a system assumes that, at any time, the system is in one of a number of possible states. When a stimulus is received, this may cause a transition to a different state. For example, a system controlling a valve may move from a state "Valve open" to a state "Valve closed" when an operator command (the stimulus) is received.

State models are an integral part of real-time system design methods. The UML supports the development of state models based on Statecharts (Harel 1987, 1988). Statecharts are formal state machine models that support hierarchical states, so that groups of states can be considered as a single entity. Douglass discusses the use of the UML in real-time systems development (Douglass 1999).

I have already illustrated this approach to system modeling in Chapter 5 where I used an example of a model of a simple microwave oven. Figure 21.5 is another example of a state model that shows the operation of a fuel delivery software system embedded in a petrol (gas) pump. The rounded rectangles represent system states, and the arrows represent stimuli that force a transition from one state to another.

Figure 21.5 State machine model of a petrol (gas) pump

The names chosen in the state machine diagram are descriptive. The associated information indicates actions taken by the system actuators or information that is displayed. Notice that this system never terminates but idles in a waiting state when the pump is not operating.

The fuel delivery system is designed to allow unattended operation, with the following sequence of actions:

1. The buyer inserts a credit card into a card reader built into the pump. This causes a transition to a **Reading** state where the card details are read and the buyer is then asked to remove the card.

2. Removal of the card triggers a transition to a **Validating** state where the card is validated.

3. If the card is valid, the system initializes the pump and, when the fuel hose is removed from its holster, transitions to the **Delivering** state, where is ready to deliver fuel. Activating the trigger on the nozzle causes fuel to be pumped; this stops when the trigger is released (for simplicity, I have ignored the pressure switch that is designed to stop fuel spillage).

Real-time Java

The Java programming language has been modified to make it suitable for real-time systems development. These modifications include asynchronous communications, the addition of time, including absolute and relative time, a new thread model where threads cannot be interrupted by garbage collection, and a new memory management model that avoids the unpredictable delays that can result from garbage collection.

http://software-engineering-book.com/web/real-time-java/

4. After the fuel delivery is complete and the buyer has replaced the hose in its holster, the system moves to a Paying state where the user's account is debited.
5. After payment, the pump software returns to the Waiting state.

State models are used in model-driven engineering, which I discussed in Chapter 5, to define the operation of a system. They can be transformed automatically or semiautomatically to an executable program.

21.1.2 Real-time programming

Programming languages for real-time systems development have to include facilities to access system hardware, and it should be possible to predict the timing of particular operations in these languages. Hard real-time systems, running on limited hardware, are still sometimes programmed in assembly language so that tight deadlines can be met. Systems programming languages, such as C, which allow efficient code to be generated, are widely used.

The advantage of using a systems programming language like C is that it allows the development of efficient programs. However, these languages do not include constructs to support concurrency or the management of shared resources. Concurrency and resource management are implemented through calls to primitives provided by the real-time operating system for mutual exclusion. Because the compiler cannot check these calls, programming errors are more likely. Programs are also often more difficult to understand because the language does not include real-time features. As well as understanding the program, the reader also has to know how real-time support is provided using system calls.

Because real-time systems must meet their timing constraints, you may not be able to use object-oriented development for hard real-time systems. Object-oriented development involves hiding data representations and accessing attribute values through operations defined with the object. There is a significant performance overhead in object-oriented systems because extra code is required to mediate access to attributes and handle calls to operations. The consequent loss of performance may make it impossible to meet real-time deadlines.

A version of Java has been developed for embedded systems development (Burns and Wellings 2009; Bruno and Bollella 2009). This language includes a modified thread mechanism, which allows threads to be specified that will not be interrupted

by the language garbage collection mechanism. Asynchronous event handling and timing specification has also been included. However, at the time of writing, this specification has mostly been used on platforms that have significant processor and memory capacity (e.g., a cell phone) rather than simpler embedded systems, with more limited resources. These systems are still usually implemented in C.

21.2 Architectural patterns for real-time software

Architectural patterns are abstract, stylized descriptions of good design practice. They capture knowledge about the organization of system architectures, when these architectures should be used, and their advantages and disadvantages. You use an architectural pattern to understand an architecture and as starting point for creating your own, specific architectural design.

The difference between real-time and interactive software means that there are distinct architectural patterns for real-time embedded systems. Real-time systems' patterns are process-oriented rather than object- or component-oriented. In this section, I discuss three real-time architectural patterns that are commonly used:

1. *Observe and React* This pattern is used when a set of sensors are routinely monitored and displayed. When the sensors show that some event has occurred (e.g., an incoming call on a cell phone), the system reacts by initiating a process to handle that event.
2. *Environmental Control* This pattern is used when a system includes sensors, which provide information about the environment and actuators that can change the environment. In response to environmental changes detected by the sensor, control signals are sent to the system actuators.
3. *Process Pipeline* This pattern is used when data has to be transformed from one representation to another before it can be processed. The transformation is implemented as a sequence of processing steps, which may be carried out concurrently. This allows for very fast data processing, because a separate core or processor can execute each transformation.

These patterns can of course be combined, and you will often see more than one of them in a single system. For example, when the Environmental Control pattern is used, it is very common for the actuators to be monitored using the Observe and React pattern. In the event of an actuator failure, the system may react by displaying a warning message, shutting down the actuator, switching in a backup system, and so forth.

The patterns that I cover are architectural patterns that describe the overall structure of an embedded system. Douglass (Douglass 2002) describes lower-level, real-time design patterns that support more detailed design decision making. These patterns include design patterns for execution control, communications, resource allocation, and safety and reliability.

Name	Observe and React
Description	The input values of a set of sensors of the same types are collected and analyzed. These values are displayed in some way. If the sensor values indicate that some exceptional condition has arisen, then actions are initiated to draw the operator's attention to that value and, if necessary, take actions in response to the exceptional value.
Stimuli	Values from sensors attached to the system.
Responses	Outputs to display, alarm triggers, signals to reacting systems.
Processes	Observer, Analysis, Display, Alarm, Reactor.
Used in	Monitoring systems, alarm systems.

Figure 21.6 The Observe and React pattern

Figure 21.7 The Observe and React process structure

These architectural patterns should be the starting point for an embedded systems design; however, they are not design templates. If you use them as such, you will probably end up with an inefficient process architecture. You have to optimize the process structure to ensure that you do not have too many processes. You also should ensure that there is a clear correspondence between the processes and the sensors and actuators in the system.

21.2.1 Observe and react

Monitoring systems are an important class of embedded real-time systems. A monitoring system examines its environment through a set of sensors and usually displays the state of the environment in some way. This could be on a built-in screen, on special-purpose instrument displays, or on a remote display. If the system detects some exceptional event or sensor state, the monitoring system takes some action.

Figure 21.8 The process structure of a burglar alarm system

This often involves raising an alarm to draw an operator's attention to the event. Sometimes the system may initiate some other preventative action, such as shutting down the system to preserve it from damage.

The Observe and React pattern (Figures 21.6 and 21.7) is commonly used in monitoring systems. The values of sensors are observed, and the system initiates actions that depend on these sensor values. Monitoring systems may be composed of several instantiations of the Observe and React pattern, one for each type of sensor in the system. Depending on the system requirements, you may then optimize the design by combining processes (e.g., you may use a single display process to display the information from all of the different types of sensor).

As an example of the use of this pattern, consider the design of a burglar alarm system to be installed in an office building:

> *A software system is to be implemented as part of a burglar alarm system for commercial buildings. This uses several different types of sensors. These sensors include movement detectors in individual rooms, door sensors that detect corridor doors opening, and window sensors on ground-floor windows that can detect when a window has been opened.*
>
> *When a sensor detects the presence of an intruder, the system automatically calls the local police and, using a voice synthesizer, reports the location of the alarm. It switches on lights in the rooms around the active sensor and sets off an audible alarm. The sensor system is normally powered by mains power but is equipped with a battery backup. Power loss is detected using a separate power circuit monitor that monitors the mains voltage. If a voltage drop is detected, the system assumes that intruders have interrupted the power supply, so an alarm is raised.*

A process architecture for the alarm system is shown in Figure 21.8. The arrows represent signals sent from one process to another. This system is a "soft" real-time system that does not have stringent timing requirements. The sensors only need to detect

Name	Environmental Control
Description	The system analyzes information from a set of sensors that collect data from the system's environment. Further information may also be collected on the state of the actuators that are connected to the system. Based on the data from the sensors and actuators, control signals are sent to the actuators, which then cause changes to the system's environment. Information about the sensor values and the state of the actuators may be displayed.
Stimuli	Values from sensors attached to the system and the state of the system actuators.
Responses	Control signals to actuators display information.
Processes	Monitor, Control, Display, Actuator driver, Actuator monitor.
Used in	Control systems.

Figure 21.9 The Environmental Control pattern

Figure 21.10 The Environmental Control process structure

the presence of people rather than high-speed events, so they only need to be polled 2 or 3 times per second. I cover the timing requirements for this system in Section 21.3.

I have already introduced the stimuli and responses in this alarm system in Figure 21.1. These responses are used as a starting point for the system design. The Observe and React pattern is used in this design. There are observer processes associated with each type of sensor and reactor processes for each type of reaction. A single analysis process checks the data from all of the sensors. The display processes in the pattern are combined into a single display process.

21.2.2 Environmental Control

The most widespread use of real-time embedded software is in control systems. In these systems, the software controls the operation of equipment, based on stimuli

Figure 21.11 Control system architecture for an anti-skid braking system

from the equipment's environment. For example, an anti-skid braking system in a car monitors the car's wheels and brake system (the system's environment). It looks for signs that the wheels are skidding when brake pressure is applied. If this is the case, the system adjusts the brake pressure to stop the wheels locking and reduce the likelihood of a skid.

Control systems may make use of the Environmental Control pattern, which is a general control pattern that includes sensor and actuator processes. This pattern is described in Figure 21.9, with the process architecture shown in Figure 21.10. A variant of this pattern leaves out the display process. This variant is used in situations where user intervention is not required or where the rate of control is so high that a display would not be meaningful.

This pattern can be the basis for a control system design with an instantiation of the Environmental Control pattern for each actuator (or actuator type) being controlled. You then optimize the design to reduce the number of processes. For example, you may combine actuator monitoring and actuator control processes, or you may have a single monitoring and control process for several actuators. The optimizations that you choose depend on the timing requirements. You may need to monitor sensors more frequently than you send control signals, in which case it may be impractical to combine control and monitoring processes. There may also be direct feedback between the actuator control and the actuator monitoring process. This allows fine-grain control decisions to be made by the actuator control process.

You can see how this pattern is used in Figure 21.11, which shows an example of a controller for a car braking system. The starting point for the design is associating an instance of the pattern with each actuator type in the system. In this case, there are four actuators, with each controlling the brake on one wheel. The individual sensor processes are combined into a single wheel-monitoring process that monitors the sensors on all

Name	Process Pipeline
Description	A pipeline of processes is set up with data moving in sequence from one end of the pipeline to another. The processes are often linked by synchronized buffers to allow the producer and consumer processes to run at different speeds. The culmination of a pipeline may be display or data storage, or the pipeline may terminate in an actuator.
Stimuli	Input values from the environment or some other process
Responses	Output values to the environment or a shared buffer
Processes	Producer, Buffer, Consumer
Used in	Data acquisition systems, multi-media systems

Figure 21.12 The Process Pipeline pattern

Figure 21.13 Process Pipeline process structure

wheels. This monitors the state of each wheel to check if the wheel is turning or locked. A separate process monitors the pressure on the brake pedal exerted by the car driver.

The system includes an anti-skid feature, which is triggered if the sensors indicate that a wheel is locked when the brake has been applied. This means that there is insufficient friction between the road and the tire; in other words, the car is skidding. If the wheel is locked, the driver cannot steer that wheel. To counteract this effect, the system sends a rapid sequence of on/off signals to the brake on that wheel, which allows the wheel to turn and control to be regained.

The **Wheel monitor** process monitors whether or not each wheel is turning. If a wheel is skidding (not turning), it informs the **Analysis** process. This then signals the processes associated with the wheels that are skidding to initiate anti-skid braking.

21.2.3 Process pipeline

Many real-time systems are concerned with collecting analog data from the system's environment. They then digitize that data for analysis and processing by the system. The system may also convert digital data to analog data, which it then sends to its environment. For example, a software radio accepts incoming packets of digital data representing the radio transmission and transforms the data into a sound signal that people can listen to.

The data processing involved in many of these systems has to be carried out very quickly. Otherwise, incoming data may be lost and outgoing signals may be broken up because essential information is missing. The Process Pipeline pattern makes this rapid processing possible by breaking down the required data processing into a sequence of separate transformations. Each of these transformations is implemented

Figure 21.14 Neutron flux data acquisition

by an independent process. This architecture is efficient for systems that use multiple processors or multicore processors. Each process in the pipeline can be associated with a separate processor or core, so that the processing steps can be carried out in parallel.

Figure 21.12 is a brief description of the data pipeline pattern, and Figure 21.13 shows the process architecture for this pattern. Notice that the processes involved produce and consume information. The processes exchange information using synchronized buffers, as I explained in Section 21.1. Producer and consumer processes can thereby operate at different speeds without data losses.

An example of a system that may use a process pipeline is a high-speed data acquisition system. Data acquisition systems collect data from sensors for subsequent processing and analysis. These systems are used in situations where the sensors are collecting large volumes of data from the system's environment and it isn't possible or necessary to process that data in real time. Rather, it is collected and stored for later analysis. Data acquisition systems are often used in scientific experiments and process control systems where physical processes, such as chemical reactions, are very rapid. In these systems, the sensors may be generating data very quickly, and the data acquisition system has to ensure that a sensor reading is collected before the sensor value changes.

Figure 21.14 is a simplified model of a data acquisition system that might be part of the control software in a nuclear reactor. This system collects data from sensors monitoring the neutron flux (the density of neutrons) in the reactor. The sensor data is placed in a buffer from which it is extracted and processed. The average flux level is displayed on an operator's display and stored for future processing.

21.3 Timing analysis

As I discussed in the introduction to this chapter, the correctness of a real-time system depends not just on the correctness of its outputs but also on the time at which these outputs were produced. Therefore, timing analysis is an important activity in the embedded, real-time software development process. In such an analysis, you calculate how often each process in the system must be executed to ensure that all inputs

are processed and all system responses are produced in a timely way. The results of the timing analysis are used to decide how frequently each process should execute and how these processes should be scheduled by the real-time operating system.

Timing analysis for real-time systems is particularly difficult when the system has to deal with a mixture of periodic and aperiodic stimuli and responses. Because aperiodic stimuli are unpredictable, you have to make assumptions about the probability of these stimuli occurring and therefore requiring service at any particular time. These assumptions may be incorrect, and system performance after delivery may not be adequate. Cooling's book (Cooling 2003) discusses techniques for real-time system performance analysis that takes aperiodic events into account.

As computers have become faster, it has become possible in many systems to design using only periodic stimuli. When processors were slow, aperiodic stimuli had to be used to ensure that critical events were processed before their deadline, as delays in processing usually involved some loss to the system. For example, the failure of a power supply in an embedded system may mean that the system has to shut down attached equipment in a controlled way, within a very short time (say 50 milliseconds). This could be implemented as a "power fail" interrupt. However, it can also be implemented using a periodic process that runs frequently and checks the power. As long as the time between process invocations is short, there is still time to perform a controlled shutdown of the system before the lack of power causes damage. For this reason, I only discuss timing issues for periodic processes.

When you are analyzing the timing requirements of embedded real-time systems and designing systems to meet these requirements, you have to consider three key factors:

1. *Deadlines* The times by which stimuli must be processed and some response produced by the system. If the system does not meet a deadline, then, if it is a hard real-time system, this is a system failure; in a soft real-time system, it results in degraded system service.

2. *Frequency* The number of times per second that a process must execute so that you are confident that it can always meet its deadlines.

3. *Execution time* The time required to process a stimulus and produce a response. Execution time is not always the same because of the conditional execution of code, delays waiting for other processes, and so on. Therefore, you may have to consider both the average execution time of a process and the worst-case execution time for that process. The worst-case execution time is the maximum time that the process takes to execute. In a hard real-time system, you may have to make assumptions based on the worst-case execution time to ensure that deadlines are not missed. In soft real-time systems, you can base your calculations on the average execution time.

To continue the example of a power supply failure, let's calculate the worst-case execution time for a process that switches equipment power from mains

Figure 21.15 Power failure timing analysis

power to a battery backup. Figure 21.15 presents a timeline showing the events in the system:

1. Assume that, after a mains power failure event, it takes 50 milliseconds (ms) for the supplied voltage to drop to a level where the equipment may be damaged. The battery backup must therefore be activated and in operation within 50 ms. Usually, you allow for a margin of error, so you should set a shorter deadline of 40 ms because of physical variations in the equipment. This means that all equipment must be running on the battery backup power supply within 40 ms.

2. However, the battery backup system cannot be instantaneously activated. It takes 16 ms from starting the backup power supply to the supply being fully operational. This means that the time available to detect the power failure and start the battery backup system is 24 ms.

3. There is a process that is scheduled to run 250 times per second, that is, every 4 ms. This process assumes that there is a power supply problem if a significant drop in voltage occurs between readings and is sustained for three readings. This time is allowed so that temporary fluctuations do not cause a switch to the battery backup system.

4. In the above timeline, the power fails immediately after a reading has been taken. Therefore, reading R1 is the start reading for the power fail check. The voltage continues to drop for readings R2–R4, so a power failure is assumed. This is the worst possible case, where a power failure event occurs immediately after a sensor check, so 16 ms have elapsed since that event.

5. At this stage, the process that switches to the battery backup is started. Because the battery backup takes 16 ms to become operational, the worst-case execution time for this process is 8 ms, so that the 40 ms deadline can be reached.

Stimulus/Response	Timing requirements
Audible alarm	The audible alarm should be switched on within half a second of an alarm being raised by a sensor.
Communications	The call to the police should be started within 2 seconds of an alarm being raised by a sensor.
Door alarm	Each door alarm should be polled twice per second.
Lights switch	The lights should be switched on within half a second of an alarm being raised by a sensor.
Movement detector	Each movement detector should be polled twice per second.
Power failure	The switch to backup power must be completed within a deadline of 50 ms.
Voice synthesizer	A synthesized message should be available within 2 seconds of an alarm being raised by a sensor.
Window alarm	Each window alarm should be polled twice per second.

Figure 21.16 Timing requirements for the burglar alarm system

The starting point for timing analysis in a real-time system is the timing requirements, which should set out the deadlines for each required response in the system. Figure 21.16 shows possible timing requirements for the office building burglar alarm system discussed in Section 21.2.1. To simplify this example, let us ignore stimuli generated by system testing procedures and external signals to reset the system in the event of a false alarm. This means there are only two types of stimulus processed by the system:

1. Power failure is detected by observing a voltage drop of more than 20%. The required response is to switch the circuit to backup power by signaling an electronic power-switching device that switches the mains power to battery backup.
2. Intruder alarm is a stimulus generated by one of the system sensors. The response to this stimulus is to compute the room number of the active sensor, set up a call to the police, initiate the voice synthesizer to manage the call, and switch on the audible intruder alarm and building lights in the area.

As shown in Figure 21.16, you should list the timing constraints for each class of sensor separately, even when (as in this case) they are the same. By considering them separately, you leave scope for future change and make it easier to compute the number of times the controlling process has to be executed each second.

Allocating the system functions to concurrent processes is the next design stage. Four types of sensors must be polled periodically, each with an associated process: the voltage sensor, door sensors, window sensors, and movement detectors. Normally, the processes associated with the sensor will execute very quickly as all

Figure 21.17
Alarm process timing

they are doing is checking whether or not a sensor has changed its status (e.g., from off to on). It is reasonable to assume that the execution time to check and assess the state of one sensor is less than 1 millisecond.

To ensure that you meet the deadlines defined by the timing requirements, you then have to decide how frequently the related processes have to run and how many sensors should be examined during each execution of the process. There are obvious trade-offs here between frequency and execution time:

1. The deadline for detecting a change of state is 0.25 second, which means that each sensor has to be checked 4 times per second. If you examine one sensor during each process execution, then if there are N sensors of a particular type, you must schedule the process 4N times per second to ensure that all sensors are checked within the deadline.

2. If you examine four sensors, say, during each process execution, then the execution time is increased to about 4 ms, but you need only run the process N times/second to meet the timing requirement.

In this case, because the system requirements define actions when two or more sensors are positive, the best strategy is to examine sensors in groups, with groups based on the physical proximity of the sensors. If an intruder has entered the building, then it will probably be adjacent sensors that are positive.

When you have completed the timing analysis, you may then annotate the process model with information about frequency of execution and their expected execution time (see Figure 21.17). Here, periodic processes are annotated with their frequency, processes that are started in response to a stimulus are annotated with R, and the testing process is a background process, annotated with B. This background process

only runs when processor time is available. In general, it is simpler to design a system so that there are a small number of process frequencies. The execution times represent the required worst-case execution times of the processes.

The final step in the design process is to design a scheduling system that will ensure that a process will always be scheduled to meet its deadlines. You can only do this if you know the scheduling approaches that are supported by the real-time operating system (OS) used (Burns and Wellings 2009). The scheduler in the real-time OS allocates a process to a processor for a given amount of time. The time can be fixed, or it may vary depending on the priority of the process.

In allocating process priorities, you have to consider the deadlines of each process so that processes with short deadlines receive processor time to meet these deadlines. For example, the voltage monitor process in the burglar alarm needs to be scheduled so that voltage drops can be detected and a switch made to backup power before the system fails. This should therefore have a higher priority than the processes that check sensor values, as these have fairly relaxed deadlines compared to their expected execution time.

21.4 Real-time operating systems

The execution platform for most application systems is an operating system that manages shared resources and provides features such as a file system and runtime process management. However, the extensive functionality in a conventional operating system takes up a great deal of space and slows down the operation of programs. Furthermore, the process management features in the system may not be designed to allow fine-grain control over the scheduling of processes.

For these reasons, standard operating systems, such as Linux and Windows, are not normally used as the execution platform for real-time systems. Very simple embedded systems may be implemented as "bare metal" systems. The systems provide their own execution support and so include system startup and shutdown, process and resource management, and process scheduling. More commonly, however, embedded applications are built on top of a real-time operating system (RTOS), which is an efficient operating system that offers the features needed by real-time systems. Examples of RTOS are Windows Embedded Compact, VxWorks, and RTLinux.

A real-time operating system manages processes and resource allocation for a real-time system. It starts and stops processes so that stimuli can be handled, and it allocates memory and processor resources. The components of an RTOS (Figure 21.18) depend on the size and complexity of the real-time system being developed. For all except the simplest systems, they usually include:

1. A real-time clock, which provides the information required to schedule processes periodically.
2. If interrupts are supported, an interrupt handler, which manages aperiodic requests for service.

Figure 21.18 Components of a real-time operating system

3. A scheduler, which is responsible for examining the processes that can be executed and for choosing one of these processes for execution.
4. A resource manager, which allocates appropriate memory and processor resources to processes that have been scheduled for execution.
5. A dispatcher, which is responsible for starting the execution of processes.

Real-time operating systems for large systems, such as process control or telecommunication systems, may have additional facilities, namely, disk storage management, fault management facilities that detect and report system faults, and a configuration manager that supports the dynamic reconfiguration of real-time applications.

21.4.1 Process management

Real-time systems have to handle external events quickly and, in some cases, meet deadlines for processing these events. The event-handling processes must therefore be scheduled for execution in time to detect the event. They must also be allocated sufficient processor resources to meet their deadline. The process manager in an RTOS is responsible for choosing processes for execution, allocating processor and memory resources, and starting and stopping process execution on a processor.

Figure 21.19 RTOS actions required to start a process

The process manager has to manage processes with different priorities. For some stimuli, such as those associated with certain exceptional events, it is essential that their processing should be completed within the specified time limits. Other processes may be safely delayed if a more critical process requires service. Consequently, the RTOS has to be able to manage at least two priority levels for system processes:

1. *Clock level* This level of priority is allocated to periodic processes.
2. *Interrupt level* This is the highest priority level. It is allocated to processes that need a very fast response. One of these processes will be the real-time clock process. This process is not required if interrupts are not supported in the system.

A further priority level may be allocated to background processes (such as a self-checking process) that do not need to meet real-time deadlines. These processes are scheduled for execution when processor capacity is available.

Periodic processes must be executed at specified time intervals for data acquisition and actuator control. In most real-time systems, there will be several types of periodic process. Using the timing requirements specified in the application program, the RTOS arranges the execution of periodic processes so that they can all meet their deadlines.

The actions taken by the operating system for periodic process management are shown in Figure 21.19. The scheduler examines the list of periodic processes and selects a process to be executed. The choice depends on the process priority, the process periods, the expected execution times, and the deadlines of the ready processes. Sometimes two processes with different deadlines should be executed at the same clock tick. In such a situation, one process must be delayed. Normally, the system will choose to delay the process with the longest deadline.

Processes that have to respond quickly to asynchronous events may be interrupt-driven. The computer's interrupt mechanism causes control to transfer to a predetermined memory location. This location contains an instruction to jump to a simple and fast interrupt service routine. The service routine disables further interrupts to avoid being interrupted itself. It then discovers the cause of the interrupt and initiates, with a high priority, a process to handle the stimulus causing the interrupt. In some high-speed data acquisition systems, the interrupt handler saves the data that the interrupt signaled was available in a buffer for later processing. Interrupts are then enabled again, and control is returned to the operating system.

At any one time several processes, all with different priorities, could be executed. The process scheduler implements system-scheduling policies that determine the order of process execution. There are two commonly used scheduling strategies:

1. *Nonpreemptive scheduling* After a process has been scheduled for execution, it runs to completion or until it is blocked for some reason, such as waiting for input. This can cause problems if there are processes with different priorities and a high-priority process has to wait for a low-priority process to finish.
2. *Preemptive scheduling* The execution of an executing process may be stopped if a higher-priority process requires service. The higher-priority process preempts the execution of the lower-priority process and is allocated to a processor.

Within these strategies, different scheduling algorithms have been developed. These include round-robin scheduling, where each process is executed in turn; rate monotonic scheduling, where the process with the shortest period (highest frequency) is given priority; and shortest deadline first scheduling, where the process in the queue with the shortest deadline is scheduled (Burns and Wellings 2009).

Information about the process to be executed is passed to the resource manager. The resource manager allocates memory and, in a multiprocessor system, also adds a processor to this process. The process is then placed on the "ready list," a list of processes that are ready for execution. When a processor finishes executing a process and becomes available, the dispatcher is invoked. It scans the ready list to find a process that can be executed on the available processor and starts its execution.

KEY POINTS

- An embedded software system is part of a hardware/software system that reacts to events in its environment. The software is "embedded" in the hardware. Embedded systems are normally real-time systems.
- A real-time system is a software system that must respond to events in real time. System correctness does not just depend on the results it produces, but also on the time when these results are produced.
- Real-time systems are usually implemented as a set of communicating processes that react to stimuli to produce responses.
- State models are an important design representation for embedded real-time systems. They are used to show how the system reacts to its environment as events trigger changes of state in the system.
- Several standard patterns can be observed in different types of embedded system. These include a pattern for monitoring the system's environment for adverse events, a pattern for actuator control, and a data-processing pattern.

- Designers of real-time systems have to do a timing analysis, which is driven by the deadlines for processing and responding to stimuli. They have to decide how often each process in the system should run and the expected and worst-case execution time for processes.
- A real-time operating system is responsible for process and resource management. It always includes a scheduler, which is the component responsible for deciding which process should be scheduled for execution.

FURTHER READING

Real-time Systems and Programming Language: Ada, Real-time Java and C/Real-time POSIX, 4th ed. An excellent and comprehensive text that provides broad coverage of all aspects of real-time systems. (A. Burns and A. Wellings, Addison-Wesley, 2009).

"Trends in Embedded Software Engineering." This article suggests that model-driven development (as discussed in Chapter 5 of this book) will become an important approach to embedded systems development. This is part of a special issue on embedded systems, and other articles, such as the one by Ebert and Jones, are also useful reading. (*IEEE Software,* 26 (3), May–June 2009). http://dx.doi.org/10.1109/MS.2009.80

Real-time systems: Design Principles for Distributed Embedded Applications, 2nd ed. This is a comprehensive textbook on modern real-time systems that may be distributed and mobile systems. The author focuses on hard real-time systems and covers important topics such as Internet connectivity and power management. (H. Kopetz, Springer, 2013).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/systems-engineering/

EXERCISES

21.1. Explain why responsiveness in real time is the critical difference between embedded systems and other software systems.

21.2. Identify possible stimuli and the expected responses for an embedded system that controls a home refrigerator or a domestic washing machine.

21.3. Using the state-based approach to modeling, as discussed in Section 21.1.1, model the operation of the embedded software for a voicemail system that is included in a landline phone.

Train protection system

- The system acquires information on the speed limit of a segment from a trackside transmitter, which continually broadcasts the segment identifier and its speed limit. The same transmitter also broadcasts information on the status of the signal controlling that track segment. The time required to broadcast track segment and signal information is 50 ms.
- The train can receive information from the trackside transmitter when it is within 10 m of a transmitter.
- The maximum train speed is 180 kph.
- Sensors on the train provide information about the current train speed (updated every 250 ms) and the train brake status (updated every 100 ms).
- If the train speed exceeds the current segment speed limit by more than 5 kph, a warning is sounded in the driver's cabin. If the train speed exceeds the current segment speed limit by more than 10 kph, the train's brakes are automatically applied until the speed falls to the segment speed limit. Train brakes should be applied within 100 ms of the time when the excessive train speed has been detected.
- If the train enters a track segment that is signaled with a red light, the train protection system applies the train brakes and reduces the speed to zero. Train brakes should be applied within 100 ms of the time when the red light signal is received.
- The system continually updates a status display in the driver's cabin.

Figure 21.20 Requirements for a train protection system

This should display the number of recorded messages on an LED display and should allow the user to dial-in and listen to the recorded messages.

21.4. What are the commonly used architectural patterns in real-time systems and when are they used?

21.5. Show how the Environmental Control pattern could be used as the basis of the design of a system to control the temperature in a greenhouse. The temperature should be between 10 and 30 degrees Celsius. If it falls below 10 degrees, the heating system should be switched on; if it goes above 30, the windows should be automatically opened.

21.6. Design a process architecture for an environmental monitoring system that collects data from a set of air quality sensors situated around a city. There are 5000 sensors organized into 100 neighborhoods. Each sensor must be interrogated four times per second. When more than 30% of the sensors in a particular neighborhood indicate that the air quality is below an acceptable level, local warning lights are activated. All sensors return the readings to a central computer, which generates reports every 15 minutes on the air quality in the city.

21.7. A train protection system automatically applies the brakes of a train if the speed limit for a segment of track is exceeded or if the train enters a track segment that is currently signaled with a red light (i.e., the segment should not be entered). Details are shown in Figure 21.20. Identify the stimuli that must be processed by the on-board train control system and the associated responses to these stimuli.

21.8. Suggest a possible process architecture for this system.

21.9. If a periodic process in the on-board train protection system is used to collect data from the trackside transmitter, how often must it be scheduled to ensure that the system is guaranteed to collect information from the transmitter? Explain how you arrived at your answer.

21.10. With the help of examples, define what a real-time operating system is. Explain how it is different from a conventional operating system. What are the components included in real-time operating systems and what are their responsibilities?

REFERENCES

Berry, G. 1989. "Real-Time Programming: Special-Purpose or General-Purpose Languages." In *Information Processing*, edited by G. Ritter, 89:11–17. Amsterdam: Elsevier Science Publishers.

Bruno, E. J., and G. Bollella. 2009. *Real-Time Java Programming: With Java RTS*. Boston: Prentice-Hall.

Burns, A., and A. Wellings. 2009. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Boston: Addison-Wesley.

Cooling, J. 2003. *Software Engineering for Real-Time Systems*. Harlow, UK: Addison-Wesley.

Douglass, B. P. 1999. *Real-Time UML: Developing Efficient Objects for Embedded Systems, 2nd ed.* Boston: Addison-Wesley.

———. 2002. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston: Addison-Wesley.

Ebert, C., and C. Jones. 2009. "Embedded Software: Facts, Figures and Future." *IEEE Computer* 26 (3): 42–52. doi:10.1109/MC.2009.118.

Harel, D. 1987. "Statecharts: A Visual Formalism for Complex Systems." *Sci. Comput. Programming* 8 (3): 231–274. doi:10.1016/0167-6423(87)90035-9.

———. 1988. "On Visual Formalisms." *Comm. ACM* 31 (5): 514–530. doi:10.1145/42411.42414.

Lee, E A. 2002. "Embedded Software." In *Advances in Computers*, edited by M. Zelkowitz. Vol. 56. London: Academic Press.

Silberschaltz, A., P. B. Galvin, and G. Gagne. 2013. *Operating System Concepts, 9th ed.* New York: John Wiley & Sons.

Stallings, W. 2014. *Operating Systems: Internals and Design Principles, 8th ed.* Boston: Prentice-Hall.

This page intentionally left blank

PART 4 Software Management

It is sometimes suggested that the key difference between software engineering and other types of programming is that software engineering is a managed process. By this, I mean that the software development takes place within an organization and is subject to a range of schedule, budget and organizational constraints. I introduce a range of management topics in this part of the book with a focus on technical management issues rather than 'softer' management issues such as people management, or the more strategic management of enterprise systems.

Chapters 22 and 23 focus on the essential project management activities, planning, risk management and people management. Chapter 22 introduces software project management and its first major section is concerned with risk management where managers identify what might go wrong and plan what they might do about it. This chapter also includes sections on people management and team working.

Chapter 23 covers project planning and estimation. I introduce bar charts as fundamental planning tools and explain why plan-driven development will remain an important development approach, in spite of the success of agile methods. I also discuss issues that influence the price charged for a system and techniques of software cost estimation. I use the COCOMO family of cost models to describe algorithmic cost modeling and explain the benefits and disadvantages of algorithmic approaches.

Chapters 24 explains the basics of software quality management, as practised in large projects. Quality management is concerned with processes and techniques for ensuring and improving the quality of software. I discuss the importance of standards in quality management, the use of reviews and inspections in the quality assurance process. The final section of this chapter covers software measurement and I discuss the benefits and problems in using metrics and software data analytics in quality management.

Finally, Chapter 25 discusses configuration management, a critical issue for all large systems. However, the need for configuration management is not always obvious to students who have only been concerned with personal software development, so I describe the various aspects of this topic here, including version management, system building, change management and release management. I explain why continuous integration or daily system building is important. An important change in this edition is the inclusion of new material on distributed version management systems, such as Git, which are being increasingly used to support software engineering by distributed teams.

22

Project management

Objectives

The objective of this chapter is to introduce software project management and two important management activities, namely, risk management and people management. When you have read the chapter you will:

- know the principal tasks of software project managers;
- have been introduced to the notion of risk management and some of the risks that can arise in software projects;
- understand factors that influence personal motivation and what these might mean for software project managers;
- understand key issues that influence team working, such as team composition, organization, and communication.

Contents

22.1 Risk management
22.2 Managing people
22.3 Teamwork

Software project management is an essential part of software engineering. Projects need to be managed because professional software engineering is always subject to organizational budget and schedule constraints. The project manager's job is to ensure that the software project meets and overcomes these constraints as well as delivering high-quality software. Good management cannot guarantee project success. However, bad management usually results in project failure: The software may be delivered late, cost more than originally estimated, or fail to meet the expectations of customers.

The success criteria for project management obviously vary from project to project, but, for most projects, important goals are:

- to deliver the software to the customer at the agreed time;
- to keep overall costs within budget;
- to deliver software that meets the customer's expectations;
- to maintain a coherent and well-functioning development team.

These goals are not unique to software engineering but are the goals of all engineering projects. However, software engineering is different from other types of engineering in a number of ways that make software management particularly challenging. Some of these differences are:

1. *The product is intangible* A manager of a shipbuilding or a civil engineering project can see the product being developed. If a schedule slips, the effect on the product is visible—parts of the structure are obviously unfinished. Software is intangible. It cannot be seen or touched. Software project managers cannot see progress by looking at the artifact that is being constructed. Rather, they rely on others to produce evidence that they can use to review the progress of the work.
2. *Large software projects are often "one-off" projects* Every large software development project is unique because every environment where software is developed is, in some ways, different from all others. Even managers who have a large body of previous experience may find it difficult to anticipate problems. Furthermore, rapid technological changes in computers and communications can make experience obsolete. Lessons learned from previous projects may not be readily transferable to new projects.
3. *Software processes are variable and organization-specific* The engineering process for some types of system, such as bridges and buildings, is well understood. However, different companies use quite different software development processes. We cannot reliably predict when a particular software process is likely to lead to development problems. This is especially true when the software project is part of a wider systems engineering project or when completely new software is being developed.

Because of these issues, it is not surprising that some software projects are late, overbudget, and behind schedule. Software systems are often new, very complex, and technically innovative. Schedule and cost overruns are also common in other

engineering projects, such as new transport systems, that are complex and innovative. Given the difficulties involved, it is perhaps remarkable that so many software projects are delivered on time and to budget.

It is impossible to write a standard job description for a software project manager. The job varies tremendously depending on the organization and the software being developed. Some of the most important factors that affect how software projects are managed are:

1. *Company size* Small companies can operate with informal management and team communications and do not need formal policies and management structures. They have less management overhead than larger organizations. In larger organizations, management hierarchies, formal reporting and budgeting, and approval processes must be followed.
2. *Software customers* If the customer is an internal customer (as is the case for software product development), then customer communications can be informal and there is no need to fit in with the customer's ways of working. If custom software is being developed for an external customer, agreement has to be reached on more formal communication channels. If the customer is a government agency, the software company must operate according to the agency's policies and procedures, which are likely to be bureaucratic.
3. *Software size* Small systems can be developed by a small team, which can get together in the same room to discuss progress and other management issues. Large systems usually need multiple development teams that may be geographically distributed and in different companies. The project manager has to coordinate the activities of these teams and arrange for them to communicate with each other.
4. *Software type* If the software being developed is a consumer product, formal records of project management decisions are unnecessary. On the other hand, if a safety-critical system is being developed, all project management decisions should be recorded and justified as these may affect the safety of the system.
5. *Organizational culture* Some organizations have a culture that is based on supporting and encouraging individuals, while others are group focused. Large organizations are often bureaucratic. Some organizations have a culture of taking risks, whereas others are risk averse.
6. *Software development processes* Agile processes typically try to operate with "lightweight" management. More formal processes require management monitoring to ensure that the development team is following the defined process.

These factors mean that project managers in different organizations may work in quite different ways. However, a number of fundamental project management activities are common to all organizations:

1. *Project planning* Project managers are responsible for planning, estimating, and scheduling project development and assigning people to tasks. They supervise

the work to ensure that it is carried out to the required standards, and they monitor progress to check that the development is on time and within budget.

2. *Risk management* Project managers have to assess the risks that may affect a project, monitor these risks, and take action when problems arise.

3. *People management* Project managers are responsible for managing a team of people. They have to choose people for their team and establish ways of working that lead to effective team performance.

4. *Reporting* Project managers are usually responsible for reporting on the progress of a project to customers and to the managers of the company developing the software. They have to be able to communicate at a range of levels, from detailed technical information to management summaries. They have to write concise, coherent documents that abstract critical information from detailed project reports. They must be able to present this information during progress reviews.

5. *Proposal writing* The first stage in a software project may involve writing a proposal to win a contract to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out. It usually includes cost and schedule estimates and justifies why the project contract should be awarded to a particular organization or team. Proposal writing is a critical task as the survival of many software companies depends on having enough proposals accepted and contracts awarded.

Project planning is an important topic in its own right, which I discuss in Chapter 23. In this chapter, I focus on risk management and people management.

22.1 Risk management

Risk management is one of the most important jobs for a project manager. You can think of a risk as something that you'd prefer not to have happen. Risks may threaten the project, the software that is being developed, or the organization. Risk management involves anticipating risks that might affect the project schedule or the quality of the software being developed, and then taking action to avoid these risks (Hall 1998; Ould 1999).

Risks can be categorized according to type of risk (technical, organizational, etc.), as I explain in Section 22.1.1. A complementary classification is to classify risks according to what these risks affect:

1. Project risks affect the project schedule or resources. An example of a project risk is the loss of an experienced system architect. Finding a replacement architect with appropriate skills and experience may take a long time; consequently, it will take longer to develop the software design than originally planned.

2. Product risks affect the quality or performance of the software being developed. An example of a product risk is the failure of a purchased component to perform

as expected. This may affect the overall performance of the system so that it is slower than expected.

3. Business risks affect the organization developing or procuring the software. For example, a competitor introducing a new product is a business risk. The introduction of a competitive product may mean that the assumptions made about sales of existing software products may be unduly optimistic.

Of course, these risk categories overlap. An experienced engineer's decision to leave a project, for example, presents a *project risk* because the software delivery schedule will be affected. It inevitably takes time for a new project member to understand the work that has been done, so he or she cannot be immediately productive. Consequently, the delivery of the system may be delayed. The loss of a team member can also be a *product risk* because a replacement may not be as experienced and so could make programming errors. Finally, losing a team member can be a *business risk* because an experienced engineer's reputation may be a critical factor in winning new contracts.

For large projects, you should record the results of the risk analysis in a risk register along with a consequence analysis. This sets out the consequences of the risk for the project, product, and business. Effective risk management makes it easier to cope with problems and to ensure that these do not lead to unacceptable budget or schedule slippage. For small projects, formal risk recording may not be required, but the project manager should be aware of them.

The specific risks that may affect a project depend on the project and the organizational environment in which the software is being developed. However, there are also common risks that are independent of the type of software being developed. These can occur in any software development project. Some examples of these common risks are shown in Figure 22.1.

Software risk management is important because of the inherent uncertainties in software development. These uncertainties stem from loosely defined requirements, requirements changes due to changes in customer needs, difficulties in estimating the time and resources required for software development, and differences in individual skills. You have to anticipate risks, understand their impact on the project, the product, and the business, and take steps to avoid these risks. You may need to draw up contingency plans so that, if the risks do occur, you can take immediate recovery action.

An outline of the process of risk management is presented in Figure 22.2. It involves several stages:

1. *Risk identification* You should identify possible project, product, and business risks.

2. *Risk analysis* You should assess the likelihood and consequences of these risks.

3. *Risk planning* You should make plans to address the risk, either by avoiding it or by minimizing its effects on the project.

4. *Risk monitoring* You should regularly assess the risk and your plans for risk mitigation and revise these plans when you learn more about the risk.

Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of company management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
Software tool underperformance	Product	Software tools that support the project do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

Figure 22.1 Examples of common project, product, and business risks

Figure 22.2 The risk management process

For large projects, you should document the outcomes of the risk management process in a risk management plan. This should include a discussion of the risks faced by the project, an analysis of these risks, and information on how you plan to manage the risk if it seems likely to be a problem.

The risk management process is an iterative process that continues throughout a project. Once you have drawn up an initial risk management plan, you monitor the situation to detect emerging risks. As more information about the risks becomes

available, you have to re-analyze the risks and decide if the risk priority has changed. You may then have to change your plans for risk avoidance and contingency management.

Risk management in agile development is less formal. The same fundamental activities should still be followed and risks discussed, although these may not be formally documented. Agile development reduces some risks, such as risks from requirements changes. However, agile development also has a downside. Because of its reliance on people, staff turnover can have significant effects on the project, product, and business. Because of the lack of formal documentation and its reliance on informal communications, it is very hard to maintain continuity and momentum if key people leave the project.

22.1.1 Risk identification

Risk identification is the first stage of the risk management process. It is concerned with identifying the risks that could pose a major threat to the software engineering process, the software being developed, or the development organization. Risk identification may be a team process in which a team gets together to brainstorm possible risks. Alternatively, project managers may identify risks based on their experience of what went wrong on previous projects.

As a starting point for risk identification, a checklist of different types of risk may be used. Six types of risk may be included in a risk checklist:

1. Estimation risks arise from the management estimates of the resources required to build the system.
2. Organizational risks arise from the organizational environment where the software is being developed.
3. People risks are associated with the people in the development team.
4. Requirements risks come from changes to the customer requirements and the process of managing the requirements change.
5. Technology risks come from the software or hardware technologies that are used to develop the system.
6. Tools risks come from the software tools and other support software used to develop the system.

Figure 22.3 shows examples of possible risks in each of these categories. When you have finished the risk identification process, you should have a long list of risks that could occur and that could affect the product, the process, and the business. You then need to prune this list to a manageable size. If you have too many risks, it is practically impossible to keep track of all of them.

Risk type	Possible risks
Estimation	1. The time required to develop the software is underestimated. 2. The rate of defect repair is underestimated. 3. The size of the software is underestimated.
Organizational	4. The organization is restructured so that different management are responsible for the project. 5. Organizational financial problems force reductions in the project budget.
People	6. It is impossible to recruit staff with the skills required. 7. Key staff are ill and unavailable at critical times. 8. Required training for staff is not available.
Requirements	9. Changes to requirements that require major design rework are proposed. 10. Customers fail to understand the impact of requirements changes.
Technology	11. The database used in the system cannot process as many transactions per second as expected. 12. Faults in reusable software components have to be repaired before these components are reused.
Tools	13. The code generated by software code generation tools is inefficient. 14. Software tools cannot work together in an integrated way.

Figure 22.3 Examples of different types of risk

22.1.2 Risk analysis

During the risk analysis process, you have to consider each identified risk and make a judgment about the probability and seriousness of that risk. There is no easy way to do so. You have to rely on your judgment and experience of previous projects and the problems that arose in them. It is not possible to make precise, numeric assessment of the probability and seriousness of each risk. Rather, you should assign the risk to one of a number of bands:

1. The probability of the risk might be assessed as insignificant, low, moderate, high, or very high.
2. The effects of the risk might be assessed as catastrophic (threaten the survival of the project), serious (would cause major delays), tolerable (delays are within allowed contingency), or insignificant.

You may then tabulate the results of this analysis process using a table ordered according to the seriousness of the risk. Figure 22.4 illustrates this for the risks that I have identified in Figure 22.3. Obviously, the assessment of probability and seriousness is arbitrary here. To make this assessment, you need

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget (5).	Low	Catastrophic
It is impossible to recruit staff with the skills required (6).	High	Catastrophic
Key staff are ill at critical times in the project (7).	Moderate	Serious
Faults in reusable software components have to be repaired before these components are reused (12).	Moderate	Serious
Changes to requirements that require major design rework are proposed (9).	Moderate	Serious
The organization is restructured so that different managements are responsible for the project (4).	High	Serious
The database used in the system cannot process as many transactions per second as expected (11).	Moderate	Serious
The time required to develop the software is underestimated (1).	High	Serious
Software tools cannot be integrated (14).	High	Tolerable
Customers fail to understand the impact of requirements changes (10).	Moderate	Tolerable
Required training for staff is not available (8).	Moderate	Tolerable
The rate of defect repair is underestimated (2).	Moderate	Tolerable
The size of the software is underestimated (3).	High	Tolerable
Code generated by code generation tools is inefficient (13).	Moderate	Insignificant

Figure 22.4 Risk types and examples

detailed information about the project, the process, the development team, and the organization.

Of course, both the probability and the assessment of the effects of a risk may change as more information about the risk becomes available and as risk management plans are implemented. You should therefore update this table during each iteration of the risk management process.

Once the risks have been analyzed and ranked, you should assess which of these risks are most significant. Your judgment must depend on a combination of the probability of the risk arising and the effects of that risk. In general, catastrophic risks should always be considered, as should all serious risks that have more than a moderate probability of occurrence.

Boehm (Boehm 1988) recommends identifying and monitoring the "top 10" risks. However, I think that the right number of risks to monitor must depend on the project. It might be 5 or it might be 15. From the risks identified in Figure 22.4, I think that it is appropriate to consider the eight risks that have catastrophic or serious consequences (Figure 22.5).

22.1.3 Risk planning

The risk planning process develops strategies to manage the key risks that threaten the project. For each risk, you have to think of actions that you might take to minimize the disruption to the project if the problem identified in the risk occurs. You should also think about the information that you need to collect while monitoring the project so that emerging problems can be detected before they become serious.

In risk planning, you have to ask "what-if" questions that consider both individual risks, combinations of risks, and external factors that affect these risks. For example, questions that you might ask are:

1. What if several engineers are ill at the same time?
2. What if an economic downturn leads to budget cuts of 20% for the project?
3. What if the performance of open-source software is inadequate and the only expert on that open-source software leaves?
4. What if the company that supplies and maintains software components goes out of business?
5. What if the customer fails to deliver the revised requirements as predicted?

Based on the answers to these "what-if" questions, you may devise strategies for managing the risks. Figure 22.5 shows possible risk management strategies that have been identified for the key risks (i.e., those that are serious or intolerable) shown in Figure 22.4. These strategies fall into three categories:

1. *Avoidance strategies* Following these strategies means that the probability that the risk will arise is reduced. An example of a risk avoidance strategy is the strategy for dealing with defective components shown in Figure 22.5.
2. *Minimization strategies* Following these strategies means that the impact of the risk is reduced. An example of a risk minimization strategy is the strategy for staff illness shown in Figure 22.5.
3. *Contingency plans* Following these strategies means that you are prepared for the worst and have a strategy in place to deal with it. An example of a contingency strategy is the strategy for organizational financial problems that I have shown in Figure 22.5.

You can see a clear analogy here with the strategies used in critical systems to ensure reliability, security, and safety, where you must avoid, tolerate, or recover from failures. Obviously, it is best to use a strategy that avoids the risk. If this is not possible, you should use a strategy that reduces the chances that the risk will have serious effects. Finally, you should have strategies in place to

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of automated code generation.

Figure 22.5 Strategies to help manage risk

cope with the risk if it arises. These should reduce the overall impact of a risk on the project or product.

22.1.4 Risk monitoring

Risk monitoring is the process of checking that your assumptions about the product, process, and business risks have not changed. You should regularly assess each of the identified risks to decide whether or not that risk is becoming more or less probable. You should also think about whether or not the effects of the risk have changed. To do this, you have to look at other factors, such as the number of requirements change requests, which give you clues about the risk probability and its effects. These factors are obviously dependent on the types of risk. Figure 22.6 gives some examples of factors that may be helpful in assessing these risk types.

You should monitor risks regularly at all stages in a project. At every management review, you should consider and discuss each of the key risks separately. You should decide if the risk is more or less likely to arise and if the seriousness and consequences of the risk have changed.

Risk type	Potential indicators
Estimation	Failure to meet agreed schedule; failure to clear reported defects.
Organizational	Organizational gossip; lack of action by senior management.
People	Poor staff morale; poor relationships among team members; high staff turnover.
Requirements	Many requirements change requests; customer complaints.
Technology	Late delivery of hardware or support software; many reported technology problems.
Tools	Reluctance by team members to use tools; complaints about software tools; requests for faster computers/more memory, and so on.

Figure 22.6 Risk indicators

22.2 Managing people

The people working in a software organization are its greatest assets. It is expensive to recruit and retain good people, and it is up to software managers to ensure that the engineers working on a project are as productive as possible. In successful companies and economies, this productivity is achieved when people are respected by the organization and are assigned responsibilities that reflect their skills and experience.

It is important that software project managers understand the technical issues that influence the work of software development. Unfortunately, however, good software engineers are not always good people managers. Software engineers often have strong technical skills but may lack the softer skills that enable them to motivate and lead a project development team. As a project manager, you should be aware of the potential problems of people management and should try to develop people management skills.

There are four critical factors that influence the relationship between a manager and the people that he or she manages:

1. *Consistency* All the people in a project team should be treated in a comparable way. No one expects all rewards to be identical, but people should not feel that their contribution to the organization is undervalued.

2. *Respect* Different people have different skills, and managers should respect these differences. All members of the team should be given an opportunity to make a contribution. In some cases, of course, you will find that people simply don't fit into a team and they cannot continue, but it is important not to jump to conclusions about them at an early stage in the project.

3. *Inclusion* People contribute effectively when they feel that others listen to them and take account of their proposals. It is important to develop a working environment where all views, even those of the least experienced staff, are considered.

4. *Honesty* As a manager, you should always be honest about what is going well and what is going badly in the team. You should also be honest about your level of technical knowledge and be willing to defer to staff with more knowledge when necessary. If you try to cover up ignorance or problems, you will eventually be found out and will lose the respect of the group.

Practical people management has to be based on experiences so my aim in this section and the following section on teamwork is to raise awareness of the most important issues that project managers may have to deal with.

22.2.1 Motivating people

As a project manager, you need to motivate the people who work with you so that they will contribute to the best of their abilities. In practice, motivation means organizing work and its environment to encourage people to work as effectively as possible. If people are not motivated, they will be less interested in the work they are doing. They will work slowly, be more likely to make mistakes, and will not contribute to the broader goals of the team or the organization.

To provide this encouragement, you should understand a little about what motivates people. Maslow (Maslow 1954) suggests that people are motivated by satisfying their needs. These needs are arranged in a series of levels, as shown in Figure 22.7. The lower levels of this hierarchy represent fundamental needs for food, sleep, and so on, and the need to feel secure in an environment. Social need is concerned with the need to feel part of a social grouping. Esteem need represents the need to feel respected by others, and self-realization need is concerned with personal development. People need to satisfy lower-level needs such as hunger before the more abstract, higher-level needs.

People working in software development organizations are not usually hungry, thirsty, or physically threatened by their environment. Therefore, making sure that peoples' social, esteem, and self-realization needs are satisfied is most important from a management point of view.

1. To satisfy social needs, you need to give people time to meet their co-workers and provide places for them to meet. Software companies such as Google provide social space in their offices for people to get together. This is relatively easy when all of the members of a development team work in the same place, but, increasingly, team members are not located in the same building or even the same town or state. They may work for different organizations or from home most of the time.

Figure 22.7 Human needs hierarchy

Social networking systems and teleconferencing can be used for remote communications, but my experience with these systems is that they are most effective when people already know each other. You should arrange some face-to-face meetings early in the project so that people can directly interact with other members of the team. Through this direct interaction, people become part of a social group and accept the goals and priorities of that group.

2. To satisfy esteem needs, you need to show people that they are valued by the organization. Public recognition of achievements is a simple and effective way of doing this. Obviously, people must also feel that they are paid at a level that reflects their skills and experience.

3. Finally, to satisfy self-realization needs, you need to give people responsibility for their work, assign them demanding (but not impossible) tasks, and provide opportunities for training and development where people can enhance their skills. Training is an important motivating influence as people like to gain new knowledge and learn new skills.

Maslow's model of motivation is helpful up to a point, but I think that a problem with it is that it takes an exclusively personal viewpoint on motivation. It does not take adequate account of the fact that people feel themselves to be part of an organization, a professional group, and one or more cultures. Being a member of a cohesive group is highly motivating for most people. People with fulfilling jobs often like to go to work because they are motivated by the people they work with and the work that they do. Therefore, as a manager, you also have to think about how a group as a whole can be motivated. I discuss this and other teamwork issues in Section 22.3.

In Figure 22.8, I illustrate a problem of motivation that managers often have to face. In this example, a competent group member loses interest in the work and in the group as a whole. The quality of her work falls and becomes unacceptable. This situation has to be dealt with quickly. If you don't sort out the problem, the other group members will become dissatisfied and feel that they are doing an unfair share of the work.

Case study: Motivation

Alice is a software project manager working in a company that develops alarm systems. This company wishes to enter the growing market of assistive technology to help elderly and disabled people live independently. Alice has been asked to lead a team of six developers that can develop new products based on the company's alarm technology.

Alice's assistive technology project starts well. Good working relationships develop within the team, and creative new ideas are developed. The team decides to develop a system that a user can initiate and control the alarm system from a cell phone or tablet computer. However, some months into the project, Alice notices that Dorothy, a hardware expert, starts coming into work late, that the quality of her work is deteriorating, and, increasingly, that she does not appear to be communicating with other members of the team.

Alice talks about the problem informally with other team members to try to find out if Dorothy's personal circumstances have changed and if this might be affecting her work. They don't know of anything, so Alice decides to talk with Dorothy to try to understand the problem.

After some initial denials of any problem, Dorothy admits that she has lost interest in the job. She expected that she would be able to develop and use her hardware interfacing skills. However, because of the product direction that has been chosen, she has little opportunity to use these skills. Basically, she is working as a C programmer on the alarm system software.

While she admits that the work is challenging, she is concerned that she is not developing her interfacing skills. She is worried that finding a job that involves hardware interfacing will be difficult after this project. Because she does not want to upset the team by revealing that she is thinking about the next project, she has decided that it is best to minimize conversation with them.

Figure 22.8 Individual motivation

In this example, Alice tries to find out if Dorothy's personal circumstances could be the problem. Personal difficulties commonly affect motivation because people cannot therefore concentrate on their work. You may have to give them time and support to resolve these issues, although you also have to make it clear that they still have a responsibility to their employer.

Dorothy's motivation problem is one that can arise when projects develop in an unexpected direction. People who expect to do one type of work may end up doing something completely different. In those circumstances, you may decide that the team member should leave the team and find opportunities elsewhere. In this example, however, Alice decides to try to convince Dorothy that broadening her experience is a positive career step. She gives Dorothy more design autonomy and organizes training courses in software engineering that will give her more opportunities after her current project has finished.

Psychological personality type also influences motivation. Bass and Dunteman (Bass and Dunteman 1963) identified three classifications for professional workers:

1. *Task-oriented people,* who are motivated by the work they do. In software engineering, these are people who are motivated by the intellectual challenge of software development.

The People Capability Maturity Model

The People Capability Maturity Model (P-CMM) is a framework for assessing how well organizations manage the development of their staff. It highlights best practice in people management and provides a basis for organizations to improve their people management processes. It is best suited to large rather than small, informal companies.

http://software-engineering-book.com/web/people-cmm/

2. *Self-oriented people,* who are principally motivated by personal success and recognition. They are interested in software development as a means of achieving their own goals. They often have longer-term goals, such as career progression, that motivate them, and they wish to be successful in their work to help realize these goals.
3. *Interaction-oriented people,* who are motivated by the presence and actions of co-workers. As more and more attention is paid to user interface design, interaction-oriented individuals are becoming more involved in software engineering.

Research has shown that interaction-oriented personalities usually like to work as part of a group, whereas task-oriented and self-oriented people usually prefer to act as individuals. Women are more likely to be interaction-oriented than men are. They are often more effective communicators. I discuss the mix of these different personality types in groups in the case study shown later in Figure 22.10.

Each individual's motivation is made up of elements of each class, but one type of motivation is usually dominant at any one time. However, individuals can change. For example, technical people who feel they are not being properly rewarded can become self-oriented and put personal interests before technical concerns. If a group works particularly well, self-oriented people can become more interaction-oriented.

22.3 Teamwork

Most professional software is developed by project teams that range in size from two to several hundred people. However, as it is impossible for everyone in a large group to work together on a single problem, large teams are usually split into a number of smaller groups. Each group is responsible for developing part of the overall system. The best size for a software engineering group is 4 to 6 members, and they should never have more than 12 members. When groups are small, communication problems are reduced. Everyone knows everyone else, and the whole group can get around a table for a meeting to discuss the project and the software that they are developing.

Putting together a group that has the right balance of technical skills, experience, and personalities is a critical management task. However, successful groups are more than simply a collection of individuals with the right balance of skills. A good group is cohesive and thinks of itself as a strong, single unit. The people involved are motivated by the success of the group as well as by their own personal goals.

In a cohesive group, members think of the group as more important than the individuals who are group members. Members of a well-led, cohesive group are loyal to the group. They identify with group goals and other group members. They attempt to protect the group, as an entity, from outside interference. This makes the group robust and able to cope with problems and unexpected situations.

The benefits of creating a cohesive group are:

1. *The group can establish its own quality standards* Because these standards are established by consensus, they are more likely to be observed than external standards imposed on the group.
2. *Individuals learn from and support each other* Group members learn by working together. Inhibitions caused by ignorance are minimized as mutual learning is encouraged.
3. *Knowledge is shared* Continuity can be maintained if a group member leaves. Others in the group can take over critical tasks and ensure that the project is not unduly disrupted.
4. *Refactoring and continual improvement is encouraged* Group members work collectively to deliver high-quality results and fix problems, irrespective of the individuals who originally created the design or program.

Good project managers should always try to encourage group cohesiveness. They may try to establish a sense of group identity by naming the group and establishing a group identity and territory. Some managers like explicit group-building activities such as sports and games, although these are not always popular with group members. Social events for group members and their families are a good way to bring people together.

One of the most effective ways of promoting cohesion is to be inclusive. That is, you should treat group members as responsible and trustworthy, and make information freely available. Sometimes managers feel that they cannot reveal certain information to everyone in the group. This invariably creates a climate of mistrust. An effective way of making people feel valued and part of a group is to make sure that they know what is going on.

You can see an example in the case study in Figure 22.9. Alice arranges regular informal meetings where she tells the other group members what is going on. She makes a point of involving people in the product development by asking them to come up with new ideas derived from their own family experiences. The "away

Case study: Team spirit

Alice, an experienced project manager, understands the importance of creating a cohesive group. As her company is developing a new product, she takes the opportunity to involve all group members in the product specification and design by getting them to discuss possible technology with elderly members of their families. She encourages them to bring these family members to meet other members of the development group.

Alice also arranges monthly lunches for everyone in the group. These lunches are an opportunity for all team members to meet informally, talk around issues of concern, and get to know each other. At the lunch, Alice tells the group what she knows about organizational news, policies, strategies, and so forth. Each team member then briefly summarizes what they have been doing, and the group discusses a general topic, such as new product ideas from elderly relatives.

Every few months, Alice organizes an "away day" for the group where the team spends two days on "technology updating." Each team member prepares an update on a relevant technology and presents it to the group. This is an offsite meeting, and plenty of time is scheduled for discussion and social interaction.

Figure 22.9 Group cohesion

days" are also good ways of promoting cohesion: People relax together while they help each other learn about new technologies.

Whether or not a group is effective depends, to some extent, on the nature of the project and the organization doing the work. If an organization is in a state of turmoil with constant reorganizations and job insecurity, it is difficult for team members to focus on software development. Similarly, if a project keeps changing and is in danger of cancellation, people lose interest in it.

Given a stable organizational and project environment, the three factors that have the biggest effect on team working are:

1. *The people in the group* You need a mix of people in a project group as software development involves diverse activities such as negotiating with clients, programming, testing, and documentation.
2. *The way the group is organized* A group should be organized so that individuals can contribute to the best of their abilities and tasks can be completed as expected.
3. *Technical and managerial communications* Good communication between group members, and between the software engineering team and other project stakeholders, is essential.

As with all management issues, getting the right team cannot guarantee project success. Too many other things can go wrong, including changes to the business and the business environment. However, if you don't pay attention to group composition, organization, and communications, you increase the likelihood that your project will run into difficulties.

22.3.1 Selecting group members

A manager or team leader's job is to create a cohesive group and organize that group so that they work together effectively. This task involves selecting a group with the right balance of technical skills and personalities. Sometimes people are hired from outside the organization; more often, software engineering groups are put together from current employees who have experience on other projects. Managers rarely have a completely free hand in team selection. They often have to use the people who are available in the company, even if they are not the ideal people for the job.

Many software engineers are motivated primarily by their work. Software development groups, therefore, are often composed of people who have their own ideas about how technical problems should be solved. They want to do the best job possible, so they may deliberately redesign systems that they think can be improved and add extra system features that are not in the system requirements. Agile methods encourage engineers to take the initiative to improve the software. However, sometimes this means that time is spent doing things that aren't really needed and that different engineers compete to rewrite each other's code.

Technical knowledge and ability should not be the only factor used to select group members. The "competing engineers" problem can be reduced if the people in the group have complementary motivations. People who are motivated by the work are likely to be the strongest technically. People who are self-oriented will probably be best at pushing the work forward to finish the job. People who are interaction-oriented help facilitate communications within the group. I think that it is particularly important to have interaction-oriented people in a group. They like to talk to people and can detect tensions and disagreements at an early stage, before these problems have a serious impact on the group.

In the case study in Figure 22.10, I have suggested how Alice, the project manager, has tried to create a group with complementary personalities. This particular group has a good mix of interaction- and task-oriented people, but I have already discussed, in Figure 22.8, how Dorothy's self-oriented personality has caused problems because she has not been doing the work that she expected. Fred's part-time role in the group as a domain expert might also be a problem. He is mostly interested in technical challenges, so he may not interact well with other group members. The fact that he is not always part of the team means that he may not fully relate to the team's goals.

It is sometimes impossible to choose a group with complementary personalities. If this is the case, the project manager has to control the group so that individual goals do not take precedence over organizational and group objectives. This control is easier to achieve if all group members participate in each stage of the project. Individual initiative is most likely to develop when group members are given instructions without being aware of the part that their task plays in the overall project.

For example, say a software engineer takes over the development of a system and notices that possible improvements could be made to the design. If he or she implements these improvements without understanding the rationale for the original design, any changes, though well-intentioned, might have adverse implications for

Case study: Group composition

In creating a group for assistive technology development, Alice is aware of the importance of selecting members with complementary personalities. When interviewing potential group members, she tried to assess whether they were task-oriented, self-oriented, or interaction-oriented. She felt that she was primarily a self-oriented type because she considered the project to be a way of getting noticed by senior management and possibly being promoted. She therefore looked for one or perhaps two interaction-oriented personalities, with task-oriented individuals to complete the team. The final assessment that she arrived at was:

Alice—self-oriented
Brian—task-oriented
Chun—interaction-oriented
Dorothy—self-oriented
Ed—interaction-oriented
Fiona—task-oriented
Fred—task-oriented
Hassan—interaction-oriented

Figure 22.10 Group composition

other parts of the system. If all the members of the group are involved in the design from the start, they are more likely to understand why design decisions have been made. They may then identify with these decisions rather than oppose them.

22.2.3 Group organization

The way a group is organized affects the group's decisions, the ways information is exchanged, and the interactions between the development group and external project stakeholders. Important organizational questions for project managers include the following:

1. Should the project manager be the technical leader of the group? The technical leader or system architect is responsible for the critical technical decisions made during software development. Sometimes the project manager has the skill and experience to take on this role. However, for large projects, it is best to separate technical and managerial roles. The project manager should appoint a senior engineer to be the project architect, who will take responsibility for technical leadership.
2. Who will be involved in making critical technical decisions, and how will these decisions be made? Will decisions be made by the system architect or the project manager or by reaching consensus among a wider range of team members?
3. How will interactions with external stakeholders and senior company management be handled? In many cases, the project manager will be responsible for these interactions, assisted by the system architect if there is one. However, an alternative organizational model is to create a dedicated role concerned with external liaison and appoint someone with appropriate interaction skills to that role.

Hiring the right people

Project managers are often responsible for selecting the people in the organization who will join their software engineering team. Getting the best possible people in this process is very important as poor selection decisions may be a serious risk to the project.

Key factors that should influence the selection of staff are education and training, application domain and technology experience, communication ability, adaptability, and problem solving ability.

http://software-engineering-book.com/web/people-selection/

4. How can groups integrate people who are not co-located? It is now common for groups to include members from different organizations and for people to work from home as well as in a shared office. This change has to be considered in group decision-making processes.

5. How can knowledge be shared across the group? Group organization affects information sharing as certain methods of organization are better for sharing than others. However, you should avoid too much information sharing as people become overloaded and excessive information distracts them from their work.

Small programming groups are usually organized in an informal way. The group leader gets involved in the software development with the other group members. In an informal group, the group as a whole discusses the work to be carried out, and tasks are allocated according to ability and experience. More senior group members may be responsible for the architectural design. However, detailed design and implementation is the responsibility of the team member who is allocated to a particular task.

Agile development teams are always informal groups. Agile enthusiasts claim that formal structure inhibits information exchange. Many decisions that are usually seen as management decisions (such as decisions on schedule) may be devolved to group members. However, there still needs to be a project manager who is responsible for strategic decision making and communications outside of the group.

Informal groups can be very successful, particularly when most group members are experienced and competent. Such a group makes decisions by consensus, which improves cohesiveness and performance. However, if a group is composed mostly of inexperienced or incompetent members, informality can be a hindrance. With no experienced engineers to direct the work, the result can be a lack of coordination between group members and, possibly, eventual project failure.

In hierarchical groups the group leader is at the top of the hierarchy. He or she has more formal authority than the group members and so can direct their work. There is a clear organizational structure, and decisions are made toward the top of the hierarchy and implemented by people lower down. Communications are primarily instructions from senior staff; the people at lower levels of the hierarchy have relatively little communication with the managers at the upper levels.

Hierarchical groups can work well when a well-understood problem can be easily broken down into software components that can be developed in different parts of the hierarchy. This grouping allows for rapid decision making, which is why military organizations follow this model. However, it rarely works well for complex software engineering. In software development, effective team communications at all levels is essential:

1. Changes to the software often require changes to several parts of the system, and this requires discussion and negotiation at all levels in the hierarchy.
2. Software technologies change so fast that more junior staff may know more about new technologies than experienced staff. Top-down communications may mean that the project manager does not find out about the opportunities of using these new technologies. More junior staff may become frustrated because of what they see as old-fashioned technologies being used for development.

A major challenge facing project managers is the difference in technical ability between group members. The best programmers may be up to 25 times more productive than the worst programmers. It makes sense to use these "super-programmers" in the most effective way and to provide them with as much support as possible.

At the same time, focusing on the super-programmers can be demotivating for other group members who are resentful that they are not given responsibility. They may be concerned that this will affect their career development. Furthermore, if a "super-programmer" leaves the company, the impact on a project can be huge. Therefore, adopting a group model that is based on individual experts can pose significant risks.

22.3.3 Group communications

It is absolutely essential that group members communicate effectively and efficiently with each other and with other project stakeholders. Group members must exchange information on the status of their work, the design decisions that have been made, and changes to previous design decisions. They have to resolve problems that arise with other stakeholders and inform these stakeholders of changes to the system, the group, and delivery plans. Good communication also helps strengthen group cohesiveness. Group members come to understand the motivations, strengths, and weaknesses of other people in the group.

The effectiveness and efficiency of communications are influenced by:

1. *Group size* As a group gets bigger, it gets harder for members to communicate effectively. The number of one-way communication links is $n * (n - 1)$, where n is the group size, so, with a group of eight members, there are 56 possible communication pathways. This means that it is quite possible that some people will rarely communicate with each other. Status differences between group members mean that communications are often one-way. Managers and experienced engineers tend to dominate communications with less experienced staff, who may be reluctant to start a conversation or make critical remarks.

The physical work environment

Group communications and individual productivity are both affected by the team's working environment. Individual workspaces are better for concentration on detailed technical work as people are less likely to be distracted by interruptions. However, shared workspaces are better for communications. A well-designed work environment takes both of these needs into account.

http://software-engineering-book.com/web/workspace/

2. *Group structure* People in informally structured groups communicate more effectively than people in groups with a formal, hierarchical structure. In hierarchical groups, communications tend to flow up and down the hierarchy. People at the same level may not talk to each other. This is a particular problem in a large project with several development groups. If people working on different subsystems only communicate through their managers, then there are more likely to be delays and misunderstandings.

3. *Group composition* People with the same personality types (discussed in Section 22.2) may clash, and, as a result, communications can be inhibited. Communication is also usually better in mixed-sex groups than in single-sex groups (Marshall and Heslin 1975). Women are often more interaction-oriented than men and may act as interaction controllers and facilitators for the group.

4. *The physical work environment* The organization of the workplace is a major factor in facilitating or inhibiting communications. While some companies use standard open-plan offices for their staff, others invest in providing a workspace that includes a mixture of private and group working areas. This allows for both collaborative activities and individual development that require a high level of concentration.

5. *The available communication channels* There are many different forms of communication—face to face, email messages, formal documents, telephone, and technologies such as social networking and wikis. As project teams become increasingly distributed, with team members working remotely, you need to make use of interaction technologies, such as conferencing systems, to facilitate group communications.

Project managers usually work to tight deadlines, and, consequently, they often try to use communication channels that don't take up too much of their time. They may rely on meetings and formal documents to pass on information to project staff and stakeholders and send long emails to project staff. Unfortunately, while this may be an efficient approach to communication from a project manager's perspective, it is not usually very effective. There are often good reasons why people can't attend meetings, and so they don't hear the presentation. People do not have time to read long documents and emails that are not directly relevant to their work. When several versions of the same document are produced, readers find it difficult to keep track of the changes.

Effective communication is achieved when communications are two-way and the people involved can discuss issues and information and establish a common understanding of proposals and problems. All this can be done through meetings, although these meetings are often dominated by powerful personalities. Informal discussions when a manager meets with the team for coffee are sometimes more effective.

More and more project teams include remote members, which also makes meetings more difficult. To involve them in communications, you may make use of wikis and blogs to support information exchange. Wikis support the collaborative creation and editing of documents, and blogs support threaded discussions about questions and comments made by group members. Wikis and blogs allow project members and external stakeholders to exchange information, irrespective of their location. They help manage information and keep track of discussion threads, which often become confusing when conducted by email. You can also use instant messaging and teleconferences, which can be easily arranged, to resolve issues that need discussion.

KEY POINTS

- Good software project management is essential if software engineering projects are to be developed on schedule and within budget.
- Software management is distinct from other engineering management. Software is intangible. Projects may be novel or innovative, so there is no body of experience to guide their management. Software processes are not as mature as traditional engineering processes.
- Risk management involves identifying and assessing major project risks to establish the probability that they will occur and the consequences for the project if that risk does arise. You should make plans to avoid, manage, or deal with likely risks if or when they arise.
- People management involves choosing the right people to work on a project and organizing the team and its working environment so that they are as productive as possible.
- People are motivated by interaction with other people, by the recognition of management and their peers, and by being given opportunities for personal development.
- Software development groups should be fairly small and cohesive. The key factors that influence the effectiveness of a group are the people in that group, the way that it is organized, and the communication between group members.
- Communications within a group are influenced by factors such as the status of group members, the size of the group, the gender composition of the group, personalities, and available communication channels.

FURTHER READING

The Mythical Man Month: Essays on Software Engineering (Anniversary Edition). The problems of software management have remained largely unchanged since the 1960s, and this is one of the best books on the topic. It presents an interesting and readable account of the management of one of the first very large software projects, the IBM OS/360 operating system. The anniversary edition (published 20 years after the original edition in 1975) includes other classic papers by Brooks. (F. P. Brooks, 1995, Addison-Wesley).

Peopleware: Productive Projects and Teams, 2nd ed. This now classic book focuses on the importance of treating people properly when managing software projects. It is one of the few books that recognizes how the place where people work influences communications and productivity. Strongly recommended. (T. DeMarco and T. Lister, 1999, Dorset House).

Waltzing with Bears: Managing Risk on Software Projects. A very practical and easy-to-read introduction to risks and risk management. (T. DeMarco and T. Lister, 2003, Dorset House).

Effective Project Management: Traditional, Agile, Extreme. 2014 (7th ed.). This is a textbook on project management in general rather than software project management. It is based on the so-called PMBOK (Project Management Body of Knowledge) and, unlike most books on this topic, discusses PM techniques for agile projects. (R. K. Wysocki, 2014).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-management/

EXERCISES

22.1. Explain why the intangibility of software systems poses special problems for software project management.

22.2. Explain how company size and software size are factors that affect software project management.

22.3. Using reported instances of project problems in the literature, list management difficulties and errors that occurred in these failed programming projects. (I suggest that you start with *The Mythical Man Month*, as suggested in Further Reading.)

22.4. In addition to the risks shown in Figure 22.1, identify at least six other possible risks that could arise in software projects.

22.5. What is risk monitoring? How can risks be monitored? List a few examples of types of risks and their potential indicators.

22.6. Fixed-price contracts, where the contractor bids a fixed price to complete a system development, may be used to move project risk from client to contractor. If anything goes wrong, the contractor has to pay. Suggest how the use of such contracts may increase the likelihood that product risks will arise.

22.7. Explain why keeping all members of a group informed about progress and technical decisions in a project can improve group cohesiveness.

22.8. What qualities of a cohesive group's members make the group robust? List out the key benefits of creating a cohesive group.

22.9. Write a case study in the style used here to illustrate the importance of communications in a project team. Assume that some team members work remotely and that it is not possible to get the whole team together at short notice.

22.10. Your manager asks you to deliver software to a schedule that you know can only be met by asking your project team to work unpaid overtime. All team members have young children. Discuss whether you should accept this demand from your manager or whether you should persuade your team to give their time to the organization rather than to their families. What factors might be significant in your decision?

REFERENCES

Bass, B. M., and G. Dunteman. 1963. "Behaviour in Groups as a Function of Self, Interaction and Task Orientation." *J. Abnorm. Soc. Psychology*. 66 (4): 19–28. doi:10.1037/h0042764.

Boehm, B. W. 1988. "A Spiral Model of Software Development and Enhancement." *IEEE Computer* 21 (5): 61–72. doi:10.1109/2.59.

Hall, E. 1998. *Managing Risk: Methods for Software Systems Development*. Reading, MA: Addison-Wesley.

Marshall, J. E., and R. Heslin. 1975. "Boys and Girls Together. Sexual Composition and the Effect of Density on Group Size and Cohesiveness." *J. of Personality and Social Psychology* 35 (5): 952–961. doi:10.1037/h0076838.

Maslow, A. A. 1954. *Motivation and Personality*. New York: Harper & Row.

Ould, M. 1999. *Managing Software Quality and Business Risk*. Chichester, UK: John Wiley & Sons.

23

Project planning

Objectives

The objective of this chapter is to introduce project planning, scheduling, and cost estimation. When you have read the chapter, you will:

- understand the fundamentals of software costing and the factors that affect the price of a software system to be developed for external clients;
- know what sections should be included in a project plan that is created within a plan-driven development process;
- understand what is involved in project scheduling and the use of bar charts to present a project schedule;
- have been introduced to agile project planning based on the "planning game";
- understand cost estimation techniques and how the COCOMO II model can be used for software cost estimation.

Contents

23.1 Software pricing
23.2 Plan-driven development
23.3 Project scheduling
23.4 Agile planning
23.5 Estimation techniques
23.6 COCOMO cost modeling

Project planning is one of the most important jobs of a software project manager. As a manager, you have to break down the work into parts and assign them to project team members, anticipate problems that might arise, and prepare tentative solutions to those problems. The project plan, which is created at the start of a project and updated as the project progresses, is used to show how the work will be done and to assess progress on the project.

Project planning takes place at three stages in a project life cycle:

1. At the proposal stage, when you are bidding for a contract to develop or provide a software system. You need a plan at this stage to help you decide if you have the resources to complete the work and to work out the price that you should quote to a customer.
2. During the project startup phase, when you have to plan who will work on the project, how the project will be broken down into increments, how resources will be allocated across your company, and so on. Here, you have more information than at the proposal stage, and you can therefore refine the initial effort estimates that you have prepared.
3. Periodically throughout the project, when you update your plan to reflect new information about the software and its development. You learn more about the system being implemented and the capabilities of your development team. As software requirements change, the work breakdown has to be altered and the schedule extended. This information allows you to make more accurate estimates of how long the work will take.

Planning at the proposal stage is inevitably speculative, as you do not have a complete set of requirements for the software to be developed. You have to respond to a call for proposals based on a high-level description of the software functionality that is required. A plan is often a required part of a proposal, so you have to produce a credible plan for carrying out the work. If you win the contract, you then have to re-plan the project, taking into account changes since the proposal was made and new information about the system, the development process, and the development team.

When you are bidding for a contract, you have to work out the price that you will propose to the customer for developing the software. As a starting point for calculating this price, you need to draw up an estimate of your costs for completing the project work. Estimation involves working out how much effort is required to complete each activity and, from this step, calculating the total cost of activities. You should always calculate software costs objectively, with the aim of accurately predicting the cost of developing the software. Once you have a reasonable estimate of the likely costs, you are then in a position to calculate the price that you will quote to the customer. As I discuss in the next section, many factors influence the pricing of a software project—it is not simply cost plus profit.

Overhead costs

When you estimate the costs of effort on a software project, you don't simply multiply the salaries of the people involved by the time spent on the project. You have to take into account all of the organizational overheads (office space, administration, etc.) that must be covered by the income from a project. You calculate the costs by computing these overheads and adding a proportion to the costs of each engineer working on a project.

http://software-engineering-book.com/web/overhead-costs/

You should use three main parameters when computing the costs of a software development project:

- effort costs (the costs of paying software engineers and managers);
- hardware and software costs, including hardware maintenance and software support; and
- travel and training costs.

For most projects, the biggest cost is the effort cost. You have to estimate the total effort (in person-months) that is likely to be required to complete the work of a project. Obviously, you have limited information to make such an estimate. You therefore make the best possible estimate and then add contingency (extra time and effort) in case your initial estimate is optimistic.

For commercial systems, you normally use commodity hardware, which is relatively cheap. However, software costs can be significant if you have to license middleware and platform software. Extensive travel may be needed when a project is developed at different sites. While travel costs themselves are usually a small fraction of the effort costs, the time spent traveling is often wasted and adds significantly to the effort costs of the project. You can use electronic meeting systems and other collaborative software to reduce travel and so have more time available for productive work.

Once a contract to develop a system has been awarded, the outline project plan for the project has to be refined to create a project startup plan. At this stage, you should know more about the requirements for this system. Your aim should be to create a project plan with enough detail to help make decisions about project staffing and budgeting. You use this plan as a basis for allocating resources to the project from within the organization and to help decide if you need to hire new staff.

The plan should also define project monitoring mechanisms. You must keep track of the progress of the project and compare actual and planned progress and costs. Although most companies have formal procedures for monitoring, a good manager should be able to form a clear picture of what is going on through informal discussions with project staff. Informal monitoring can predict potential project problems by revealing difficulties as they occur. For example, daily discussions with project

staff might reveal that the team is having problems with a software fault in the communications systems. The project manager can then immediately assign a communications expert to the problem to help find and solve the problem.

The project plan always evolves during the development process because of requirements changes, technology issues, and development problems. Development planning is intended to ensure that the project plan remains a useful document for staff to understand what is to be achieved and when it is to be delivered. Therefore, the schedule, cost estimate, and risks all have to be revised as the software is developed.

If an agile method is used, there is still a need for a project startup plan because regardless of the approach used, the company still needs to plan how resources will be allocated to a project. However, this is not a detailed plan, and you only need to include essential information about the work breakdown and project schedule. During development, an informal project plan and effort estimates are drawn up for each release of the software, with the whole team involved in the planning process. Some aspects of agile planning have already been covered in Chapter 3, and I discuss other approaches in Section 23.4.

23.1 Software pricing

In principle, the price of a software system developed for a customer is simply the cost of development plus profit for the developer. In practice, however, the relationship between the project cost and the price quoted to the customer is not usually so simple. When calculating a price, you take broader organizational, economic, political, and business considerations into account (Figure 23.1). You need to think about organizational concerns, the risks associated with the project, and the type of contract that will be used. These issues may cause the price to be adjusted upward or downward.

To illustrate some of the project pricing issues, consider the following scenario:

A small software company, PharmaSoft, employs 10 software engineers. It has just finished a large project but only has contracts in place that require five development staff. However, it is bidding for a very large contract with a major pharmaceutical company that requires 30 person-years of effort over two years. The project will not start for at least 12 months but, if granted, it will transform the finances of the company.

PharmaSoft gets an opportunity to bid on a project that requires six people and has to be completed in 10 months. The costs (including overheads of this project) are estimated at $1.2 million. However, to improve its competitive position, PharmaSoft decides to bid a price to the customer of $0.8 million. This means that, although it loses money on this contract, it can retain specialist staff for the more profitable future projects that are likely to come on stream in a year's time.

Factor	Description
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged might then be reduced to reflect the value of the source code to the developer.
Cost estimate uncertainty	If an organization is unsure of its cost estimate, it may increase its price by a contingency over and above its normal profit.
Financial health	Companies with financial problems may lower their price to gain a contract. It is better to make a smaller-than-normal profit or break even than to go out of business. Cash flow is more important than profit in difficult economic times.
Market opportunity	A development organization may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organization the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Requirements volatility	If the requirements are likely to change, an organization may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.

Figure 23.1 Factors affecting software pricing

This is an example of an approach to software pricing called "pricing to win." Pricing to win means that a company has some *idea* of the price that the customer expects to pay and makes a bid for the contract based on the customer's expected price. This may seem unethical and unbusinesslike, but it does have advantages for both the customer and the system provider.

A project cost is agreed on the basis of an outline proposal. Negotiations then take place between client and customer to establish the detailed project specification. This specification is constrained by the agreed cost. The buyer and seller must agree on what is acceptable system functionality. The fixed factor in many projects is not the project requirements but the cost. The requirements may be changed so that the project costs remain within budget.

For example, say a company (OilSoft) is bidding for a contract to develop a fuel delivery system for an oil company that schedules deliveries of fuel to its service stations. There is no detailed requirements document for this system, so OilSoft estimates that a price of $900,000 is likely to be competitive and within the oil company's budget. After being granted the contract, OilSoft then negotiates the detailed requirements of the system so that basic functionality is delivered. It then estimates the additional costs for other requirements.

This approach has advantages for both the software developer and the customer. The requirements are negotiated to avoid requirements that are difficult to implement and potentially very expensive. Flexible requirements make it easier to reuse software. The oil company has awarded the contract to a known company that it can trust. Furthermore, it may be possible to spread the cost of

the project over several versions of the system. This may reduce the costs of system deployment and allow the client to budget for the project cost over several financial years.

23.2 Plan-driven development

Plan-driven or plan-based development is an approach to software engineering where the development process is planned in detail. A project plan is created that records the work to be done, who will do it, the development schedule, and the work products. Managers use the plan to support project decision making and as a way of measuring progress. Plan-driven development is based on engineering project management techniques and can be thought of as the "traditional" way of managing large software development projects. Agile development involves a different planning process, discussed in Section 23.4, where decisions are delayed.

The problem with plan-driven development is that early decisions have to be revised because of changes to the environments in which the software is developed and used. Delaying planning decisions avoids unnecessary rework. However, the arguments in favor of a plan-driven approach are that early planning allows organizational issues (availability of staff, other projects, etc.) to be taken into account. Potential problems and dependencies are discovered before the project starts, rather than once the project is underway.

In my view, the best approach to project planning involves a sensible mixture of plan-based and agile development. The balance depends on the type of project and skills of the people who are available. At one extreme, large security and safety-critical systems require extensive up-front analysis and may have to be certified before they are put into use. These systems should be mostly plan-driven. At the other extreme, small to medium-size information systems, to be used in a rapidly changing competitive environment, should be mostly agile. Where several companies are involved in a development project, a plan-driven approach is normally used to coordinate the work across each development site.

23.2.1 Project plans

In a plan-driven development project, a project plan sets out the resources available to the project, the work breakdown, and a schedule for carrying out the work. The plan should identify the approach that is taken to risk management as well as risks to the project and the software under development. The details of project plans vary depending on the type of project and organization but plans normally include the following sections:

1. *Introduction* Briefly describes the objectives of the project and sets out the constraints (e.g., budget, time) that affect the management of the project.
2. *Project organization* Describes the way in which the development team is organized, the people involved, and their roles in the team.

Plan	Description
Configuration management plan	Describes the configuration management procedures and structures to be used.
Deployment plan	Describes how the software and associated hardware (if required) will be deployed in the customer's environment. This should include a plan for migrating data from existing systems.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources, and schedule used for system validation.

Figure 23.2 Project plan supplements

3. *Risk analysis* Describes possible project risks, the likelihood of these risks arising, and the risk reduction strategies (discussed in Chapter 22) that are proposed.
4. *Hardware and software resource requirements* Specifies the hardware and support software required to carry out the development. If hardware has to be purchased, estimates of the prices and the delivery schedule may be included.
5. *Work breakdown* Sets out the breakdown of the project into activities and identifies the inputs to and the outputs from each project activity.
6. *Project schedule* Shows the dependencies between activities, the estimated time required to reach each milestone, and the allocation of people to activities. The ways in which the schedule may be presented are discussed in the next section of the chapter.
7. *Monitoring and reporting mechanisms* Defines the management reports that should be produced, when these should be produced, and the project monitoring mechanisms to be used.

The main project plan should always include a project risk assessment and a schedule for the project. In addition, you may develop a number of supplementary plans for activities such as testing and configuration management. Figure 23.2 shows some supplementary plans that may be developed. These are all usually needed in large projects developing large, complex systems.

23.2.2 The planning process

Project planning is an iterative process that starts when you create an initial project plan during the project startup phase. Figure 23.3 is a UML activity diagram that shows a typical workflow for a project planning process. Plan changes are inevitable. As more information about the system and the project team becomes available

Figure 23.3 The project planning process

during the project, you should regularly revise the plan to reflect requirements, schedule, and risk changes. Changing business goals also leads to changes in project plans. As business goals change, this could affect all projects, which may then have to be re-planned.

At the beginning of a planning process, you should assess the constraints affecting the project. These constraints are the required delivery date, staff available, overall budget, available tools, and so on. In conjunction with this assessment, you should also identify the project milestones and deliverables. Milestones are points in the schedule against which you can assess progress, for example, the handover of the system for testing. Deliverables are work products that are delivered to the customer, for example, a requirements document for the system.

The process then enters a loop that terminates when the project is complete. You draw up an estimated schedule for the project, and the activities defined in the schedule are initiated or are approved to continue. After some time (usually about two to three weeks), you should review progress and note discrepancies from the planned schedule. Because initial estimates of project parameters are inevitably approximate, minor slippages are normal and you will have to make modifications to the original plan.

You should make realistic rather than optimistic assumptions when you are defining a project plan. Problems of some description always arise during a project, and these lead to project delays. Your initial assumptions and scheduling should therefore be pessimistic and take unexpected problems into account. You should include contingency in your plan so that if things go wrong, then your delivery schedule is not seriously disrupted.

If there are serious problems with the development work that are likely to lead to significant delays, you need to initiate risk mitigation actions to reduce the risks of project failure. In conjunction with these actions, you also have to re-plan the project. This may involve renegotiating the project constraints and deliverables with the customer. A new schedule of when work should be completed also has to be established and agreed to with the customer.

If this renegotiation is unsuccessful or the risk mitigation actions are ineffective, then you should arrange for a formal project technical review. The objectives of this review are to find an alternative approach that will allow the project to continue. Reviews should also check that the customer's goals are unchanged and that the project remains aligned with these goals.

The outcome of a review may be a decision to cancel a project. This may be a result of technical or managerial failings but, more often, is a consequence of external changes that affect the project. The development time for a large software project is often several years. During that time, the business objectives and priorities inevitably change. These changes may mean that the software is no longer required or that the original project requirements are inappropriate. Management may then decide to stop software development or to make major changes to the project to reflect the changes in the organizational objectives.

23.3 Project scheduling

Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed. You estimate the calendar time needed to complete each task and the effort required, and you suggest who will work on the tasks that have been identified. You also have to estimate the hardware and software resources that are needed to complete each task. For example, if you are developing an embedded system, you have to estimate the time that you need on specialized hardware and the costs of running a system simulator. In terms of the planning stages that I introduced in the introduction of this chapter, an initial project schedule is usually created during the project startup phase. This schedule is then refined and modified during development planning.

Both plan-based and agile processes need an initial project schedule, although less detail is included in an agile project plan. This initial schedule is used to plan how people will be allocated to projects and to check the progress of the project against its contractual commitments. In traditional development processes, the complete schedule is initially developed and then modified as the project progresses. In agile processes, there has to be an overall schedule that identifies when the major phases of the project will be completed. An iterative approach to scheduling is then used to plan each phase.

Scheduling in plan-driven projects (Figure 23.4) involves breaking down the total work involved in a project into separate tasks and estimating the time required to complete each task. Tasks should normally last at least a week and no longer than 2 months. Finer subdivision means that a disproportionate amount of time must be spent on re-planning and updating the project plan. The maximum amount of time for any task should be 6 to 8 weeks. If a task will take longer than this, it should be split into subtasks for project planning and scheduling.

Some of these tasks are carried out in parallel, with different people working on different components of the system. You have to coordinate these parallel tasks and organize the work so that the workforce is used optimally and you don't introduce

Figure 23.4 The project scheduling process

unnecessary dependencies between the tasks. It is important to avoid a situation where the whole project is delayed because a critical task is unfinished.

If a project is technically advanced, initial estimates will almost certainly be optimistic even when you try to consider all eventualities. In this respect, software scheduling is no different from scheduling any other type of large advanced project. New aircraft, bridges, and even new models of cars are frequently late because of unanticipated problems. Schedules, therefore, must be continually updated as better progress information becomes available. If the project being scheduled is similar to a previous project, previous estimates may be reused. However, projects may use different design methods and implementation languages, so experience from previous projects may not be applicable in the planning of a new project.

When you are estimating schedules, you must take into account the possibility that things will go wrong. People working on a project may fall ill or leave, hardware may fail, and essential support software or hardware may be delivered late. If the project is new and technically advanced, parts of it may turn out to be more difficult and take longer than originally anticipated.

A good rule of thumb is to estimate as if nothing will go wrong and then increase your estimate to cover anticipated problems. A further contingency factor to cover unanticipated problems may also be added to the estimate. This extra contingency factor depends on the type of project, the process parameters (deadline, standards, etc.), and the quality and experience of the software engineers working on the project. Contingency estimates may add 30 to 50% to the effort and time required for the project.

23.3.1 Schedule presentation

Project schedules may simply be documented in a table or spreadsheet showing the tasks, estimated effort, duration, and task dependencies (Figure 23.5). However, this style of presentation makes it difficult to see the relationships and dependencies between the different activities. For this reason, alternative graphical visualizations of project schedules have been developed that are often easier to read and understand. Two types of visualization are commonly used:

1. Calendar-based bar charts show who is responsible for each activity, the expected elapsed time, and when the activity is scheduled to begin and end. Bar charts are also called Gantt charts, after their inventor, Henry Gantt.

Task	Effort (person-days)	Duration (days)	Dependencies
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

Figure 23.5 Tasks, durations, and dependencies

2. Activity networks show the dependencies between the different activities making up a project. These networks are described in an associated web section.

Project activities are the basic planning element. Each activity has:

- a duration in calendar days or months;
- an effort estimate, which shows the number of person-days or person-months to complete the work;
- a deadline by which the activity should be complete; and
- a defined endpoint, which might be a document, the holding of a review meeting, the successful execution of all tests, or the like.

When planning a project, you may decide to define project milestones. A milestone is a logical end to a stage of the project where the progress of the work can be reviewed. Each milestone should be documented by a brief report (often simply an email) that summarizes the work done and whether or not the work has been completed as planned. Milestones may be associated with a single task or with groups of related activities. For example, in Figure 23.5, milestone M1 is associated with task T1 and marks the end of that activity. Milestone M3 is associated with a pair of tasks T2 and T4; there is no individual milestone at the end of these tasks.

Activity charts

An activity chart is a project schedule representation that presents the project plan as a directed graph. It shows which tasks can be carried out in parallel and those that must be executed in sequence due to their dependencies on earlier activities. If a task is dependent on several other tasks, then all of these tasks must be completed before it can start. The "critical path" through the activity chart is the longest sequence of dependent tasks. This defines the project duration.

http://software-engineering-book.com/web/planning-activities/

Some activities create project deliverables—outputs that are delivered to the software customer. Usually, the deliverables that are required are specified in the project contract, and the customer's view of the project's progress depends on these deliverables. Milestones and deliverables are not the same thing. Milestones are short reports that are used for progress reporting, whereas deliverables are more substantial project outputs such as a requirements document or the initial implementation of a system.

Figure 23.5 shows a hypothetical set of tasks, their estimated effort and duration, and task dependencies. From this table, you can see that task T3 is dependent on task T1. This means that task T1 has to be completed before T3 starts. For example, T1 might be the selection of a system for reuse and T3, the configuration of the selected system. You can't start system configuration until you have chosen and installed the application system to be modified.

Notice that the estimated duration for some tasks is more than the effort required and vice versa. If the effort is less than the duration, the people allocated to that task are not working full time on it. If the effort exceeds the duration, this means that several team members are working on the task at the same time.

Figure 23.6 takes the information in Figure 23.5 and presents the project schedule as a bar chart showing a project calendar and the start and finish dates of tasks. Reading from left to right, the bar chart clearly shows when tasks start and end. The milestones (M1, M2, etc.) are also shown on the bar chart. Notice that tasks that are independent may be carried out in parallel. For example, tasks T1, T2, and T4 all start at the beginning of the project.

As well as planning the delivery schedule for the software, project managers have to allocate resources to tasks. The key resource is, of course, the software engineers who will do the work. They have to be assigned to project activities. The resource allocation can be analyzed by project management tools, and a bar chart can be generated showing when staff are working on the project (Figure 23.7). People may be working on more than one task at the same time, and sometimes they are not working on the project. They may be on holiday, working on other projects, or attending training courses. I show part-time assignments using a diagonal line crossing the bar.

Large organizations usually employ a number of specialists who work on a project when needed. In Figure 23.7, you can see that Mary is a specialist who works on

Figure 23.6 Activity bar chart

only a single task (T5) in the project. The use of specialists is unavoidable when complex systems are being developed, but it can lead to scheduling problems. If one project is delayed while a specialist is working on it, this may affect other projects where the specialist is also required. These projects may be delayed because the specialist is not available.

If a task is delayed, later tasks that are dependent on it may be affected. They cannot start until the delayed task is completed. Delays can cause serious problems with staff allocation, especially when people are working on several projects at the same time. If a task (T) is delayed, the people allocated to it may be assigned to other work (W). To complete this work may take longer than the delay, but, once assigned, they cannot simply be reassigned back to the original task. This may then lead to further delays in T as they complete W.

Normally, you should use a project planning tool, such as the Basecamp or Microsoft project, to create, update, and analyze project schedule information. Project management tools usually expect you to input project information into a table, and they create a database of project information. Bar charts and activity charts can then be generated automatically from this database.

Figure 23.7 Staff allocation chart

23.4 Agile planning

Agile methods of software development are iterative approaches where the software is developed and delivered to customers in increments. Unlike plan-driven approaches, the functionality of these increments is not planned in advance but is decided during the development. The decision on what to include in an increment depends on progress and on the customer's priorities. The argument for this approach is that the customer's priorities and requirements change, so it makes sense to have a flexible plan that can accommodate these changes. Cohn's book (Cohn 2005) is an excellent introduction to agile planning.

Agile development methods such as Scrum (Rubin 2013) and Extreme Programming (Beck and Andres 2004) have a two-stage approach to planning, corresponding to the startup phase in plan-driven development and development planning:

1. *Release planning,* which looks ahead for several months and decides on the features that should be included in a release of a system.
2. *Iteration planning,* which has a shorter term outlook and focuses on planning the next increment of a system. This usually represents 2 to 4 weeks of work for the team.

I have already explained the Scrum approach to planning in Chapter 3, which is based on project backlogs and daily reviews of work to be done. It is primarily geared

Figure 23.8 The "planning game"

to iteration planning. Another approach to agile planning, which was developed as part of Extreme Programming, is based on user stories. The so-called planning game can be used in both release planning and iteration planning.

The basis of the planning game (Figure 23.8) is a set of user stories (see Chapter 3) that cover all of the functionality to be included in the final system. The development team and the software customer work together to develop these stories. The team members read and discuss the stories and rank them based on the amount of time they think it will take to implement the story. Some stories may be too large to implement in a single iteration, and these are broken down into smaller stories.

The problem with ranking stories is that people often find it difficult to estimate how much effort or time is needed to do something. To make this easier, relative ranking may be used. The team compares stories in pairs and decides which will take the most time and effort, without assessing exactly how much effort will be required. At the end of this process, the list of stories has been ordered, with the stories at the top of the list taking the most effort to implement. The team then allocates notional effort points to all of the stories in the list. A complex story may have 8 points and a simple story 2 points.

Once the stories have been estimated, the relative effort is translated into the first estimate of the total effort required by using the idea of "velocity." Velocity is the number of effort points implemented by the team, per day. This can be estimated either from previous experience or by developing one or two stories to see how much time is required. The velocity estimate is approximate but is refined during the development process. Once you have a velocity estimate, you can calculate the total effort in person-days to implement the system.

Release planning involves selecting and refining the stories that will reflect the features to be implemented in a release of a system and the order in which the stories should be implemented. The customer has to be involved in this process. A release date is then chosen, and the stories are examined to see if the effort estimate is consistent with that date. If not, stories are added or removed from the list.

Iteration planning is the first stage in developing a deliverable system increment. Stories to be implemented during that iteration are chosen, with the number of stories reflecting the time to deliver an workable system (usually 2 or 3 weeks) and the team's velocity. When the delivery date is reached, the development iteration is complete, even if all of the stories have not been implemented. The team considers the stories that have been implemented and adds up their effort points. The velocity can then be recalculated, and this measure is used in planning the next version of the system.

At the start of each development iteration, there is a task planning stage where the developers break down stories into development tasks. A development task should take 4–16 hours. All of the tasks that must be completed to implement all of the stories in that iteration are listed. The individual developers then sign up for the specific

tasks that they will implement. Each developer knows their individual velocity and so should not sign up for more tasks than they can implement in the time allotted.

This approach to task allocation has two important benefits:

1. The whole team gets an overview of the tasks to be completed in an iteration. They therefore have an understanding of what other team members are doing and who to talk to if task dependencies are identified.
2. Individual developers choose the tasks to implement; they are not simply allocated tasks by a project manager. They therefore have a sense of ownership in these tasks, and this is likely to motivate them to complete the task.

Halfway through an iteration, progress is reviewed. At this stage, half of the story effort points should have been completed. So, if an iteration involves 24 story points and 36 tasks, 12 story points and 18 tasks should have been completed. If this is not the case, then there has to be discussions with the customer about which stories should be removed from the system increment that is being developed.

This approach to planning has the advantage that a software increment is always delivered at the end of each project iteration. If the features to be included in the increment cannot be completed in the time allowed, the scope of the work is reduced. The delivery schedule is never extended. However, this can cause problems as it means that customer plans may be affected. Reducing the scope may create extra work for customers if they have to use an incomplete system or change the way they work between one release of the system and another.

A major difficulty in agile planning is that it relies on customer involvement and availability. This involvement can be difficult to arrange, as customer representatives sometimes have to prioritize other work and are not available for the planning game. Furthermore, some customers may be more familiar with traditional project plans and may find it difficult to engage in an agile planning process.

Agile planning works well with small, stable development teams that can get together and discuss the stories to be implemented. However, where teams are large and/or geographically distributed, or when team membership changes frequently, it is practically impossible for everyone to be involved in the collaborative planning that is essential for agile project management. Consequently, large projects are usually planned using traditional approaches to project management.

23.5 Estimation techniques

Estimating project schedules is difficult. You have to make initial estimates on the basis of an incomplete user requirements definition. The software may have to run on unfamiliar platforms or use new development technology. The people involved in the project and their skills will probably not be known. There are so many uncertainties that it is impossible to estimate system development costs accurately during the early

Figure 23.9 Estimate uncertainty

stages of a project. Nevertheless, organizations need to make software effort and cost estimates. Two types of techniques can be used for making estimates:

1. *Experience-based techniques* The estimate of future effort requirements is based on the manager's experience of past projects and the application domain. Essentially, the manager makes an informed judgment of what the effort requirements are likely to be.
2. *Algorithmic cost modeling* In this approach, a formulaic approach is used to compute the project effort based on estimates of product attributes, such as size, process characteristics, and experience of staff involved.

In both cases, you need to use your judgment to estimate either the effort directly or the project and product characteristics. In the startup phase of a project, these estimates have a wide margin of error. Based on data collected from a large number of projects, Boehm et al. (B. Boehm et al. 1995) discovered that startup estimates vary significantly. If the initial estimate of effort required is x months of effort, they found that the range may be from $0.25x$ to $4x$ of the actual effort as measured when the system was delivered. During development planning, estimates become more and more accurate as the project progresses (Figure 23.9).

Experience-based techniques rely on the manager's experience of past projects and the actual effort expended in these projects on activities that are related to software development. Typically, you identify the deliverables to be produced in a project and the different software components or systems that are to be developed. You document these in a spreadsheet, estimate them individually, and compute the total effort required. It usually helps to get a group of people involved in the effort estimation and to ask each member of the group to explain their estimate. This often reveals factors that others have not considered, and you then iterate toward an agreed group estimate.

The difficulty with experience-based techniques is that a new software project may not have much in common with previous projects. Software development changes very quickly, and a project will often use unfamiliar techniques such as web services, application system configuration, or HTML5. If you have not worked with these techniques, your previous experience may not help you to estimate the effort required, making it more difficult to produce accurate costs and schedule estimates.

It is impossible to say whether experience-based or algorithmic approaches are more accurate. Project estimates are often self-fulfilling. The estimate is used to define the project budget, and the product is adjusted so that the budget figure is realized. A project that is within budget may have achieved this at the expense of features in the software being developed.

To make a comparison of the accuracy of these techniques, a number of controlled experiments would be required where several techniques were used independently to estimate the project effort and costs. No changes to the project would be allowed, and the final effort could them be compared. The project manager would not know the effort estimates, so no bias would be introduced. However, this scenario is completely impossible in real projects, so we will never have an objective comparison of these approaches.

23.5.1 Algorithmic cost modeling

Algorithmic cost modeling uses a mathematical formula to predict project costs based on estimates of the project size, the type of software being developed, and other team, process, and product factors. Algorithmic cost models are developed by analyzing the costs and attributes of completed projects, then finding the closest-fit formula to the actual costs incurred.

Algorithmic cost models are primarily used to make estimates of software development costs. However, Boehm and his collaborators (B. W. Boehm et al. 2000) discuss a range of other uses for these models, such as the preparation of estimates for investors in software companies, alternative strategies to help assess risks and to inform decisions about reuse, redevelopment, or outsourcing.

Most algorithmic models for estimating effort in a software project are based on a simple formula:

$$\text{Effort} = A \times \text{Size}^B \times M$$

A: a constant factor, which depends on local organizational practices and the type of software that is developed.

Size: an assessment of the code size of the software or a functionality estimate expressed in function or application points.

B: represents the complexity of the software and usually lies between 1 and 1.5.

M: is a factor that takes into account process, product and development attributes, such as the dependability requirements for the software and the experience of the development team. These attributes may increase or decrease the overall difficulty of developing the system.

The number of lines of source code (SLOC) in the delivered system is the fundamental size metric that is used in many algorithmic cost models. To estimate the number of lines of code in a system, you may use a combination of approaches:

1. Compare the system to be developed with similar systems and use their code size as the basis for your estimate.
2. Estimate the number of function or application points in the system (see the following section) and formulaically convert these to lines of code in the programming language used.
3. Rank the system components using judgment of their relative sizes and use a known reference component to translate this ranking to code sizes.

Most algorithmic estimation models have an exponential component (B in the above equation) that increases with the size and complexity of the system. This reflects the fact that costs do not usually increase linearly with project size. As the size and complexity of the software increase, extra costs are incurred because of the communication overhead of larger teams, more complex configuration management, more difficult system integration, and so on. The more complex the system, the more these factors affect the cost.

The idea of using a scientific and objective approach to cost estimation is an attractive one, but all algorithmic cost models suffer from two key problems:

1. It is practically impossible to estimate Size accurately at an early stage in a project, when only the specification is available. Function-point and application-point estimates (see later) are easier to produce than estimates of code size but are also usually inaccurate.
2. The estimates of the complexity and process factors contributing to B and M are subjective. Estimates vary from one person to another, depending on their background and experience of the type of system that is being developed.

Accurate code size estimation is difficult at an early stage in a project because the size of the final program depends on design decisions that may not have been made when the estimate is required. For example, an application that requires high-performance data management may either implement its own data management system or use a commercial database system. In the initial cost estimation, you are unlikely to know if there is a commercial database system that performs well enough to meet the performance requirements. You therefore don't know how much data management code will be included in the system.

The programming language used for system development also affects the number of lines of code to be developed. A language like Java might mean that more lines of code are necessary than if C (say) was used. However, this extra code allows more compile-time checking, so validation costs are likely to be reduced. It is not clear how this should be taken into account in the estimation process. Code reuse also

Software productivity

Software productivity is an estimate of the average amount of development work that software engineers complete in a week or a month. It is therefore expressed as lines of code/month, function points/month, and so forth.

However, while productivity can be easily measured where there is a tangible outcome (e.g., an administrator processes *N* travel claims/day), software productivity is more difficult to define. Different people may implement the same functionality in different ways, using different numbers of lines of code. The quality of the code is also important but is, to some extent, subjective. Therefore, you can't really compare the productivity of individual engineers. It only makes sense to use productivity measures with large groups.

http://software-engineering-book.com/web/productivity/

makes a difference, and some models explicitly estimate the number of lines of code reused. However, if application systems or external services are reused, it is very difficult to compute the number of lines of source code that these replace.

Algorithmic cost models are a systematic way to estimate the effort required to develop a system. However, these models are complex and difficult to use. There are many attributes and considerable scope for uncertainty in estimating their values. This complexity means that the practical application of algorithmic cost modeling has been limited to a relatively small number of large companies, mostly working in defense and aerospace systems engineering.

Another barrier that discourages the use of algorithmic models is the need for calibration. Model users should calibrate their model and the attribute values using their own historical project data, as this reflects local practice and experience. However, very few organizations have collected enough data from past projects in a form that supports model calibration. Practical use of algorithmic models, therefore, has to start with the published values for the model parameters. It is practically impossible for a modeler to know how closely these relate to his or her organization.

If you use an algorithmic cost estimation model, you should develop a range of estimates (worst, expected, and best) rather than a single estimate and apply the costing formula to all of them. Estimates are most likely to be accurate when you understand the type of software that is being developed and have calibrated the costing model using local data, or when programming language and hardware choices are predefined.

23.6 COCOMO cost modeling

The best known algorithmic cost modeling technique and tool is the COCOMO II model. This empirical model was derived by collecting data from a large number of software projects of different sizes. These data were analyzed to discover the formulas that were the best fit to the observations. These formulas linked the size of the

Figure 23.10 COCOMO estimation models

system and product, project, and team factors to the effort to develop the system. COCOMO II is a freely available model that is supported with open-source tools.

COCOMO II was developed from earlier COCOMO (Constructive Cost Modeling) cost estimation models, which were largely based on original code development (B. W. Boehm 1981; B. Boehm and Royce 1989). The COCOMO II model takes into account modern approaches to software development, such as rapid development using dynamic languages, development with reuse, and database programming. COCOMO II embeds several submodels based on these techniques, which produce increasingly detailed estimates.

The submodels (Figure 23.10) that are part of the COCOMO II model are:

1. *An application composition model* This models the effort required to develop systems that are created from reusable components, scripting, or database programming. Software size estimates are based on application points, and a simple size/productivity formula is used to estimate the effort required.

2. *An early design model* This model is used during early stages of the system design after the requirements have been established. The estimate is based on the standard estimation formula that I discussed in the introduction of this chapter, with a simplified set of seven multipliers. Estimates are based on function points, which are then converted to number of lines of source code.

   Function points are a language-independent way of quantifying program functionality. You compute the total number of function points in a program by measuring or estimating the number of external inputs and outputs, user interactions, external interfaces, and files or database tables used by the system.

3. *A reuse model* This model is used to compute the effort required to integrate reusable components and/or automatically generated program code. It is normally used in conjunction with the post-architecture model.

4. *A post-architecture model* Once the system architecture has been designed, a more accurate estimate of the software size can be made. Again, this model uses the standard formula for cost estimation discussed above. However, it includes a more extensive set of 17 multipliers reflecting personnel capability, product, and project characteristics.

Of course, in large systems, different parts of the system may be developed using different technologies, and you may not have to estimate all parts of the system to the same level of accuracy. In such cases, you can use the appropriate submodel for each part of the system and combine the results to create a composite estimate.

The COCOMO II model is a very complex model and, to make it easier to explain, I have simplified its presentation. You could use the models as I have explained them here for simple cost estimation. However, to use COCOMO properly, you should refer to Boehm's book and the manual for the COCOMO II model (B. W. Boehm et al. 2000; Abts et al. 2000).

23.6.1 The application composition model

The application composition model was introduced into COCOMO II to support the estimation of effort required for prototyping projects and for projects where the software is developed by composing existing components. It is based on an estimate of weighted application points (sometimes called object points), divided by a standard estimate of application point productivity (B. W. Boehm et al. 2000). The number of application points in a program is derived from four simpler estimates:

- the number of separate screens or web pages that are displayed;
- the number of reports that are produced;
- the number of modules in imperative programming languages (such as Java); and
- the number of lines of scripting language or database programming code.

This estimate is then adjusted according to the difficulty of developing each application point. Productivity depends on the developer's experience and capability as well as the capabilities of the software tools (ICASE) used to support development. Figure 23.11 shows the levels of application-point productivity suggested by the COCOMO model developers.

Application composition usually relies on reusing existing software and configuring application systems. Some of the application points in the system will therefore be implemented using reusable components. Consequently, you have to adjust the

Developer's experience and capability	Very low	Low	Nominal	High	Very high
ICASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD (NAP/month)	4	7	13	25	50

Figure 23.11 Application-point productivity

estimate to take into account the percentage of reuse expected. Therefore, the final formula for effort computation for system prototypes is:

$$PM = (NAP \times (1 - \%reuse/100)) / PROD$$

PM: the effort estimate in person-months.

NAP: the total number of application points in the delivered system.

%reuse: an estimate of the amount of reused code in the development.

PROD: the application-point productivity as shown in Figure 23.11.

23.6.2 The early design model

This model may be used during the early stages of a project, before a detailed architectural design for the system is available. The early design model assumes that user requirements have been agreed and initial stages of the system design process are underway. Your goal at this stage should be to make a quick and approximate cost estimate. Therefore, you have to make simplifying assumptions, such as the assumption that there is no effort involved in integrating reusable code.

Early design estimates are most useful for option exploration where you need to compare different ways of implementing the user requirements. The estimates produced at this stage are based on the standard formula for algorithmic models, namely:

$$Effort = A \times Size^{B} \times M$$

Based on his own large dataset, Boehm proposed that the co-efficient **A** should be 2.94. The size of the system is expressed in KSLOC, which is the number of thousands of lines of source code. You calculate KSLOC by estimating the number of function points in the software. You then use standard tables, which relate software size to function points for different programming languages (QSM 2014) to compute an initial estimate of the system size in KSLOC.

The exponent **B** reflects the increased effort required as the size of the project increases. This can vary from 1.1 to 1.24 depending on the novelty of the project, the development flexibility, the risk resolution processes used, the cohesion of the development team, and the process maturity level (see web Chapter 26) of the organization. I discuss how the value of this exponent is calculated using these parameters in the description of the COCOMO II post-architecture model.

This results in an effort computation as follows:

$$PM = 2.94 \times Size^{(1.1 \text{ to } 1.24)} \times M$$

$$M = PERS \times PREX \times RCPX \times RUSE \times PDIF \times SCED \times FSIL$$

PERS: personnel capability

PREX: personnel experience

RCPX: product reliability and complexity

RUSE: reuse required

PDIF: platform difficulty

SCED: schedule

FSIL: support facilities

The multiplier **M** is based on seven project and process attributes that increase or decrease the estimate. I explain these attributes on the book's web pages. You estimate values for these attributes using a six-point scale, where 1 corresponds to "very low" and 6 corresponds to "very high"; for example, **PERS = 6** means that expert staff are available to work on the project.

23.6.3 The reuse model

The COCOMO reuse model is used to estimate the effort required to integrate reusable or generated code. As I have discussed in Chapter 15, software reuse is now the norm in all software development. Most large systems include a significant amount of code that has been reused from previous development projects.

COCOMO II considers two types of reused code. Black-box code is code that can be reused without understanding the code or making changes to it. Examples of black-box code are components that are automatically generated from UML models or application libraries such as graphics libraries. It is assumed that the development effort for black-box code is zero. Its size is not taken into account in the overall effort computation.

White-box code is reusable code that has to be adapted to integrate it with new code or other reused components. Development effort is required for reuse because the code has to be understood and modified before it can work correctly in the system. White-box code could be automatically generated code that needs manual changes or additions. Alternatively, it can be reused components from other systems that have to be modified in the system that is being developed.

Three factors contribute to the effort involved in reusing white-box code components:

1. The effort involved in assessing whether or not a component could be reused in a system that is being developed.
2. The effort required to understand the code that is being reused.
3. The effort required to modify the reused code to adapt it and integrate it with the system being developed.

The development effort in the reuse model is calculated using the COCOMO early design model and is based on the total number of lines of code in the system. The code size includes new code developed for components that are not reused plus an additional factor that allows for the effort involved in reusing and integrating existing code. This additional factor is called ESLOC, the equivalent number of lines of new source code. That is, you express the reuse effort as the effort that would be involved in developing some additional source code.

The formula used to calculate the source code equivalence is:

$$ESLOC = (ASLOC \times (1\text{-}AT/100) \times AAM)$$

ESLOC: the equivalent number of lines of new source code.

ASLOC: an estimate of the number of lines of code in the reused components that have to be changed.

AT: the percentage of reused code that can be modified automatically.

AAM: an Adaptation Adjustment Multiplier that reflects the additional effort required to reuse components.

In some cases, the adjustments required to reuse code are syntactic and can be implemented by an automated tool. These do not involve significant effort, so you should estimate what fraction of the changes made to reused code can be automated (AT). This reduces the total number of lines of code that have to be adapted.

The Adaptation Adjustment Multiplier (AAM) adjusts the estimate to reflect the additional effort required to reuse code. The COCOMO model documentation (Abts et al. 2000) discusses in detail how AAM should be calculated. Simplistically, AAM is the sum of three components:

1. An assessment factor (referred to as AA) that represents the effort involved in deciding whether or not to reuse components. AA varies from 0 to 8 depending on the amount of time you need to spend looking for and assessing potential candidates for reuse.
2. An understanding component (referred to as SU) that represents the costs of understanding the code to be reused and the familiarity of the engineer with the code that is being reused. SU ranges from 50 for complex, unstructured code to 10 for well-written, object-oriented code.
3. An adaptation component (referred to as AAF) that represents the costs of making changes to the reused code. These include design, code, and integration changes.

Once you have calculated a value for ESLOC, you apply the standard estimation formula to calculate the total effort required, where the Size parameter = ESLOC. Therefore, the formula to estimate the reuse effort is:

$$Effort = A \times ESLOC^{B} \times M$$

where A, B, and M have the same values as used in the early design model.

COCOMO cost drivers

COCOMO II cost drivers are attributes that reflect some of the product, team, process, and organizational factors that affect the amount of effort needed to develop a software system. For example, if a high level of reliability is required, extra effort will be needed; if there is a need for rapid delivery, extra effort will be required; if the team members change, extra effort will be required.

There are 17 of these attributes in the COCOMO II model, which have been assigned estimated values by the model developers.

http://software-engineering-book.com/web/cost-drivers/

23.6.4 The post-architecture level

The post-architecture model is the most detailed of the COCOMO II models. It is used when you have an initial architectural design for the system. The starting point for estimates produced at the post-architecture level is the same basic formula used in the early design estimates:

$$PM = A \times Size^{B} \times M$$

By this stage in the process, you should be able to make a more accurate estimate of the project size, as you know how the system will be decomposed into subsystems and components. You make this estimate of the overall code size by adding three code size estimates:

1. An estimate of the total number of lines of new code to be developed (SLOC).
2. An estimate of the reuse costs based on an equivalent number of source lines of code (ESLOC), calculated using the reuse model.
3. An estimate of the number of lines of code that may be changed because of changes to the system requirements.

The final component in the estimate—the number of lines of modified code—reflects the fact that software requirements always change. This leads to rework and development of extra code, which you have to take into account. Of course there will often be even more uncertainty in this figure than in the estimates of new code to be developed.

The exponent term (B) in the effort computation formula is related to the levels of project complexity. As projects become more complex, the effects of increasing system size become more significant. The value of the exponent B is based on five factors, as shown in Figure 23.12. These factors are rated on a six-point scale from 0 to 5, where 0 means “extra high” and 5 means “very low.” To calculate B, you add the ratings, divide them by 100, and add the result to 1.01 to get the exponent that should be used.

Scale factor	Explanation
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis; extra-high means a complete and thorough risk analysis.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; extra-high means that the client sets only general goals.
Precedentedness	Reflects the previous experience of the organization with this type of project. Very low means no previous experience; extra-high means that the organization is completely familiar with this application domain.
Team cohesion	Reflects how well the development team knows each other and works together. Very low means very difficult interactions; extra-high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organization as discussed in web chapter 26. The computation of this value depends on the CMM Maturity Questionnaire, but an estimate can be achieved by subtracting the CMM process maturity level from 5.

Figure 23.12 Scale factors used in the exponent computation in the post-architecture model

For example, imagine that an organization is taking on a project in a domain in which it has little previous experience. The project client has not defined the process to be used or allowed time in the project schedule for significant risk analysis. A new development team must be put together to implement this system. The organization has recently put in place a process improvement program and has been rated as a Level 2 organization according to the SEI capability assessment, as discussed in Chapter 26 (web chapter). These characteristics lead to estimates of the ratings used in exponent calculation as follows:

1. *Precedentedness,* rated low (4). This is a new project for the organization.
2. *Development flexibility,* rated very high (1). There is no client involvement in the development process, so there are few externally imposed changes.
3. *Architecture/risk resolution,* rated very low (5). There has been no risk analysis carried out.
4. *Team cohesion,* rated nominal (3). This is a new team, so there is no information available on cohesion.
5. *Process maturity,* rated nominal (3). Some process control is in place.

The sum of these values is 16. You then calculate the final value of the exponent by dividing this sum by 100 and adding 0.01 to the result. The adjusted value of B is therefore 1.17.

The overall effort estimate is refined using an extensive set of 17 product, process, and organizational attributes (see breakout box) rather than the seven attributes used in the early design model. You can estimate values for these attributes because you have more information about the software itself, its non-functional requirements, the development team, and the development process.

Exponent value	1.17
System size (including factors for reuse and requirements volatility)	128 KLOC
Initial COCOMO estimate without cost drivers	**730 person-months**
Reliability	Very high, multiplier = 1.39
Complexity	Very high, multiplier = 1.3
Memory constraint	High, multiplier = 1.21
Tool use	Low, multiplier = 1.12
Schedule	Accelerated, multiplier = 1.29
Adjusted COCOMO estimate	**2306 person-months**
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
Adjusted COCOMO estimate	**295 person-months**

Figure 23.13 The effect of cost drivers on effort estimates

Figure 23.13 shows how the cost driver attributes influence effort estimates. Assume that the exponent value is 1.17 as discussed in the above example. Reliability (RELY), complexity (CPLX), storage (STOR), tools (TOOL), and schedule (SCED) are the key cost drivers in the project. All of the other cost drivers have a nominal value of 1, so they do not affect the effort computation.

In Figure 23.13, I have assigned maximum and minimum values to the key cost drivers to show how they influence the effort estimate. The values used are those from the COCOMO II reference manual (Abts et al. 2000). You can see that high values for the cost drivers lead an effort estimate that is more than three times the initial estimate, whereas low values reduce the estimate to about one third of the original. This highlights the significant differences between different types of project and the difficulties of transferring experience from one application domain to another.

23.6.5 Project duration and staffing

As well as estimating the overall costs of a project and the effort that is required to develop a software system, project managers must also estimate how long the software will take to develop and when staff will be needed to work on the project. Increasingly, organizations are demanding shorter development schedules so that their products can be brought to market before their competitor's.

The COCOMO model includes a formula to estimate the calendar time required to complete a project:

$$\text{TDEV} = 3 \times (\text{PM})^{(0.33 + 0.2*(B - 1.01))}$$

TDEV: the nominal schedule for the project, in calendar months, ignoring any multiplier that is related to the project schedule.

PM: the effort computed by the COCOMO model.

B: a complexity-related exponent, as discussed in section 23.5.2.

If B = 1.17 and PM = 60 then

$$\text{TDEV} = 3 \times (60)^{0.36} = 13 \text{ months}$$

The nominal project schedule predicted by the COCOMO model does not necessarily correspond with the schedule required by the software customer. You may have to deliver the software earlier or (more rarely) later than the date suggested by the nominal schedule. If the schedule is to be compressed (i.e., software is to be developed more quickly), this increases the effort required for the project. This is taken into account by the SCED multiplier in the effort estimation computation.

Assume that a project estimated TDEV as 13 months, as suggested above, but the actual schedule required was 10 months. This represents a schedule compression of approximately 25%. Using the values for the SCED multiplier as derived by Boehm's team, we see that the effort multiplier for this level of schedule compression is 1.43. Therefore, the actual effort that will be required if this accelerated schedule is to be met is almost 50% more than the effort required to deliver the software according to the nominal schedule.

There is a complex relationship between the number of people working on a project, the effort that will be devoted to the project. and the project delivery schedule. If four people can complete a project in 13 months (i.e., 52 person-months of effort), then you might think that by adding one more person, you could complete the work in 11 months (55 person-months of effort). However, the COCOMO model suggests that you will, in fact, need six people to finish the work in 11 months (66 person-months of effort).

The reason for this is that adding people to a project reduces the productivity of existing team members. As the project team increases in size, team members spend more time communicating and defining interfaces between the parts of the system developed by other people. Doubling the number of staff (for example) therefore does not mean that the duration of the project will be halved.

Consequently, when you add an extra person, the actual increment of effort added is less than one person as others become less productive. If the development team is large, adding more people to a project sometimes increases rather than reduces the development schedule because of the overall effect on productivity.

You cannot simply estimate the number of people required for a project team by dividing the total effort by the required project schedule. Usually, a small number of people are needed at the start of a project to carry out the initial design. The team then

builds up to a peak during the development and testing of the system, and then declines in size as the system is prepared for deployment. A very rapid build-up of project staff has been shown to correlate with project schedule slippage. As a project manager, you should therefore avoid adding too many staff to a project early in its lifetime.

KEY POINTS

- The price charged for a system does not just depend on its estimated development costs and the profit required by the development company. Organizational factors may mean that the price is increased to compensate for increased risk or decreased to gain competitive advantage.
- Software is often priced to gain a contract, and the functionality of the system is then adjusted to meet the estimated price.
- Plan-driven development is organized around a complete project plan that defines the project activities, the planned effort, the activity schedule, and who is responsible for each activity.
- Project scheduling involves the creation of various graphical representations of part of the project plan. Bar charts, which show the activity duration and staffing timelines, are the most commonly used schedule representations.
- A project milestone is a predictable outcome of an activity or set of activities. At each milestone, a formal report of progress should be presented to management. A deliverable is a work product that is delivered to the project customer.
- The agile planning game involves the whole team in project planning. The plan is developed incrementally, and, if problems arise, it is adjusted so that software functionality is reduced instead of delaying the delivery of an increment.
- Estimation techniques for software may be experience-based, where managers judge the effort required, or algorithmic, where the effort required is computed from other estimated project parameters.
- The COCOMO II costing model is a mature algorithmic cost model that takes project, product, hardware, and personnel attributes into account when formulating a cost estimate.

FURTHER READING

Further reading suggested in Chapter 22 is also relevant to this chapter.

"Ten Unmyths of Project Estimation." A pragmatic article that discusses the practical difficulties of project estimation and challenges some fundamental assumptions in this area. (P. Armour, *Comm. ACM*, 45(11), November 2002). http://dx.doi.org/10.1145/581571.581582

Agile Estimating and Planning. This book is a comprehensive description of story-based planning as used in XP, as well as a rationale for using an agile approach to project planning. The book also includes a good, general introduction to project planning issues. (M. Cohn, 2005, Prentice-Hall).

"Achievements and Challenges in COCOMO-based Software Resource Estimation." This article presents a history of the COCOMO models and influences on these models, and discusses the variants of these models that have been developed. It also identifies further possible developments in the COCOMO approach. (B. W. Boehm and R. Valeridi, *IEEE Software*, 25 (5), September/October 2008). http://dx.doi.org/10.1109/MS.2008.133

All About Agile; Agile Planning. This website on agile methods includes an excellent set of articles on agile planning from a number of different authors. (2007–2012). http://www.allaboutagile.com/category/agile-planning/

Project Management Knowhow: Project Planning. This website has a number of useful articles on project management in general. These are aimed at people who don't have previous experience in this area. (P. Stoemmer, 2009–2014). http://www.project-management-knowhow.com/project_planning.html

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-management/

EXERCISES

23.1. Describe the factors that affect software pricing. Define the "pricing to win" approach in software pricing.

23.2. Explain why the process of project planning is iterative and why a plan must be continually reviewed during a software project.

23.3. Define project scheduling. What are the things to be considered while estimating schedules?

23.4. What is algorithmic cost modeling? What problems does it suffer from when compared with other approaches to cost estimation?

23.5. Figure 23.14 sets out a number of tasks, their durations, and their dependencies. Draw a bar chart showing the project schedule.

Task	Duration	Dependencies
T1	10	
T2	15	T1
T3	10	T1, T2
T4	20	
T5	10	
T6	15	T3, T4
T7	20	T3
T8	35	T7
T9	15	T6
T10	5	T5, T9
T11	10	T9
T12	20	T10
T13	35	T3, T4
T14	10	T8, T9
T15	20	T12, T14
T16	10	T15

Figure 23.14
Scheduling example

23.6. Figure 23.14 shows the task durations for software project activities. Assume that a serious, unanticipated setback occurs, and instead of taking 10 days, task T5 takes 40 days. Draw up new bar charts showing how the project might be reorganized.

23.7. The planning game is based on the notion of planning to implement the stories that represent the system requirements. Explain the potential problems with this approach when software has high performance or dependability requirements.

23.8. A software manager is in charge of the development of a safety-critical software system, which is designed to control a radiotherapy machine to treat patients suffering from cancer. This system is embedded in the machine and must run on a special-purpose processor with a fixed amount of memory (256 Mbytes). The machine communicates with a patient database system to obtain the details of the patient and, after treatment, automatically records the radiation dose delivered and other treatment details in the database.

The COCOMO method is used to estimate the effort required to develop this system, and an estimate of 26 person-months is computed. All cost driver multipliers were set to 1 when making this estimate.

Explain why this estimate should be adjusted to take project, personnel, product, and organizational factors into account. Suggest four factors that might have significant effects on the initial COCOMO estimate and propose possible values for these factors. Justify why you have included each factor.

23.9. Some very large software projects involve writing millions of lines of code. Explain why the effort estimation models, such as COCOMO, might not work well when applied to very large systems.

23.10. Is it ethical for a company to quote a low price for a software contract knowing that the requirements are ambiguous and that they can charge a high price for subsequent changes requested by the customer?

REFERENCES

Abts, C., B. Clark, S. Devnani-Chulani, and B. W. Boehm. 2000. "COCOMO II Model Definition Manual." Center for Software Engineering, University of Southern California. http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_modelman2000.0.pdf

Beck, K., and C. Andres. 2004. *Extreme Programming Explained: 2nd ed*. Boston: Addison-Wesley.

Boehm, B., B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. 1995. "Cost Models for Future Software Life Cycle Processes: COCOMO 2." *Annals of Software Engineering*: 1–31. doi:10.1007/BF02249046.

Boehm, B., and W. Royce. 1989. "Ada COCOMO and the Ada Process Model." In *Proc. 5th COCOMO Users' Group Meeting*. Pittsburgh: Software Engineering Institute. http://www.dtic.mil/dtic/tr/fulltext/u2/a243476.pdf

Boehm, B. W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall.

Boehm, B. W., C. Abts, A. W. Brown, S. Chulani, B K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece. 2000. *Software Cost Estimation with COCOMO II*. Englewood Cliffs, NJ: Prentice-Hall.

Cohn, M. 2005. *Agile Estimating and Planning*. Englewood-Cliffs, NJ: Prentice Hall.

QSM. 2014. "Function Point Languages Table." http://www.qsm.com/resources/function-point-languages-table

Rubin, K. S. 2013. *Essential Scrum*. Boston: Addison-Wesley.

24

Quality management

Objectives

The objectives of this chapter are to introduce software quality management and software measurement. When you have read the chapter, you will:

- have been introduced to the quality management process and know why quality planning is important;
- be aware of the importance of standards in the quality management process and know how standards are used in quality assurance;
- understand how reviews and inspections are used as a mechanism for software quality assurance;
- understand how quality management in agile methods is based on the development of a team quality culture;
- understand how measurement may be helpful in assessing some software quality attributes, the notion of software analytics, and the limitations of software measurement.

Contents

24.1 Software quality
24.2 Software standards
24.3 Reviews and inspections
24.4 Quality management and agile development
24.5 Software measurement

Software quality management is concerned with ensuring that developed software systems are "fit for purpose." That is, systems should meet the needs of their users, should perform efficiently and reliably, and should be delivered on time and within budget. The use of quality management techniques along with new software technologies and testing methods has led to significant improvements in the level of software quality over the past 20 years.

Formalized quality management (QM) is particularly important in teams that are developing large, long-lifetime systems that take several years to develop. These systems are developed for external clients, usually using a plan-based process. For these systems, quality management is both an organizational and an individual project issue:

1. At an organizational level, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high-quality software. The QM team should take responsibility for defining the software development processes to be used and standards that should apply to the software and related documentation, including the system requirements, design, and code.
2. At a project level, quality management involves the application of specific quality processes, checking that these planned processes have been followed, and ensuring that the project outputs meet the defined project standards. Project quality management may also involve defining a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

Software quality management techniques have their roots in methods and techniques that were developed in manufacturing industries, where the terms *quality assurance* and *quality control* are widely used. Quality assurance is the definition of processes and standards that should lead to high-quality products and the introduction of quality processes into the manufacturing process. Quality control is the application of these quality processes to weed out products that are not of the required level of quality. Both quality assurance and quality control are part of quality management.

In the software industry, some companies see quality assurance as the definition of procedures, processes, and standards to ensure that software quality is achieved. In other companies, quality assurance also includes all configuration management, verification, and validation activities that are applied after a product has been handed over by a development team.

Quality management provides an independent check on the software development process. The QM team checks the project deliverables to ensure that they are consistent with organizational standards and goals (Figure 24.1). They also check process documentation, which records the tasks that have been completed by each team working on this project. The QM team uses documentation to check that important tasks have not been forgotten or that one group has not made incorrect assumptions about what other groups have done.

The QM team in large companies is usually responsible for managing the release testing process. As I discussed in Chapter 8, this means that they manage the testing of the software before it is released to customers. In addition, they are responsible

Figure 24.1 Quality management and software development

for checking that the system tests provide coverage of the requirements and that proper records of the testing process are maintained.

The QM team should be independent and not part of the software development group so that they can take an objective view of the quality of the software. They can report on software quality without being influenced by software development issues. Ideally, the QM team should have organization-wide responsibility for quality management. They should report to management above the project manager level.

Because project managers have to maintain the project budget and schedule, they may be tempted to compromise on product quality to meet that schedule. An independent QM team ensures that the organizational goals of quality are not influenced by short-term budget and schedule considerations. In smaller companies, however, this is practically impossible. Quality management and software development are inevitably intertwined with people having both development and quality responsibilities.

Formalized quality planning is an integral part of plan-based development processes. It is the process of developing a quality plan for a project. The quality plan should set out the desired software qualities and describe how these qualities are to be assessed. It defines what "high-quality" software actually means for a particular system. Engineers, therefore, have a shared understanding of the most important software quality attributes.

Humphrey (Humphrey 1989), in his classic book on software management, suggests an outline structure for a quality plan. This outline includes the following:

1. *Product introduction* A description of the product, its intended market, and the quality expectations for the product.
2. *Product plans* The critical release dates and responsibilities for the product, along with plans for distribution and product servicing.
3. *Process descriptions* The development and service processes and standards that should be used for product development and management.
4. *Quality goals* The quality goals and plans for the product, including an identification and justification of critical product quality attributes.
5. *Risks and risk management* The key risks that might affect product quality and the actions to be taken to address these risks.

Quality plans, which are developed as part of the general project planning process, differ in detail depending on the size and type of system being developed. However, when writing quality plans, you should try to keep them as short as possible. If the document is too long, people will not read it, so defeating the purpose of producing the quality plan.

Traditional quality management is a formal process that relies on maintaining extensive documentation about testing and system validation and on how processes have been followed. In this respect, it is diametrically opposed to agile development, where the aim is to spend as little time as possible in writing documents and formalizing how the development work should be done. QM techniques have therefore had to evolve when agile methods are used. I discuss QM and agile development in Section 24.4.

24.1 Software quality

The manufacturing industry established the fundamentals of quality management in a drive to improve the quality of the products that were being made. As part of this effort, the industry developed a definition of quality that was based on conformance with a detailed product specification. The underlying assumption was that products could be completely specified and procedures could be established that could check a manufactured product against its specification. Of course, products will never exactly meet a specification, so some tolerance was allowed. If the product was "almost right," it was classed as acceptable.

Software quality is not directly comparable with quality in manufacturing. The idea of tolerances is applicable in analog systems but does not apply to software. Furthermore, it is often impossible to come to an objective conclusion about whether or not a software system meets its specification:

1. It is difficult to write complete and unambiguous software requirements. Software developers and customers may interpret the requirements in different ways, and it may be impossible to reach agreement on whether or not software conforms to its specification.

2. Specifications usually integrate requirements from several classes of stakeholder. These requirements are inevitably a compromise and may not include the requirements of all stakeholder groups. The excluded stakeholders may therefore perceive the system as a poor-quality system, even though it implements the agreed requirements.

3. It is impossible to measure certain quality characteristics (e.g., maintainability) directly, and so they cannot be specified in an unambiguous way. I discuss the difficulties of measurement in Section 24.4.

Because of these problems, the assessment of software quality is a subjective process. The quality management team uses their judgment to decide if an acceptable level of quality has been achieved. They decide whether or not the software is fit for

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

Figure 24.2 Software quality attributes

its intended purpose. This decision involves answering questions about the system's characteristics. For example:

1. Has the software been properly tested, and has it been shown that all requirements have been implemented?
2. Is the software sufficiently dependable to be put into use?
3. Is the performance of the software acceptable for normal use?
4. Is the software usable?
5. Is the software well structured and understandable?
6. Have programming and documentation standards been followed in the development process?

There is a general assumption in software quality management that the system will be tested against its requirements. The judgment on whether or not it delivers the required functionality should be based on the results of these tests. Therefore, the QM team should review the tests that have been developed and examine the test records to check that testing has been properly carried out. In some companies, the QM group carries out final system testing; in others, a dedicated system testing team reports to the system quality manager.

The subjective quality of a software system is largely based on its non-functional characteristics. This reflects practical user experience—if the software's functionality is not what is expected, then users will often just work around this deficiency and find other ways to do what they want to do. However, if the software is unreliable or too slow, then it is practically impossible for them to achieve their goals.

Therefore, software quality is not just about whether the software functionality has been correctly implemented, but also depends on non-functional system attributes as shown in Figure 24.2. These attributes reflect the software dependability, usability, efficiency, and maintainability.

It is not possible for any system to be optimized for all of these attributes. For example, improving security may lead to loss of performance. The quality plan should therefore define the most important quality attributes for the software that is being developed. It may be that efficiency is critical and other factors have to be sacrificed to achieve it. If you have emphasized the importance of efficienty in the quality plan, the engineers working on the development can work together to achieve this. The plan should also include a definition of the quality assessment process.

Figure 24.3 Process-based quality

This process should be an agreed way of assessing whether some quality, such as maintainability or robustness, is present in the product.

Traditional software quality management is based on the assumption that the quality of software is directly related to the quality of the software development process. This assumption comes from manufacturing systems where product quality is intimately related to the production process. A manufacturing process involves configuring, setting up, and operating the machines involved in the process. Once the machines are operating correctly, product quality naturally follows. You measure the quality of the product and change the process until you achieve the quality level that you need. Figure 24.3 illustrates this process-based approach to achieving product quality.

There is a clear link between process and product quality in manufacturing because the process is relatively easy to standardize and monitor. Once manufacturing systems are calibrated, they can be run again and again to output high-quality products. However, software is designed rather than manufactured, and the relationship between process quality and product quality is more complex. Software design is a creative process, so the influence of individual skills and experience is significant. External factors, such as the novelty of an application or commercial pressure for an early product release, also affect product quality irrespective of the process used.

Without doubt, the development process used has a significant influence on the quality of the software, and good processes are more likely to lead to good quality software. Process quality management and improvement can result in fewer defects in the software being developed. However, it is difficult to assess software quality attributes, such as reliability and maintainability, without using the software for a long period. Consequently, it is hard to tell how process characteristics influence these attributes. Furthermore, because of the role of design and creativity in the software process, process standardization can sometimes stifle creativity, which may lead to poorer rather than better quality software.

Defined processes are important, but quality managers should also aim to develop a "quality culture" in which everyone responsible for software development is committed to achieving a high level of product quality. They should encourage teams to take responsibility for the quality of their work and to develop new approaches to quality improvement. While standards and procedures are the basis of quality management, good-quality managers recognize that there are intangible aspects to software quality (elegance, readability, etc.) that cannot be embodied in standards. They should support people who are interested in the intangible aspects of quality and encourage professional behavior in all team members.

Documentation standards

Project documents are a tangible way of describing the different representations of a software system (requirements, UML, code, etc.) and its production process. Documentation standards define the organization of different types of document as well as the document format. They are important because they make it easier to check that important material has not been omitted from documents and ensure that project documents have a common "look and feel." Standards may be developed for the process of writing documents, for the documents themselves and for document exchange.

http://software-engineering-book.com/web/documentation-standards/

24.2 Software standards

Software standards play an important role in plan-based software quality management. As I have discussed, an important part of quality assurance is the definition or selection of standards that should apply to the software development process or software product. As part of this process, tools and methods to support the use of these standards may also be chosen. Once standards have been selected for use, project-specific processes have to be defined to monitor the use of the standards and check that they have been followed.

Software standards are important for three reasons:

1. Standards capture wisdom that is of value to the organization. They are based on knowledge about the best or most appropriate practice for the company. This knowledge is often acquired only after a great deal of trial and error. Building it into a standard helps the company reuse this experience and avoid previous mistakes.
2. Standards provide a framework for defining what quality means in a particular setting. As I have discussed, software quality is subjective, and by using standards you establish a basis for deciding if a required level of quality has been achieved. Of course, this depends on setting standards that reflect user expectations for software dependability, usability, and performance.
3. Standards assist continuity when work carried out by one person is taken up and continued by another. Standards ensure that all engineers within an organization adopt the same practices. Consequently, the learning effort required when starting new work is reduced.

Two related types of software engineering standard may be defined and used in software quality management:

1. *Product standards* These apply to the software product being developed. They include document standards, such as the structure of requirements documents, documentation standards, such as a standard comment header for an object class definition, and coding standards, which define how a programming language should be used.

Product standards	Process standards
Design review form	Design review conduct
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Figure 24.4 Product and process standards

2. *Process standards* These define the processes that should be followed during software development. They should encapsulate good development practice. Process standards may include definitions of specification, design, and validation processes, process support tools, and a description of the documents that should be written during these processes.

Examples of product and process standards that may be used are shown in Figure 24.4.

Standards have to deliver value, in the form of increased product quality. There is no point in defining standards that are expensive in terms of time and effort to apply that only lead to marginal improvements in quality. Product standards have to be designed so that they can be applied and checked in a cost-effective way, and process standards should include the definition of processes that check if product standards have been followed.

The software engineering standards that are used within a company are usually adapted from broader national or international standards. National and international standards have been developed covering software engineering terminology, programming languages such as Java and C++, notations such as charting symbols, procedures for deriving and writing software requirements, quality assurance procedures, and software verification and validation processes (IEEE 2003). More specialized standards have been developed for safety and security critical systems.

Software engineers sometimes consider standards to be overprescriptive and irrelevant to the technical activity of software development. This is particularly likely when project standards require tedious documentation and work recording. Although they usually agree about the general need for standards, engineers often find good reasons why standards are not necessarily appropriate to their particular project. Quality managers who set the standards should therefore consider possible actions to convince engineers of the value of standards:

1. *Involve software engineers in the selection of product standards* If developers understand why standards have been selected, they are more likely to be committed to these standards. Ideally, the standards document should not just set out the standard to be followed but should also include commentary explaining why standardization decisions have been made.

2. *Review and modify standards regularly to reflect changing technologies*
   Standards are expensive to develop, and they tend to be enshrined in a company standards handbook. Because of the costs and discussion required, there is often a reluctance to change them. A standards handbook is essential, but it should evolve to reflect changing circumstances and technology.

3. *Make sure that tool support is available to support standards-based development*
   Developers often find standards to be a bugbear when conformance to them involves tedious manual work that could be done by a software tool. If tool support is available, standards can be followed without much extra effort. For example, program layout standards can be defined and implemented by a syntax-directed program editing system.

Different types of software need different development processes, so standards have to be adaptable. There is no point in prescribing a particular way of working if it is inappropriate for a project or project team. Each project manager should have the authority to modify process standards according to individual circumstances. However, when changes are made, it is important to ensure that these changes do not lead to a loss of product quality.

The project manager and the quality manager can avoid the problems of inappropriate standards by careful quality planning early in the project. They should decide which of the organizational standards should be used without change, which should be modified, and which should be ignored. New standards may have to be created in response to customer or project requirements. For example, standards for formal specifications may be required if these standards have not been used in previous projects.

24.2.1 The ISO 9001 standards framework

The international set of standards used in the development of quality management systems in all industries is called ISO 9000. ISO 9000 standards can be applied to a range of organizations from manufacturing through to service industries. ISO 9001, the most general of these standards, applies to organizations that design, develop, and maintain products, including software. The ISO 9001 standard was originally developed in 1987. I explain the 2008 version of the standard here, but the standard may change in 2015 when a new version is scheduled for release.

The ISO 9001 standard is not a standard for software development but rather is a framework for developing software standards. It sets out general quality principles, describes quality processes in general, and lays out the organizational standards and procedures that should be defined. These should be documented in an organizational quality manual.

A major revision of the ISO 9001 standard in 2000 reoriented the standard around nine core processes (Figure 24.5). If an organization is to be ISO 9001 conformant, it must document how its processes relate to these core processes. It must also define and maintain records demonstrating that the defined organizational processes have

Figure 24.5 ISO 9001 core processes

been followed. The company quality manual should describe the relevant processes and the process data that has to be collected and maintained.

The ISO 9001 standard does not define or prescribe the specific quality processes that a company should use. To be conformant with ISO 9001, a company must define the types of process shown in Figure 24.5 and have procedures in place demonstrating that its quality processes are being followed. This allows flexibility across industrial sectors and company sizes.

Quality standards can be defined that are appropriate for the type of software being developed. Small companies can have simple processes without much documentation and still be ISO 9001 compliant. However, this flexibility means that you cannot make assumptions about the similarities or differences between the processes in different ISO 9001–compliant companies. Some companies may have very rigid quality processes that keep detailed records while others may be much less formal, with minimal additional documentation.

The relationships between ISO 9001, organizational quality manuals, and individual project quality plans are shown in Figure 24.6. This diagram has been adapted from a model given by Ince (Ince 1994), who explains how the general ISO 9001 standard can be used as a basis for software quality management processes. Bamford and Deibler (Bamford and Deibler 2003) explain how the later ISO 9001: 2000 standard can be applied in software companies.

Some software customers demand that their suppliers be ISO 9001 certified. The customers can then be confident that the software development company has an approved quality management system in place. Independent accreditation authorities examine the quality management processes and process documentation and decide if these processes cover all of the areas specified in ISO 9001. If so, they certify that a company's quality processes, as defined in the quality manual, conform to the ISO 9001 standard.

Some people mistakenly think that ISO 9001 certification means that the quality of the software produced by certified companies will always be better than that from

Figure 24.6 ISO 9001 and quality management

uncertified companies. The ISO 9001 standard focuses on ensuring that the organization has quality management procedures in place and that it follows these procedures. There is no guarantee that ISO 9001 certified companies use the best software development practices or that their processes lead to high-quality software.

The ISO 9001 certification is inadequate, in my view, because it defines quality to be the conformance to standards. It takes no account of quality as experienced by users of the software. For example, a company could define test coverage standards specifying that all methods in objects must be called at least once. Unfortunately, this standard can be met by incomplete software testing that does not include tests with different method parameters. As long as the defined testing procedures are followed and test records are maintained, the company could be ISO 9001 certified.

24.3 Reviews and inspections

Reviews and inspections are quality assurance activities that check the quality of project deliverables. This involves checking the software, its documentation, and records of the process to discover errors and omissions as well as standards violations. As I explained in Chapter 8, reviews and inspections are used alongside program testing as part of the general process of software verification and validation.

During a review, several people examine the software and its associated documentation, looking for potential problems and nonconformance with standards. The review team makes informed judgments about the level of quality of the software or project documents. Project managers may then use these assessments to make planning decisions and allocate resources to the development process.

Quality reviews are based on documents that have been produced during the software development process. As well as software specifications, designs, code, process models, test plans, configuration management procedures, process standards,

Figure 24.7 The software review process

and user manuals may all be reviewed. The review should check the consistency and completeness of the documents or code under review and, if standards have been defined, make sure that these quality standards have been followed.

Reviews are not just about checking conformance to standards. They are also used to help discover problems and omissions in the software or project documentation. The conclusions of the review should be formally recorded as part of the quality management process. If problems have been discovered, the reviewers' comments should be passed to the author of the software or whoever is responsible for correcting errors or omissions.

The purpose of reviews and inspections is to improve software quality, not to assess the performance of people in the development team. Reviewing is a public process of error detection, compared with the more private component-testing process. Inevitably, mistakes that are made by individuals are revealed to the whole programming team. To ensure that all developers engage constructively with the review process, project managers have to be sensitive to individual concerns. They must develop a working culture that provides support without blame when errors are discovered.

Quality reviews are not management progress reviews, although information about the software quality may be used in making management decisions. Progress reviews compare the actual progress in a software project against the planned progress. Their prime concern is whether or not the project will deliver useful software on time and on budget. Progress reviews take external factors into account, and changed circumstances may mean that software under development is no longer required or has to be radically changed. Projects that have developed high-quality software may have to be canceled because of changes to the business or its operating environment.

24.3.1 The review process

Although there are many variations in the details of reviews, review processes (Figure 24.7) are structured into three phases:

1. *Pre-review activities* These are preparatory activities that are essential for the review to be effective. Typically, pre-review activities are concerned with review planning and review preparation. Review planning involves setting up a review team, arranging a time and place for the review, and distributing the documents to be reviewed. During review preparation, the team may meet to get an overview of the software to be reviewed. Individual review team members read and understand the software or documents and relevant standards.

Roles in the inspection process

When program inspection was established at IBM (Fagan, 1986), a number of formal roles were defined for members of the inspection team. These included moderator, code reader, and scribe. Other users of inspections have modified these roles, but it is generally accepted that an inspection should involve the code author, an inspector, and a scribe and should be chaired by a moderator.

http://software-engineering-book.com/web/qm-roles

They work independently to find errors, omissions, and departures from standards. Reviewers may supply written comments on the software if they cannot attend the review meeting.

2. *The review meeting* During the review meeting, an author of the document or program being reviewed should "walk through" the document with the review team. The review itself should be relatively short—two hours at most. One team member should chair the review, and another should formally record all review decisions and actions to be taken. During the review, the chair is responsible for ensuring that all submitted comments are considered. The review chair should sign a record of comments and actions agreed during the review.

3. *Post-review activities* After a review meeting has ended, the issues and problems raised during the review must be addressed. Actions may involve fixing software bugs, refactoring software so that it conforms to quality standards, or rewriting documents. Sometimes the problems discovered in a quality review are such that a management review is also necessary to decide if more resources should be made available to correct them. After changes have been made, the review chair may check that all the review comments have been taken into account. Sometimes a further review will be required to check that the changes made cover all of the previous review comments.

Review teams should normally have a core of three to four people who are selected as principal reviewers. One member should be an experienced designer who will take the responsibility for making significant technical decisions. The principal reviewers may invite other project members, such as the designers of related subsystems, to contribute to the review. They may not be involved in reviewing the whole document but should concentrate on those sections that affect their work. Alternatively, the review team may circulate the document and ask for written comments from a broad spectrum of project members. The project manager need not be involved in the review, unless problems are anticipated that require changes to the project plan.

The processes suggested for reviews assume that the review team has a face-to-face meeting to discuss the software or documents they are reviewing. However, project teams are now often distributed, sometimes across countries or continents, so it is impractical for team members to meet face to face. Remote reviewing can be supported using shared documents where each review team member can annotate the document with their comments. Face-to-face meetings may be impossible

because of work schedules or the fact that people work in different time zones. The review chair is responsible for coordinating comments and for discussing changes individually with the review team members.

24.3.2 Program inspections

Program inspections are peer reviews where team members collaborate to find bugs in the program that is being developed. As I discussed in Chapter 8, inspections may be part of the software verification and validation processes. They complement testing as they do not require the program to be executed. Incomplete versions of the system can be verified, and representations such as UML models can be checked. Program tests may be reviewed. Test reviews often find problems with tests and so improve their effectiveness in detecting program bugs.

Program inspections involve team members from different backgrounds who make a careful, line-by-line review of the program source code. They look for defects and problems and describe them at an inspection meeting. Defects may be logical errors, anomalies in the code that might indicate an erroneous condition or features that have been omitted from the code. The review team examines the design models or the program code in detail and highlights anomalies and problems for repair.

During an inspection, a checklist of common programming errors is often used to focus the search for bugs. This checklist may be based on examples from books or from knowledge of defects that are common in a particular application domain. You use different checklists for different programming languages because each language has its own characteristic errors. Humphrey (Humphrey, 1989), in a comprehensive discussion of inspections, gives a number of examples of inspection checklists.

Possible checks that might be made during the inspection process are shown in Figure 24.8. Organizations should develop their own inspection checklists based on local standards and practices. These checklists should be regularly updated, as new types of defects are found. The items in the checklist vary according to programming language because of the different levels of checking that are possible at compile-time. For example, a Java compiler checks that functions have the correct number of parameters; a C compiler does not.

Companies that use inspections have found that they are effective in finding bugs. In early work, Fagan (Fagan 1986) reported that more than 60% of the errors in a program were detected using informal program inspections. McConnell (McConnell 2004) compares unit testing, where the defect detection rate is about 25%, with inspections, where the defect detection rate was 60%. These comparisons were made before widespread automated testing. We don't know how inspections compare to this approach.

In spite of their well-publicized cost-effectiveness, many software development companies are reluctant to use inspections or peer reviews. Software engineers with experience in program testing are sometimes unwilling to accept the fact that inspections can be more effective for defect detection than testing. Managers may be suspicious because inspections require additional costs during design and development. They may not want to take the risk that there will be no corresponding savings in program testing costs.

Fault class	Inspection check
Data faults	■ Are all program variables initialized before their values are used? ■ Have all constants been named? ■ Should the upper bound of arrays be equal to the size of the array or Size −1? ■ If character strings are used, is a delimiter explicitly assigned? ■ Is there any possibility of buffer overflow?
Control faults	■ For each conditional statement, is the condition correct? ■ Is each loop certain to terminate? ■ Are compound statements correctly bracketed? ■ In case statements, are all possible cases accounted for? ■ If a break is required after each case in case statements, has it been included?
Input/output faults	■ Are all input variables used? ■ Are all output variables assigned a value before they are output? ■ Can unexpected inputs cause corruption?
Interface faults	■ Do all function and method calls have the correct number of parameters? ■ Do formal and actual parameter types match? ■ Are the parameters in the right order? ■ If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	■ If a linked structure is modified, have all links been correctly reassigned? ■ If dynamic storage is used, has space been allocated correctly? ■ Is space explicitly de-allocated after it is no longer required?
Exception management faults	■ Have all possible error conditions been taken into account?

Figure 24.8 An inspection checklist

24.4 Quality management and agile development

Agile methods of software engineering focus on the development of code. They minimize documentation and processes that are not directly concerned with code development and emphasize the importance of informal communications among team members rather than communications based on project documents. Quality, in agile development, means code quality and practices such as refactoring, and test-driven development are used to ensure that high-quality code is produced.

Quality management in agile development is informal rather than document-based. It relies on establishing a quality culture, where all team members feel responsible for software quality and take actions to ensure that quality is maintained. The agile community is fundamentally opposed to what it sees as the bureaucratic overhead of standards-based approaches and quality processes as embodied in ISO 9001. Companies that use agile development methods are rarely concerned with ISO 9001 certification.

In agile development, quality management is based on shared good practice rather than formal documentation. Some examples of this good practice are:

1. *Check before check-in* Programmers are responsible for organizing their own code reviews with other team members before the code is checked in to the build system.

2. *Never break the build* It is not acceptable for team members to check in code that causes the system as a whole to fail. Therefore, individuals have to test their code changes against the whole system and be confident that these codes work as expected. If the build is broken, the person responsible is expected to give top priority to fixing the problem.

3. *Fix problems when you see them* The code of the system belongs to the team rather than to individuals. Therefore, if a programmer discovers problems or obscurities in code developed by someone else, he or she can fix these problems directly rather than referring them back to the original developer.

Agile processes rarely use formal inspection or review processes. In Scrum, the development team meets after each iteration to discuss quality issues and problems. The team may decide on changes to the way they work to avoid any quality problems that have emerged. A collective decision may be made to focus on refactoring and quality improvement during a sprint rather than the addition of new system functionality.

Code reviews may be the responsibility of individuals (check before check-in) or may rely on the use of pair programming. As I discussed in Chapter 3, pair programming is an approach in which two people are responsible for code development and work together to achieve it. Code developed by an individual is therefore constantly being examined and reviewed by another team member. Two people look at every line of code and check it before it is accepted.

Pair programming leads to a deep knowledge of a program, as both programmers have to understand the program in detail to continue development. This depth of knowledge is sometimes difficult to achieve in other inspection processes, and so pair programming can find bugs that sometimes would not be discovered in formal inspections. However, the two people involved cannot be as objective as an external inspection team inasmuch as they are examining their own work. Potential problems are:

1. *Mutual misunderstandings* Both members of a pair may make the same mistake in understanding the system requirements. Discussions may reinforce these errors.

2. *Pair reputation* Pairs may be reluctant to look for errors because they do not want to slow down the progress of the project.

3. *Working relationships* The pair's ability to discover defects is likely to be compromised by their close working relationship that often leads to reluctance to criticize work partners.

The informal approach to quality management adopted in agile methods is particularly effective for software product development where the company developing the software also controls its specification. There is no need to deliver quality reports to an external customer, nor is there any need to integrate with other quality management teams. However, when a large system is being developed for an

external customer, agile approaches to quality management with minimal documentation may be impractical:

1. If the customer is a large company, it may have its own quality management processes and may expect the software development company to report on progress in a way that is compatible with these processes. Therefore, the development team may have to produce a formal quality plan and quality reports as required by the customer.
2. Where several geographically distributed teams are involved in development, perhaps from different companies, then informal communications may be impractical. Different companies may have different approaches to quality management, and you may have to agree to produce some formal documentation.
3. For long-lifetime systems, the team involved in development will change over time. If there is no documentation, new team members may find it impossible to understand why development decisions have been made.

Consequently, the informal approach to quality management in agile methods may have to be adapted so that some quality documentation and processes are introduced. Generally, this approach is integrated with the iterative development process. Instead of developing software, one of the sprints or iterations should focus on producing essential software documentation.

24.5 Software measurement

Software measurement is concerned with quantifying some attribute of a software system such as its complexity or its reliability. By comparing the measured values to each other and to the standards that apply across an organization, you may be able to draw conclusions about the quality of software or assess the effectiveness of software processes, tools, and methods. In an ideal world, quality management could rely on measurements of attributes that affect the software quality. You could then objectively assess process and tool changes that aim to improve software quality.

For example, say you work in a company that plans to introduce a new software-testing tool. Before introducing the tool, you record the number of software defects discovered in a given time. This is a baseline for assessing the effectiveness of the tool. After using the tool for some time, you repeat this process. If more defects have been found in the same amount of time, after the tool has been introduced, then you may decide that it provides useful support for the software validation process.

The long-term goal of software measurement is to use measurement to make judgments about software quality. Ideally, a system could be assessed using a range of metrics to measure its attributes. From the measurements made, a value for the quality of the system could be inferred. If the software had reached a required quality threshold, then it could be approved without review. When appropriate, the measurement tools might also highlight areas of the software that could be improved.

Figure 24.9 Predictor and control measurements

However, we are still a long way from this ideal situation, and automated quality assessment is unlikely to become a reality in the near future.

A software metric is a characteristic of a software system, system documentation, or development process that can be objectively measured. Examples of metrics include the size of a product in lines of code, the Fog index, which is a measure of the readability of narrative text, the number of reported faults in a delivered software product, and the number of person-days required to develop a system component.

Software metrics may be either control metrics or predictor metrics. As the names imply, control metrics support process management, and predictor metrics help you predict characteristics of the software. Control metrics are usually associated with software processes. Examples of control or process metrics are the average effort and the time required to repair reported defects. Three kinds of process metrics can be used:

1. *The time taken for a particular process to be completed* This can be the total time devoted to the process, calendar time, the time spent on the process by particular engineers, and so on.
2. *The resources required for a particular process* Resources might include total effort in person-days, travel costs, or computer resources.
3. *The number of occurrences of a particular event* Examples of events that might be monitored include the number of defects discovered during code inspection, the number of requirements changes requested, the number of bug reports in a delivered system, and the average number of lines of code modified in response to a requirements change.

Predictor metrics (sometimes called product metrics) are associated with the software itself. Examples of predictor metrics are the cyclomatic complexity of a module, the average length of identifiers in a program, and the number of attributes and operations associated with object classes in a design. Both control and predictor metrics may influence management decision making as shown in Figure 24.9. Managers use process measurements to decide if process changes should be made and predictor metrics to decide if software changes are necessary and if the software is ready for release.

Figure 24.10 Relationships between internal and external software attributes

In this chapter, I focus on predictor metrics, whose values are automatically assessed by analyzing code or documents. I discuss control metrics and how they are used in process improvement in web Chapter 26.

Measurements of a software system may be used in two ways:

1. *To assign a value to system quality attributes* By measuring the characteristics of system components and then aggregating these measurements, you may be able to assess system quality attributes, such as maintainability.
2. *To identify the system components whose quality is substandard* Measurements can identify individual components with characteristics that deviate from the norm. For example, you can measure components to discover those with the highest complexity. These components are most likely to contain bugs because the complexity makes it more likely that the component developer has made mistakes.

It is difficult to make direct measurements of many of the software quality attributes shown in Figure 24.2. Quality attributes such as maintainability, understandability, and usability are external attributes that relate to how developers and users experience the software. They are affected by subjective factors, such as user experience and education, and they cannot therefore be measured objectively. To make a judgment about these attributes, you have to measure some internal attributes of the software (such as its size and complexity) and assume that these are related to the quality characteristics that you are concerned with.

Figure 24.10 shows some external software quality attributes and internal attributes that could, intuitively, be related to them. The diagram suggests that there may be relationships between external and internal attributes, but it does not say how these attributes are related. Kitchenham (Kitchenham 1990) suggested that if the measure of the internal attribute is to be a useful predictor of the external software characteristic, three conditions must hold:

1. The internal attribute must be measured accurately. However, measurement is not always straightforward and may require specially developed tools.
2. A relationship must exist between the attribute that can be measured and the external quality attribute that is of interest. That is, the value of the quality attribute must be related, in some way, to the value of the attribute than can be measured.
3. This relationship between the internal and external attributes must be understood, validated, and expressed in terms of a formula or model. Model formulation involves identifying the functional form of the model (linear, exponential, etc.) by analysis of collected data, identifying the parameters that are to be included in the model and calibrating these parameters using existing data.

Recent work in the area of software analytics (Zhang et al. 2013) has used data-mining and machine-learning techniques to analyze repositories of software product and process data. The idea behind software analytics (Menzies and Zimmermann 2013) is that we do not, in fact, need a model that reflects the relationships between software quality and collected data. Rather, if there is enough data, correlations can be discovered and predictions made about software attributes. I discuss software analytics in Section 24.5.4.

We have very little published information about systematic software measurement in industry. Many companies do collect information about their software, such as the number of requirements change requests or the number of defects discovered in testing. However, it is not clear if they then use these measurements systematically to compare software products and processes or assess the impact of changes to software processes and tools. There are several reasons why this is difficult:

1. It is impossible to quantify the return on investment of introducing an organizational metrics or software analytics program. We have seen significant improvements in software quality over the past few years without the use of metrics, so it is difficult to justify the initial costs of introducing systematic software measurement and assessment.
2. There are no standards for software metrics or standardized processes for measurement and analysis. Many companies are reluctant to introduce measurement programs until such standards and supporting tools are available.
3. Measurement may require the development and maintenance of specialized software tools. It is difficult to justify the costs of tool development when the returns from measurement are unknown.
4. In many companies, software processes are not standardized and are poorly defined and controlled. As such, there is too much process variability within the same company for measurements to be used in a meaningful way.
5. Much of the research on software measurement and metrics has focused on code-based metrics and plan-driven development processes. However, more and more software is now developed by reusing and configuring existing application

systems, or by using agile methods. We don't know how previous research on metrics applies to these software development techniques.

6. Introducing measurement adds overhead to processes. This contradicts the aims of agile methods, which recommend the elimination of process activities that are not directly related to program development. Companies that have adopted agile methods are therefore not likely to adopt a metrics program.

Software measurement and metrics are the basis of empirical software engineering. In this research area, experiments on software systems and the collection of data about real projects have been used to form and validate hypotheses about software engineering methods and techniques. Researchers working in this area argue that we can be confident of the value of software engineering methods and techniques only if we can provide concrete evidence that they actually provide the benefits their inventors suggest.

However, research on empirical software engineering has not had a significant impact on software engineering practice. It is difficult to relate generic research to an individual project that differs from the research study. Many local factors are likely to be more important than general empirical results. For this reason, researchers in software analytics argue that analysts should not try to draw general conclusions but should provide analyses of the data for specific systems.

24.5.1 Product metrics

Product metrics are predictor metrics used to quantify internal attributes of a software system. Examples of product metrics include the system size, measured in lines of code, or the number of methods associated with each object class. Unfortunately, as I have explained earlier in this section, software characteristics that can be easily measured, such as size and cyclomatic complexity, do not have a clear and consistent relationship with quality attributes such as understandability and maintainability. The relationships vary depending on the development processes and technology used and the type of system that is being developed.

Product metrics fall into two classes:

1. *Dynamic metrics*, which are collected by measurements made of a program in execution. These metrics can be collected during system testing or after the system has gone into use. An example might be the number of bug reports or the time taken to complete a computation.

2. *Static metrics*, which are collected by measurements made of representations of the system, such as the design, program, or documentation. Examples of static metrics are shown in Figure 24.11.

These types of metrics are related to different quality attributes. Dynamic metrics help to assess the efficiency and reliability of a system. Static metrics help assess the complexity, understandability, and maintainability of a system or its components.

Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.
Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

Figure 24.11 Static software product metrics

A clear relationship usually exists between dynamic metrics and software quality characteristics. It is fairly easy to measure the execution time required for particular functions and to assess the time required to start up a system. These functions relate directly to the system's efficiency. Similarly, the number of system failures and the type of failure can be logged and related directly to the reliability of the software. I have explained how reliability can be measured in Chapter 12.

Static metrics, as shown in Figure 24.11, have an indirect relationship with quality attributes. A large number of different metrics have been proposed, and many experiments have tried to derive and validate the relationships between these metrics and attributes, such as system complexity and maintainability. None of these experiments have been conclusive, but program size and control complexity appear be the most reliable predictors of understandability, system complexity, and maintainability.

The metrics in Figure 24.11 are applicable to any program, but more specific object-oriented metrics have also been proposed. Figure 24.12 summarizes Chidamber and Kemerer's suite (sometimes called the CK suite) of six object-oriented metrics (Chidamber and Kemerer 1994). Although these metrics were originally proposed in the early 1990s, they are still the most widely used object-oriented (OO) metrics. Some UML design tools automatically collect values for these metrics as UML diagrams are created.

El-Amam's review of object-oriented metrics discussed the CK metrics and other OO metrics (El-Amam 2001). It concluded that there was insufficient evidence to understand how these and other object-oriented metrics relate to external software

Object-oriented metric	Description
Weighted methods per class (WMC)	This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so they cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.
Coupling between object classes (CBO)	Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent. Therefore, it is more likely that changing one class will affect other classes in the program.
Response for a class (RFC)	RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class. Again, RFC is related to complexity. The higher the value for RFC, the more complex a class, and hence the more likely it is that it will include errors.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated, and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

Figure 24.12 The CK object-oriented metrics suite

qualities. This situation has not really changed since his analysis in 2001. We still don't know how to use measurements of object-oriented programs to draw reliable conclusions about their quality.

24.5.2 Software component analysis

A measurement process that may be part of a software quality assessment process is shown in Figure 24.13. Each system component can be analyzed separately using a range of metrics. The values of these metrics may then be compared for different components and, perhaps, with historical measurement data collected on previous projects. Anomalous measurements, which deviate significantly from the norm, usually indicate problems with the quality of these components.

The key stages in this component measurement process are:

1. *Choose measurements to be made* The questions that the measurement is intended to answer should be formulated and the measurements required to

Figure 24.13 The process of product measurement

answer these questions defined. Measurements that are not directly relevant to these questions need not be collected.

2. *Select components to be assessed* You may not need to assess metric values for all of the components in a software system. Sometimes you can select a representative selection of components for measurement, allowing you to make an overall assessment of system quality. At other times, you may wish to focus on the core components of the system that are in almost constant use. The quality of these components is more important than the quality of components that are infrequently executed.

3. *Measure component characteristics* The selected components are measured, and the associated metric values are computed. This step normally involves processing the component representation (design, code, etc.) using an automated data collection tool. This tool may be specially written or may be a feature of design tools that are already in use.

4. *Identify anomalous measurements* After the component measurements have been made, you then compare them with each other and to previous measurements that have been recorded in a measurement database. You should look for unusually high or low values for each metric, as these suggest that there could be problems with the component exhibiting these values.

5. *Analyze anomalous components* When you have identified components that have anomalous values for your chosen metrics, you should examine them to decide whether or not these anomalous metric values mean that the quality of the component is compromised. An anomalous metric value for complexity (say) does not necessarily mean a poor-quality component. There may be some other reason for the high value, so there may not be any component quality problems.

If possible, you should maintain all collected data as an organizational resource and keep historical records of all projects even when data has not been used during a particular project. Once a sufficiently large measurement database has been established, you can then make comparisons of software quality across projects and validate the relations between internal component attributes and quality characteristics.

24.5.3 Measurement ambiguity

When you collect quantitative data about software and software processes, you have to analyze that data to understand its meaning. It is easy to misinterpret data and to make incorrect inferences. You cannot simply look at the data on its own. You must also consider the context in which the data is collected.

To illustrate how collected data can be interpreted in different ways, consider the following scenario, which is concerned with the number of change requests made by a system's users:

A manager decides to measure the number of change requests submitted by customers based on an assumption that there is a relationship between these change requests and product usability and suitability. She assumes that the higher the number of change requests, the less the software meets the needs of the customer.

Handling change requests and changing the software are expensive. The organization therefore decides to modify its process with the aim of improving customer satisfaction and, at the same time, reducing the costs of making changes. The intent is that the process changes will result in better products and fewer change requests. Processes are changed to increase customer involvement in the software design process. Beta testing of all products is introduced, and customer-requested modifications are incorporated in the delivered product.

After the process changes have been made, the measurement of change requests continues. New versions of products, developed with the modified process, are delivered. In some cases, the number of change requests is reduced; in others, it is increased. The manager is baffled and finds it impossible to understand the effects of the process changes on the product quality.

To understand why this kind of ambiguity can occur, you have to understand why users might make change requests:

1. The software is not good enough and does not do what customers want it to do. They therefore request changes to deliver the functionality they require.
2. Alternatively, the software may be very good, and so it is widely and heavily used. Change requests may be generated because many software users creatively think of new things that could be done with the software.

Increasing the customer's involvement in the process may reduce the number of change requests for products where the customers were unhappy. The process changes have been effective and have made the software more usable and suitable. Alternatively, however, the process changes may not have worked, and customers may have decided to look for an alternative system. The number of change requests might decrease because the product has lost market share to a rival product and there are consequently fewer product users.

On the other hand, the process changes might lead to many new, happy customers who wish to participate in the product development process. They therefore generate more change requests. Changes to the process of handling change requests may contribute to this increase. If the company is more responsive to customers, they may generate more change requests because they know that these requests will be taken seriously. They believe that their suggestions will probably be incorporated in later versions of the software. Alternatively, the number of change requests might have increased because the beta-test sites were not typical of most usage of the program.

To analyze the change request data, you do not simply need to know the number of change requests. You need to know who made the request, how the software is used, and why the request was made. You also need information about external factors such as modifications to the change request procedure or market changes that might have an effect. With this information, you are in a better position to find out if the process changes have been effective in increasing product quality.

This illustrates the difficulties of understanding the effects of changes. The "scientific" approach to this problem is to reduce the number of factors that might affect the measurements made. However, processes and products that are being measured are not insulated from their environment. The business environment is constantly changing, and it is impossible to avoid changes to work practice just because they may make comparisons of data invalid. As such, quantitative data about human activities cannot always be taken at face value. The reasons a measured value changes are often ambiguous. These reasons must be investigated in detail before any conclusions can be drawn from any measurements.

24.5.4 Software analytics

Over the past few years, the notion of "big data analysis" has emerged as a means of discovering insights by automatically mining and analyzing very large volumes of automatically collected data. It is possible to discover relationships between data items that could not be found by manual data analysis and modeling. Software analytics is the application of such techniques to data about software and software processes.

Two factors have made software analytics possible:

1. The automated collection of user data by software product companies when their product is used. If the software fails, information about the failure and the state of the system can be sent over the Internet from the user's computer to servers run by the product developer. As a result, large volumes of data about individual products such as Internet Explorer or Photoshop have become available for analysis.

2. The use of open-source software available on platforms such as Sourceforge and GitHub and open-source repositories of software engineering data (Menzies and Zimmermann 2013). The source code of open-source software is available for automated analysis and can sometimes be linked with data in the open-source repository.

Menzies and Zimmerman (Menzies and Zimmermann 2013) define software analytics as:

> *Software analytics is analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions.*

Menzies and Zimmermann emphasize that the point of analytics is not to derive general theories about software but to identify specific issues that are of interest to software developers and managers. Analytics aims to provide information about these issues in real time so that actions can be taken in response to the information provided by the analysis. In a study of managers at Microsoft, Buse and Zimmermann (Buse and Zimmermann 2012) identified information needs such as how to target testing, inspections, and refactoring, when to release software, and how to understand the needs of software customers.

A range of different data mining and analysis tools can be used for software analytics (Witten, Frank, and Hall 2011). In general, it is impossible to know what are the best analysis tools to use in a particular situation. You have to experiment with several tools to discover which are most effective. Buse and Zimmerman suggest a number of guidelines for tool use:

- Tools should be easy to use, as managers are unlikely to have experience with analysis.
- Tools should run quickly and produce concise outputs rather than large volumes of information.
- Tools should make many measurements using as many parameters as possible. It is impossible to predict in advance what insights might emerge.
- Tools should be interactive and allow managers and developers to explore the analyses. They should recognize that managers and developers are interested in different things. They should not be predictive but should support decision making based on the analysis of past and current data.

Zhang and her colleagues (Zhang et al. 2013) describe an excellent practical application of software analytics for performance debugging. User software was instrumented to collect data on response times and the associated system state. When the response time was greater than expected, this data was sent for analysis. The automated analysis highlighted performance bottlenecks in the software. The development team could then improve the algorithms to eliminate the bottleneck so that performance was improved in a later software release.

At the time of writing, software analytics is immature, and it is too early to say what effect it will have. Not only are there general problems of "big data" processing (Harford 2013), but it will always be the case that our knowledge depends on collected data from large companies. This data is primarily from software products, and it is unclear if the tools and techniques that are appropriate for products can also be used with custom software. Small companies are unlikely to invest in the data collection systems that are required for automated analysis and so they may not be able to use software analytics.

KEY POINTS

- Software quality management is concerned with ensuring that software has a low number of defects and that it reaches the required standards of maintainability, reliability, portability, and so forth. It includes defining standards for processes and products and establishing processes to check that these standards have been followed.
- Software standards are important for quality assurance as they represent an identification of best practice. When developing software, standards provide a solid foundation for building good-quality software.
- Reviews of the software process deliverables involve a team of people who check that quality standards are being followed. Reviews are the most widely used technique for assessing quality.
- In a program inspection or peer review, a small team systematically checks the code. They read the code in detail and look for possible errors and omissions. The problems detected are then discussed at a code review meeting.
- Agile quality management does not usually rely on a separate quality management team. Instead, it relies on establishing a quality culture where the development team works together to improve software quality.
- Software measurement can be used to gather quantitative data about software and the software process. You may be able to use the values of the software metrics that are collected to make inferences about product and process quality.
- Product quality metrics are particularly useful for highlighting anomalous components that may have quality problems. These components should then be analyzed in more detail.
- Software analytics is the automated analysis of large volumes of software product and process data to discover relationships that may provide insights for project managers and developers.

FURTHER READING

Software Quality Assurance: From Theory to Implementation. An excellent, still relevant, book on the principles and practice of software quality assurance. It includes a discussion of standards such as ISO 9001. (D. Galin, Addison-Wesley, 2004).

"Misleading Metrics and Unsound Analyses." An excellent article by leading metrics researchers that discusses the difficulties of understanding what measurements really mean. (B. Kitchenham, R. Jeffrey and C. Connaughton, *IEEE Software*, 24 (2), March–April 2007). http://dx.doi.org/10.1109/MS.2007.49

"A Practical Guide to Implementing an Agile QA Process on Scrum Projects." This slide set presents an overview of how to integrate software quality assurance with agile development using Scrum. (S. Rayhan, 2008). https://www.scrumalliance.org/system/resource_files/0000/0459/agileqa.pdf

"Software Analytics: So What?" This is a good introductory article that explains what software analytics is and why it is increasingly important. It is the introduction to a special issue on software analytics, and you may find several other articles in that issue to be helpful in understanding software analytics. (T. Menzies and T. Zimmermann, *IEEE Software,* 30 (4), July–August 2013). http://dx.doi.org/10.1109/MS.2013.86

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-management/

EXERCISES

24.1. Define the terms quality assurance and quality control. List out the key points included in Humphrey's outline structure for software management.

24.2. Explain how standards may be used to capture organizational wisdom about effective methods of software development. Suggest four types of knowledge that might be captured in organizational standards.

24.3. Discuss the assessment of software quality according to the quality attributes shown in Figure 24.2. You should consider each attribute in turn and explain how it might be assessed

24.4. Briefly describe possible standards that might be used for:

- the use of control constructs in C, C#, or Java;
- reports that might be submitted for a term project in a university;
- the process of making and approving program changes (web Chapter 26); and
- the process of purchasing and installing a new computer.

24.5. Assume you work for an organization that develops database products for individuals and small businesses. This organization is interested in quantifying its software development. Write a report suggesting appropriate metrics and suggest how these can be collected.

24.6. Briefly explain what happens during the software quality review process and the software quality inspection process.

24.7. What problems are likely to arise if formalized program inspections are introduced in a company where some software is developed using agile methods.

24.8. What is a software metric? Define different types of software metrics with examples.

24.9. You work for a software product company and your manager has read an article on software analytics. She asks you to do some research in this area. Survey the literature on analytics and write a short report that summarizes work in software analytics and issues to be considered if analytics is introduced.

24.10 A colleague who is a very good programmer produces software with a low number of defects but consistently ignores organizational quality standards. How should her managers react to this behavior?

REFERENCES

Bamford, R., and W. J. Deibler. 2003. "ISO 9001:2000 for Software and Systems Providers: An Engineering Approach." Boca Raton, FL: CRC Press.

Buse, R. P. L., and T. Zimmermann. 2012. "Information Needs for Software Development Analytics." In *Int. Conf. on Software Engineering*, 987–996. doi:10.1109/ICSE.2012.6227122.

Chidamber, S., and C. Kemerer. 1994. "A Metrics Suite for Object-Oriented Design." *IEEE Trans. on Software Eng.* 20 (6): 476–493. doi:10.1109/32.295895.

El-Amam, K. 2001. "Object-Oriented Metrics: A Review of Theory and Practice." National Research Council of Canada. http://seg.iit.nrc.ca/English/abstracts/NRC44190.html.

Fagan, M. E. 1986. "Advances in Software Inspections." *IEEE Trans. on Software Eng.* SE-12 (7): 744–751. doi:10.1109/TSE.1986.6312976.

Harford, T. 2013. "Big Data: Are We Making a Big Mistake?" *Financial Times*, March 28. http://timharford.com/2014/04/big-data-are-we-making-a-big-mistake/

Humphrey, W. 1989. *Managing the Software Process*. Reading, MA: Addison-Wesley.

IEEE. 2003. *IEEE Software Engineering Standards Collection on CD-ROM*. Los Alamitos, CA: IEEE Computer Society Press.

Ince, D. 1994. *ISO 9001 and Software Quality Assurance*. London: McGraw-Hill.

Kitchenham, B. 1990. "Software Development Cost Models." In *Software Reliability Handbook*, edited by P. Rook, 487–517. Amsterdam: Elsevier.

McConnell, S. 2004. *Code Complete: A Practical Handbook of Software Construction, 2nd ed.* Seattle, WA: Microsoft Press.

Menzies, T., and T. Zimmermann. 2013. "Software Analytics: So What?" *IEEE Software* 30 (4): 31–37. doi:10.1109/MS.2013.86.

Witten, I. H., E. Frank, and M. A. Hall. 2011. *Data Mining: Practical Machine Learning Tools and Techniques*. Burlington, MA: Morgan Kaufmann.

Zhang, D, S. Han, Y. Dang, J-G. Lou, H. Zhang, and T. Xie. 2013. "Software Analytics in Practice." *IEEE Software* 30 (5): 30–37. doi:10.1109/MS.2013.94.

25

Configuration management

Objectives

The objective of this chapter is to introduce you to software configuration management processes and tools. When you have read the chapter, you will:

- know the essential functionality that should be provided by a version control system, and how this is realized in centralized and distributed systems;
- understand the challenges of system building and the benefits of continuous integration and system building;
- understand why software change management is important and the essential activities in the change management process;
- understand the basics of software release management and how it differs from version management.

Contents

25.1 Version management
25.2 System building
25.3 Change management
25.4 Release management

Software systems are constantly changing during development and use. Bugs are discovered and have to be fixed. System requirements change, and you have to implement these changes in a new version of the system. New versions of hardware and system platforms are released, and you have to adapt your systems to work with them. Competitors introduce new features in their system that you have to match. As changes are made to the software, a new version of a system is created. Most systems, therefore, can be thought of as a set of versions, each of which may have to be maintained and managed.

Configuration management (CM) is concerned with the policies, processes, and tools for managing changing software systems (Aiello and Sachs 2011). You need to manage evolving systems because it is easy to lose track of what changes and component versions have been incorporated into each system version. Versions implement proposals for change, corrections of faults, and adaptations for different hardware and operating systems. Several versions may be under development and in use at the same time. If you don't have effective configuration management procedures in place, you may waste effort modifying the wrong version of a system, delivering the wrong version of a system to customers, or forgetting where the software source code for a particular version of the system or component is stored.

Configuration management is useful for individual projects as it is easy for one person to forget what changes have been made. It is essential for team projects where several developers are working at the same time on a software system. Sometimes these developers are all working in the same place, but, increasingly, development teams are distributed with members in different locations across the world. The configuration management system provides team members with access to the system being developed and manages the changes that they make to the code.

The configuration management of a software system product involves four closely related activities (Figure 25.1):

1. *Version control* This involves keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.

2. *System building* This is the process of assembling program components, data, and libraries, then compiling and linking these to create an executable system.

3. *Change management* This involves keeping track of requests for changes to delivered software from customers and developers, working out the costs and impact of making these changes, and deciding if and when the changes should be implemented.

4. *Release management* This involves preparing software for external release and keeping track of the system versions that have been released for customer use.

Because of the large volume of information to be managed and the relationships between configuration items, tool support is essential for configuration management. Configuration management tools are used to store versions of system components, build systems from these components, track the releases of system versions to

Figure 25.1 Configuration management activities

customers, and keep track of change proposals. CM tools range from simple tools that support a single configuration management task, such as bug tracking, to integrated environments that support all configuration management activities.

Agile development, where components and systems are changed several times a day, is impossible without using CM tools. The definitive versions of components are held in a shared project repository, and developers copy them into their own workspace. They make changes to the code and then use system-building tools to create a new system on their own computer for testing. Once they are happy with the changes made, they return the modified components to the project repository. This makes the modified components available to other team members.

The development of a software product or custom software system takes place in three distinct phases:

1. *A development phase* where the development team is responsible for managing the software configuration and new functionality is being added to the software. The development team decides on the changes to be made to the system.

2. *A system testing phase* where a version of the system is released internally for testing. This may be the responsibility of a quality management team or an individual or group within the development team. At this stage, no new functionality is added to the system. The changes made at this stage are bug fixes, performance improvements, and security vulnerability repairs. There may be some customer involvement as beta testers during this phase.

3. *A release phase* where the software is released to customers for use. After the release has been distributed, customers may submit bug reports and change requests. New versions of the released system may be developed to repair bugs and vulnerabilities and to include new features suggested by customers.

For large systems, there is never just one "working" version of a system; there are always several versions of the system at different stages of development. Several

Figure 25.2 Multiversion system development

teams may be involved in the development of different system versions. Figure 25.2 shows situations where three versions of a system are being developed:

1. Version 1.5 of the system has been developed to repair bug fixes and improve the performance of the first release of the system. It is the basis of the second system release (R1.1).
2. Version 2.4 is being tested with a view to it becoming release 2.0 of the system. No new features are being added at this stage.
3. Version 3 is a development system where new features are being added in response to change requests from customers and the development team. This will eventually be released as release 3.0.

These different versions have many common components as well as components or component versions that are unique to that system version. The CM system keeps track of the components that are part of each version and includes them as required in the system build.

In large software projects, configuration management is sometimes part of software quality management (covered in Chapter 24). The quality manager is responsible for both quality management and configuration management. When a pre-release version of the software is ready, the development team hands it over to the quality management team. The QM team checks that the system quality is acceptable. If so, it then becomes a controlled system, which means that all changes to the system have to be agreed on and recorded before they are implemented.

Many specialized terms are used in configuration management. Unfortunately, these are not standardized. Military software systems were the first systems in which software CM was used, so the terminology for these systems reflected the processes and terminology used in hardware configuration management. Commercial systems developers did not know about military procedures or terminology and so often invented their own terms. Agile methods have also devised new terminology in order to distinguish the agile approach from traditional CM methods.

Term	Explanation
Baseline	A collection of component versions that make up a system. Baselines are controlled, which means that the component versions used in the baseline cannot be changed. It is always possible to re-create a baseline from its constituent components.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Codeline	A set of versions of a software component and other configuration items on which that component depends.
Configuration (version) control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. Configuration items always have a unique identifier.
Mainline	A sequence of baselines representing different versions of a system.
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Repository	A shared database of versions of software components and meta-information about changes to these components.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions should always have a unique identifier.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.

Figure 25.3 CM terminology

The definition and use of configuration management standards are essential for quality certification in both ISO 9000 and the SEI's capability maturity model (Bamford and Deibler 2003; Chrissis, Konrad, and Shrum 2011). CM standards in a company may be based on generic standards such as IEEE 828-2012, an IEEE standard for configuration management. These standards focus on CM processes and the documents produced during the CM process (IEEE 2012). Using the external standards as a starting point, companies may then develop more detailed, company-specific standards that are tailored to their specific needs. However, agile methods rarely use these standards because of the documentation overhead involved.

25.1 Version management

Version management is the process of keeping track of different versions of software components and the systems in which these components are used. It also involves ensuring that changes made by different developers to these versions do not interfere with each other. In other words, version management is the process of managing codelines and baselines.

Figure 25.4 illustrates the differences between codelines and baselines. A codeline is a sequence of versions of source code, with later versions in the sequence derived from earlier versions. Codelines normally apply to components of systems so that there are different versions of each component. A baseline is a definition of a specific system. The baseline specifies the component versions that are included in the system and identifies the libraries used, configuration files, and other system information. In Figure 25.4, you can see that different baselines use different versions of the components from each codeline. In the diagram, I have shaded the boxes representing components in the baseline definition to indicate that these are actually references to components in a codeline. The mainline is a sequence of system versions developed from an original baseline.

Baselines may be specified using a configuration language in which you define what components should be included in a specific version of a system. It is possible to explicitly specify an individual component version (X.1.2, say) or simply to specify the component identifier (X). If you simply include the component identifier in the configuration description, the most recent version of the component should be used.

Baselines are important because you often have to re-create an individual version of a system. For example, a product line may be instantiated so that there are specific system versions for each system customer. You may have to re-create the version delivered to a customer if they report bugs in their system that have to be repaired.

Version control (VC) systems identify, store, and control access to the different versions of components. There are two types of modern version control system:

1. *Centralized systems*, where a single master repository maintains all versions of the software components that are being developed. Subversion (Pilato, Collins-Sussman, and Fitzpatrick 2008) is a widely used example of a centralized VC system.

2. *Distributed systems*, where multiple versions of the component repository exist at the same time. Git (Loeliger and McCullough 2012), is a widely used example of a distributed VC system.

Centralized and distributed VC systems provide comparable functionality but implement this functionality in different ways. Key features of these systems include:

1. *Version and release identification* Managed versions of a component are assigned unique identifiers when they are submitted to the system. These identifiers allow different versions of the same component to be managed, without changing the component name. Versions may also be assigned attributes, with the set of attributes used to uniquely identify each version.

Figure 25.4 Codelines and baselines

2. *Change history recording* The VC system keeps records of the changes that have been made to create a new version of a component from an earlier version. In some systems, these changes may be used to select a particular system version. This involves tagging components with keywords describing the changes made. You then use these tags to select the components to be included in a baseline.

3. *Independent development* Different developers may be working on the same component at the same time. The version control system keeps track of components that have been checked out for editing and ensures that changes made to a component by different developers do not interfere.

4. *Project support* A version control system may support the development of several projects, which share components. It is usually possible to check in and check out all of the files associated with a project rather than having to work with one file or directory at a time.

5. *Storage management* Rather than maintain separate copies of all versions of a component, the version control system may use efficient mechanisms to ensure that duplicate copies of identical files are not maintained. Where there are only small differences between files, the VC system may store these differences rather than maintain multiple copies of files. A specific version may be automatically re-created by applying the differences to a master version.

Most software development is a team activity, so several team members often work on the same component at the same time. For example, let's say Alice is making some changes to a system, which involves changing components A, B, and C. At the same time, Bob is working on changes that require making changes to components X, Y, and C. Both Alice and Bob are therefore changing C. It's important to avoid situations where changes interfere with each other—Bob's changes to C overwriting Alice's or vice versa.

To support independent development without interference, all version control systems use the concept of a project repository and a private workspace. The project repository maintains the "master" version of all components, which is used to create baselines for system building. When modifying components, developers copy

Figure 25.5 Check-in and check-out from a centralized version repository

(check-out) these from the repository into their workspace and work on these copies. When they have completed their changes, the changed components are returned (checked-in) to the repository. However, centralized and distributed VC systems support independent development of shared components in different ways.

In centralized systems, developers check out components or directories of components from the project repository into their private workspace and work on these copies in their private workspace. When their changes are complete, they check-in the components back to the repository. This creates a new component version that may then be shared. For an illustration, see Figure 25.5.

Here, Alice has checked out versions A1.0, B1.0, and C1.0. She has worked on these versions and has created new versions A1.1, B1.1, and C1.1. She checks these new versions into the repository. Bob checks out X1.0, Y1.0, and C1.0. He creates new versions of these components and checks them back in to the repository. However, Alice has already created a new version of C, while Bob has been working on it. His check-in therefore creates another version C1.2, so that Alice's changes are not overwritten.

If two or more people are working on a component at the same time, each must check out the component from the repository. If a component has been checked out, the version control system warns other users wanting to check out that component that it has been checked out by someone else. The system will also ensure that when the modified components are checked in, the different versions are assigned different version identifiers and are stored separately.

In a distributed VC system, such as Git, a different approach is used. A "master" repository is created on a server that maintains the code produced by the development team. Instead of simply checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on his or her computer.

Developers work on the files required and maintain the new versions on their private repository on their own computer. When they have finished making changes, they "commit" these changes and update their private server repository. They may then "push" these changes to the project repository or tell the integration manager that changed versions are available. He or she may then "pull" these files to the project repository (see Figure 25.6). In this example, both Bob and Alice have cloned the project repository and have updated files. They have not yet pushed these back to the project repository.

Figure 25.6 Repository cloning

This model of development has a number of advantages:

1. It provides a backup mechanism for the repository. If the repository is corrupted, work can continue and the project repository can be restored from local copies.
2. It allows for offline working so that developers can commit changes if they do not have a network connection.
3. Project support is the default way of working. Developers can compile and test the entire system on their local machines and test the changes they have made.

Distributed version control is essential for open-source development where several people may be working simultaneously on the same system without any central coordination. There is no way for the open-source system "manager" to know when changes will be made. In this case, as well as a private repository on their own computer, developers also maintain a public server repository to which they push new versions of components that they have changed. It is then up to the open-source system "manager" to decide when to pull these changes into the definitive system. This organization is shown in Figure 25.7.

In this example, Charlie is the integration manager for the open-source system. Alice and Bob work independently on system development and clone the definitive project repository (1). As well as their private repositories, both Alice and Bob maintain a public repository on a server than can be accessed by Charlie. When they have made and tested changes, they push the changed versions from their private repositories to their personal public repositories and tell Charlie that these repositories are available (2). Charlie pulls these from their repositories into his

Figure 25.7 Open-source development

own private repository for testing (3). Once he is satisfied that the changes are acceptable, he then updates the definitive project repository (4).

A consequence of the independent development of the same component is that codelines may branch. Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences, as shown in Figure 25.8. This is normal in system development, where different developers work independently on different versions of the source code and change it in different ways. It is generally recommended when working on a system that a new branch should be created so that changes do not accidentally break a working system.

At some stage, it may be necessary to merge codeline branches to create a new version of a component that includes all changes that have been made. This is also shown in Figure 25.8, where component versions 2.1.2 and 2.3 are merged to create version 2.4. If the changes made involve completely different parts of the code, the component versions may be merged automatically by the version control system by combining the code changes. This is the normal mode of operation when new features have been added. These code changes are merged into the master copy of the system. However, the changes made by different developers sometimes overlap. The changes may be incompatible and interfere with each other. In this case, a developer has to check for clashes and make changes to the components to resolve the incompatibilities between the different versions.

When version control systems were first developed, storage management was one of their most important functions. Disk space was expensive, and it was important to

Figure 25.8 Branching and merging

Figure 25.9 Storage management using deltas

minimize the disk space used by the different copies of components. Instead of keeping a complete copy of each version, the system stores a list of differences (deltas) between one version and another. By applying these to a master version (usually the most recent version), a target version can be re-created. This is illustrated in Figure 25.9.

When a new version is created, the system simply stores a delta, a list of differences, between the new version and the older version used to create that new version. In Figure 25.9, the shaded boxes represent earlier versions of a component that are automatically re-created from the most recent component version. Deltas are usually stored as lists of changed lines, and, by applying these automatically, one version of a component can be created from another. As the most recent version of a component will most likely be the one used, most systems store that version in full. The deltas then define how to re-create earlier system versions.

One of the problems with a delta-based approach to storage management is that it can take a long time to apply all of the deltas. As disk storage is now relatively cheap, Git uses an alternative, faster approach. Git does not use deltas but applies a standard compression algorithm to stored files and their associated meta-information. It does not store duplicate copies of files. Retrieving a file simply involves decompressing it, with no need to apply a chain of operations. Git also uses the notion of packfiles where several smaller files are combined into an indexed single file. This reduces the overhead associated with lots of small files. Deltas are used within packfiles to further reduce their size.

25.2 System building

System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, and other information. System-building tools and version control tools must be integrated as the build process takes component versions from the repository managed by the version control system.

System building involves assembling a large amount of information about the software and its operating environment. Therefore, it always makes sense to use an automated build tool to create a system build (Figure 25.10). Notice that you don't just need the source code files that are involved in the build. You may have to link these with externally provided libraries, data files (such as a file of error messages), and configuration files that define the target installation. You may have to specify the versions of

Figure 25.10 System building

the compiler and other software tools that are to be used in the build. Ideally, you should be able to build a complete system with a single command or mouse click.

Tools for system integration and building include some or all of the following features:

1. *Build script generation* The build system should analyze the program that is being built, identify dependent components, and automatically generate a build script (configuration file). The system should also support the manual creation and editing of build scripts.

2. *Version control system integration* The build system should check out the required versions of components from the version control system.

3. *Minimal recompilation* The build system should work out what source code needs to be recompiled and set up compilations if required.

4. *Executable system creation* The build system should link the compiled object code files with each other and with other required files, such as libraries and configuration files, to create an executable system.

5. *Test automation* Some build systems can automatically run automated tests using test automation tools such as JUnit. These check that the build has not been "broken" by changes.

6. *Reporting* The build system should provide reports about the success or failure of the build and the tests that have been run.

7. *Documentation generation* The build system may be able to generate release notes about the build and system help pages.

The build script is a definition of the system to be built. It includes information about components and their dependencies, and the versions of tools used to compile and link the system. The configuration language used to define the build script includes constructs to describe the system components to be included in the build and their dependencies.

Building is a complex process, which is potentially error-prone, as three different system platforms may be involved (Figure 25.11):

1. *The development system*, which includes development tools such as compilers and source code editors. Developers check out code from the version control system into

Figure 25.11 Development, build, and target platforms

a private workspace before making changes to the system. They may wish to build a version of a system for testing in their development environment before committing changes that they have made to the version control system. This involves using local build tools that use checked-out versions of components in the private workspace.

2. *The build server*, which is used to build definitive, executable versions of the system. This server maintains the definitive versions of a system. All of the system developers check in code to the version control system on the build server for system building.

3. *The target environment*, which is the platform on which the system executes. This may be the same type of computer that is used for the development and build systems. However, for real-time and embedded systems, the target environment is often smaller and simpler than the development environment (e.g., a cell phone). For large systems, the target environment may include databases and other application systems that cannot be installed on development machines. In these situations, it is not possible to build and test the system on the development computer or on the build server.

Agile methods recommend that very frequent system builds should be carried out, with automated testing used to discover software problems. Frequent builds are part of a process of continuous integration as shown in Figure 25.12. In keeping with the agile methods notion of making many small changes, continuous integration involves rebuilding the mainline frequently, after small source code changes have been made. The steps in continuous integration are:

1. Extract the mainline system from the VC system into the developer's private workspace.

2. Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken, and you should inform whoever checked in the last baseline system. He or she is responsible for repairing the problem.

3. Make the changes to the system components.

4. Build the system in a private workspace and rerun system tests. If the tests fail, continue editing.

Figure 25.12
Continuous integration

5. Once the system has passed its tests, check it into the build system server but do not commit it as a new system baseline in the VC system.

6. Build the system on the build server and run the tests. Alternatively, if you are using Git, you can pull recent changes from the server to your private workspace. You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.

7. If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

Tools such as Jenkins (Smart 2011) are used to support continuous integration. These tools can be set up to build a system as soon as a developer has completed a repository update.

The advantage of continuous integration is that it allows problems caused by the interactions between different developers to be discovered and repaired as soon as possible. The most recent system in the mainline is the definitive working system. However, although continuous integration is a good idea, it is not always possible to implement this approach to system building:

1. If the system is very large, it may take a long time to build and test, especially if integration with other application systems is involved. It may be impractical to build the system being developed several times per day.

2. If the development platform is different from the target platform, it may not be possible to run system tests in the developer's private workspace. There may be differences in hardware, operating system, or installed software. Therefore, more time is required for testing the system.

For large systems or for systems where the execution platform is not the same as the development platform, continuous integration is usually impossible. In those circumstances, frequent system building is supported using a daily build system:

1. The development organization sets a delivery time (say 2 p.m.) for system components. If developers have new versions of the components that they are writing, they must deliver them by that time. Components may be incomplete but should provide some basic functionality that can be tested.
2. A new version of the system is built from these components by compiling and linking them to form a complete system.
3. This system is then delivered to the testing team, which carries out a set of predefined system tests.
4. Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

The advantages of using frequent builds of software are that the chances of finding problems stemming from component interactions early in the process are increased. Frequent building encourages thorough unit testing of components. Psychologically, developers are put under pressure not to "break the build"; that is, they try to avoid checking in versions of components that cause the whole system to fail. They are therefore reluctant to deliver new component versions that have not been properly tested. Consequently, less time is spent during system testing discovering and coping with software faults that could have been found by the developer.

As compilation is a computationally intensive process, tools to support system building may be designed to minimize the amount of compilation that is required. They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component. Therefore, there has to be a way of unambiguously linking the source code of a component with its equivalent object code.

This linking is accomplished by associating a unique signature with each file where a source code component is stored. The corresponding object code, which has been compiled from the source code, has a related signature. The signature identifies each source code version and is changed when the source code is edited. By comparing the signatures on the source and object code files, it is possible to decide if the source code component was used to generate the object code component.

Two types of signature may be used, as shown in Figure 25.13:

1. *Modification timestamps* The signature on the source code file is the time and date when that file was modified. If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.

   For example, say components Comp.java and Comp.class have modification signatures of 17:03:05:02:14:2014 and 16:58:43:02:14:2014, respectively. This means that the Java code was modified at 3 minutes and 5 seconds past 5 on the 14th of February 2014 and the compiled version was modified at 58 minutes and 43 seconds past 4 on the 14th of February 2014. In this case, the system would automatically recompile Comp.java because the compiled version has an earlier modification date than the most recent version of the component.

Figure 25.13 Linking source and object code

2. *Source code checksums* The signature on the source code file is a checksum calculated from data in the file. A checksum function calculates a unique number using the source text as input. If you change the source code (even by one character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different. The checksum is assigned to the source code just before compilation and uniquely identifies the source file. The build system then tags the generated object code file with the checksum signature. If there is no object code file with the same signature as the source code file to be included in a system, then recompilation of the source code is necessary.

As object code files are not normally versioned, the first approach means that only the most recently compiled object code file is maintained in the system. This is normally related to the source code file by name; that is, it has the same name as the source code file but with a different suffix. Therefore, the source file Comp.Java may generate the object file Comp.class. Because source and object files are linked by name, it is not usually possible to build different versions of a source code component into the same directory at the same time. The compiler would generate object files with the same name, so only the most recently compiled version would be available.

The checksum approach has the advantage of allowing many different versions of the object code of a component to be maintained at the same time. The signature rather than the filename is the link between source and object code. The source code and object code files have the same signature. Therefore, when you recompile a component, it does not overwrite the object code, as would normally be the case when the timestamp is used. Rather, it generates a new object code file and tags it with the source code signature. Parallel compilation is possible, and different versions of a component may be compiled at the same time.

25.3 Change management

Change is a fact of life for large software systems. Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired, and systems have to adapt to changes in their environment. To ensure that the changes are applied

Figure 25.14 The change management process

to the system in a controlled way, you need a set of tool-supported, change management processes. Change management is intended to ensure that the evolution of the system is controlled and that the most urgent and cost-effective changes are prioritized.

Change management is the process of analyzing the costs and benefits of proposed changes, approving those changes that are cost-effective, and tracking which components in the system have been changed. Figure 25.14 is a model of a change management process that shows the main change management activities. This process should come into effect when the software is handed over for release to customers or for deployment within an organization.

Many variants of this process are in use depending on whether the software is a custom system, a product line, or an off-the-shelf product. The size of the company also makes a difference—small companies use a less formal process than large companies that are working with corporate or government customers. However, all change management processes should include some way of checking, costing, and approving changes.

Tools to support change management may be relatively simple issue or bug tracking systems or software that is integrated with a configuration management package for large-scale systems, such as Rational Clearcase. Issue tracking systems allow anyone to report a bug or make a suggestion for a system change, and they keep track of how the development team has responded to the issues. These systems do not impose a process on the users and so can be used in many different settings. More complex systems are built around a process model of the change management process. They

Change Request Form

Project: SICSA/AppProcessing **Number:** 23/02
Change requester: I. Sommerville **Date:** 20/07/12
Requested change: The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

Change analyzer: R. Looek **Analysis date:** 25/07/12
Components affected: ApplicantListDisplay, StatusUpdater

Associated components: StudentDatabase

Change assessment: Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

Change priority: Medium
Change implementation:
Estimated effort: 2 hours
Date to SGA app. team: 28/07/12 **CCB decision date:** 30/07/12
Decision: Accept change. Change to be implemented in Release 1.2
Change implementor: **Date of change:**
Date submitted to QM: **QM decision:**
Date submitted to CM:
Comments:

Figure 25.15 A partially completed change request form

automate the entire process of handling change requests from the initial customer proposal to final change approval and change submission to the development team.

The change management process is initiated when a system stakeholder completes and submits a change request describing the change required to the system. This could be a bug report, where the symptoms of the bug are described, or a request for additional functionality to be added to the system. Some companies handle bug reports and new requirements separately, but, in principle, both are simply change requests. Change requests may be submitted using a change request form (CRF). Stakeholders may be system owners and users, beta testers, developers, or the marketing department of a company.

Electronic change request forms record information that is shared between all groups involved in change management. As the change request is processed, information is added to the CRF to record decisions made at each stage of the process. At any time, it therefore represents a snapshot of the state of the change request. In addition to recording the change required, the CRF records the recommendations regarding the change, the estimated costs of the change, and the dates when the change was requested, approved, implemented, and validated. The CRF may also include a section where a developer outlines how the change may be implemented. Again, the degree of formality in the CRF varies depending on the size and type of organization that is developing the system.

Figure 25.15 is an example of a type of CRF that might be used in a large complex systems engineering project. For smaller projects, I recommend that change requests should be formally recorded; the CRF should focus on describing the

Customers and changes

Agile methods emphasize the importance of involving customers in the change prioritization process. The customer representative helps the team decide on the changes that should be implemented in the next development iteration. While this can be effective for systems that are in development for a single customer, it can be a problem in product development where no real customer is working with the team. In those cases, the team has to make its own decisions on change prioritization.

http://software-engineering-book.com/web/agile-changes/

change required, with less emphasis on implementation issues. System developers decide how to implement the change and estimate the time required to complete the change implementation.

After a change request has been submitted, it is checked to ensure that it is valid. The checker may be from a customer or application support team or, for internal requests, may be a member of the development team. The change request may be rejected at this stage. If the change request is a bug report, the bug may have already been reported and repaired. Sometimes, what people believe to be problems are actually misunderstandings of what the system is expected to do. On occasions, people request features that have already been implemented but that they don't know about. If any of these features are true, the issue is closed and the form is updated with the reason for closure. If it is a valid change request, it is then logged as an outstanding request for subsequent analysis.

For valid change requests, the next stage of the process is change assessment and costing. This function is usually the responsibility of the development or maintenance team as they can work out what is involved in implementing the change. The impact of the change on the rest of the system must be checked. To do this, you have to identify all of the components affected by the change. If making the change means that further changes elsewhere in the system are needed, this will obviously increase the cost of change implementation. Next, the required changes to the system modules are assessed. Finally, the cost of making the change is estimated, taking into account the costs of changing related components.

Following this analysis, a separate group decides if it is cost-effective for the business to make the change to the software. For military and government systems, this group is often called the change control board (CCB). In industry, it may be called something like a "product development group" responsible for making decisions about how a software system should evolve. This group should review and approve all change requests, unless the changes simply involve correcting minor errors on screen displays, web pages, or documents. These small requests should be passed to the development team for immediate implementation.

The CCB or product development group considers the impact of the change from a strategic and organizational rather than a technical point of view. It decides whether the change in question is economically justified, and it prioritizes accepted changes for implementation. Accepted changes are passed back to the development group;

rejected change requests are closed and no further action is taken. The factors that influence the decision on whether or not to implement a change include:

1. *The consequences of not making the change* When assessing a change request, you have to consider what will happen if the change is not implemented. If the change is associated with a reported system failure, the seriousness of that failure has to be taken into account. If the system failure causes the system to crash, this is very serious, and failure to make the change may disrupt the operational use of the system. On the other hand, if the failure has a minor effect, such as incorrect colors on a display, then it is not important to fix the problem quickly. The change should therefore have a low priority.

2. *The benefits of the change* Will the change benefit many users of the system, or will it only benefit the change proposer?

3. *The number of users affected by the change* If only a few users are affected, then the change may be assigned a low priority. In fact, making the change may be inadvisable if it means that the majority of system users have to adapt to it.

4. *The costs of making the change* If making the change affects many system components (hence increasing the chances of introducing new bugs) and/or takes a lot of time to implement, then the change may be rejected.

5. *The product release cycle* If a new version of the software has just been released to customers, it may make sense to delay implementation of the change until the next planned release (see Section 25.4).

Change management for software products (e.g., a CAD system product), rather than custom systems specifically developed for a certain customer, are handled in a different way. In software products, the customer is not directly involved in decisions about system evolution, so the relevance of the change to the customer's business is not an issue. Change requests for these products come from the customer support team, the company marketing team, and the developers themselves. These requests may reflect suggestions and feedback from customers or analyses of what is offered by competing products.

The customer support team may submit change requests associated with bugs that have been discovered and reported by customers after the software has been released. Customers may use a web page or email to report bugs. A bug management team then checks that the bug reports are valid and translates them into formal system change requests. Marketing staff may meet with customers and investigate competitive products. They may suggest changes that should be included to make it easier to sell a new version of a system to new and existing customers. The system developers themselves may have some good ideas about new features that can be added to the system.

The change request process shown in Figure 25.14 is initiated after a system has been released to customers. During development, when new versions of the system are created through daily (or more frequent) system builds, there is no need for a formal change management process. Problems and requested changes are recorded in an issue tracking system and discussed in daily meetings. Changes that only affect individual components are passed directly to the system developer, who either accepts them or makes a case for why they are not required. However, an independent

```
// SICSA project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Object: currentRole
// Author: R. Looek
// Creation date: 13/11/2012
//
// © St Andrews University 2012
//
// Modification history
// Version    Modifier    Date          Change       Reason
// 1.0        J. Jones    11/11/2009    Add header   Submitted to CM
// 1.1        R. Looek    13/11/2009    New field    Change req. R07/02
```

Figure 25.16
Derivation history

authority, such as the system architect, should assess and prioritize changes that cut across system modules that have been produced by different development teams.

In some agile methods, customers are directly involved in deciding whether a change should be implemented. When they propose a change to the system requirements, they work with the team to assess the impact of that change and then decide whether the change should take priority over the features planned for the next increment of the system. However, changes that involve software improvement are left to the discretion of the programmers working on the system. Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.

As the development team changes software components, they should maintain a record of the changes made to each component. This is sometimes called the derivation history of a component. A good way to keep the derivation history is in a standardized comment at the beginning of the component source code (Figure 25.16). This comment should reference the change request that triggered the software change. These comments can be processed by scripts that scan all components for the derivation histories and then generate component change reports. For documents, records of changes incorporated in each version are usually maintained in a separate page at the front of the document. I discuss this in the web chapter on documentation (Chapter 30).

25.4 Release management

A system release is a version of a software system that is distributed to customers. For mass-market software, it is usually possible to identify two types of release: major releases, which deliver significant new functionality, and minor releases, which repair bugs and fix customer problems that have been reported. For example, this book is being written on an Apple Mac computer where the operating system is OS 10.9.2. This means minor release 2 of major release 9 of OS 10. Major releases are very important economically to the software vendor, as customers usually have to pay for them. Minor releases are usually distributed free of charge.

A software product release is not just the executable code of the system. The release may also include:

- configuration files defining how the release should be configured for particular installations;
- data files, such as files of error messages in different languages, that are needed for successful system operation;
- an installation program that is used to help install the system on target hardware;
- electronic and paper documentation describing the system;
- packaging and associated publicity that have been designed for that release.

Preparing and distributing a system release for mass-market products is an expensive process. In addition to the technical work involved in creating a release distribution, advertising and publicity material have to be prepared. Marketing strategies may have to be designed to convince customers to buy the new release of the system. Careful thought must be given to release timing. If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it. If system releases are infrequent, market share may be lost as customers move to alternative systems.

The various technical and organizational factors that you should take into account when deciding on when to release a new version of a software product are shown in Figure 25.17.

Release creation is the process of creating the collection of files and documentation that include all components of the system release. This process involves several steps:

1. The executable code of the programs and all associated data files must be identified in the version control system and tagged with the release identifier.
2. Configuration descriptions may have to be written for different hardware and operating systems.
3. Updated instructions may have to be written for customers who need to configure their own systems.
4. Scripts for the installation program may have to be written.
5. Web pages have to be created describing the release, with links to system documentation.
6. Finally, when all information is available, an executable master image of the software must be prepared and handed over for distribution to customers or sales outlets.

For custom software or software product lines, the complexity of the system release management process depends on the number of system customers. Special releases of the system may have to be produced for each customer. Individual customers may be running several different releases of the system at the same time on different hardware. Where the software is part of a complex system of systems, several

Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date. For marketing reasons, it may be necessary to include new features in a system so that users can be persuaded to upgrade from a previous release.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Technical quality of the system	If serious system faults are reported that affect the way in which many customers use the system, it may be necessary to correct them in a new system release. Minor system faults may be repaired by issuing patches, distributed over the Internet, which can be applied to the current release of the system.

Figure 25.17 Factors influencing system release planning

different variants of the individual systems may have to be created. For example, in specialized fire-fighting vehicles, each type of vehicle may have its own version of a software system that is adapted to the equipment in that vehicle.

A software company may have to manage tens or even hundreds of different releases of their software. Their configuration management systems and processes have to be designed to provide information about which customers have which releases of the system and the relationship between releases and system versions. In the event of a problem with a delivered system, you have to be able to recover all of the component versions used in that specific system.

Therefore, when a system release is produced, it must be documented to ensure that it can be re-created exactly in the future. This is particularly important for customized, long-lifetime embedded systems, such as military systems and those that control complex machines. These systems may have a long lifetime—30 years in some cases. Customers may use a single release of these systems for many years and may require specific changes to that release long after it has been superseded.

To document a release, you have to record the specific versions of the source code components that were used to create the executable code. You must keep copies of the source code files, corresponding executables, and all data and configuration files. It may be necessary to keep copies of older operating systems and other support software because they may still be in operational use. Fortunately, this no longer means that old hardware always has to be maintained. The older operating systems can run in a virtual machine.

You should also record the versions of the operating system, libraries, compilers, and other tools used to build the software. These tools may be required in order to build exactly the same system at some later date. Accordingly, you may have to store copies of the platform software and the tools used to create the system in the version control system, along with the source code of the target system.

When planning the installation of new system releases, you cannot assume that customers will always install new system releases. Some system users may be happy with

an existing system and may not consider it worthwhile to absorb the cost of changing to a new release. New releases of the system cannot, therefore, rely on the installation of previous releases. To illustrate this problem, consider the following scenario:

1. Release 1 of a system is distributed and put into use.
2. Release 2 requires the installation of new data files, but some customers do not need the facilities of release 2 and so remain with release 1.
3. Release 3 requires the data files installed in release 2 and has no new data files of its own.

The software distributor cannot assume that the files required for release 3 have already been installed in all sites. Some sites may go directly from release 1 to release 3, skipping release 2. Some sites may have modified the data files associated with release 2 to reflect local circumstances. Therefore, the data files must be distributed and installed with release 3 of the system.

One benefit of delivering software as a service (SaaS) is that it avoids all of these problems. It simplifies both release management and system installation for customers. The software developer is responsible for replacing the existing release of a system with a new release, which is made available to all customers at the same time. However, this approach requires that all servers running the services be updated at the same time. To support server updates, specialized distribution management tools such as Puppet (Loope 2011) have been developed for "pushing" new software to servers.

KEY POINTS

- Configuration management is the management of an evolving software system. When maintaining a system, a CM team is put in place to ensure that changes are incorporated into the system in a controlled way and that records are maintained with details of the changes that have been implemented.
- The main configuration management processes are concerned with version control, system building, change management, and release management. Software tools are available to support all of these processes.
- Version control involves keeping track of the different versions of software components that are created as changes are made to them.
- System building is the process of assembling system components into an executable program to run on a target computer system.
- Software should be frequently rebuilt and tested immediately after a new version has been built. This makes it easier to detect bugs and problems that have been introduced since the last build.
- Change management involves assessing proposals for changes from system customers and other stakeholders and deciding if it is cost-effective to implement these changes in a new release of a system.

- System releases include executable code, data files, configuration files, and documentation. Release management involves making decisions on system release dates, preparing all information for distribution and documenting each system release.

FURTHER READING

Software Configuration Management Patterns: Effective Teamwork, Practical Integration. A relatively short, easy-to-read book that gives good practical advice on configuration management practice, especially for agile methods of development. (S. P. Berczuk with B. Appleton, Addison-Wesley, 2003).

"Agile Configuration Management for Large Organizations." This web article describes configuration management practices that can be used in agile development processes, with a particular emphasis on how these can scale to large projects and companies. (P. Schuh, 2007). http://www.ibm.com/developerworks/rational/library/mar07/schuh/index.html

Configuration Management Best Practices This is a nicely written book that presents a broader view of configuration management than I have discussed here, including hardware configuration management. It's geared to large systems projects and does not really cover agile development issues. (Bob Aiello and Leslie Sachs, Addison-Wesley, 2011).

"A Behind the Scenes Look at Facebook Release Engineering." This is an interesting article that covers the problems of releasing new versions of large systems in the cloud, something that I haven't discussed in this chapter. The challenge here is to make sure that all of the servers are updated at the same time so that users don't see different versions of the system. (P. Ryan, arstechnica.com, 2012). http://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/

"Git SVn Comparison." This wiki compares the Git and Subversion version control systems. (2013, https://git.wiki.kernel.org/index.php/GitSvnComparsion).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-management/

EXERCISES

25.1. Suggest five possible problems that could arise if a company does not develop effective configuration management policies and processes.

25.2. In version management, what do codeline and baseline terminologies stand for? List the features included in a version control system.

25.3. Imagine a situation where two developers are simultaneously modifying three different software components. What difficulties might arise when they try to merge the changes they have made?

25.4. Software is now often developed by distributed teams, with team members working at different locations and in different time zones. Suggest features in a version control system that could be included to support distributed software development.

25.5. Describe the difficulties that may arise when building a system from its components. What particular problems might occur when a system is built on a host computer for some target machine?

25.6. With reference to system building, explain why you may sometimes have to maintain obsolete computers on which large software systems were developed.

25.7. A common problem with system building occurs when physical filenames are incorporated in system code and the file structure implied in these names differs from that of the target machine. Write a set of programmer's guidelines that helps avoid this and any other system-building problems that you can think of.

25.8. What are the factors that influence the decision on whether or not a change should be implemented?

25.9. Describe six essential features that should be included in a tool to support change management processes.

25.10. Explain why preparing and distributing a system release for mass-market products is an expensive process.

REFERENCES

Aiello, B., and L. Sachs. 2011. *Configuration Management Best Practices*. Boston: Addison-Wesley.

Bamford, R., and W. J. Deibler. 2003. "ISO 9001:2000 for Software and Systems Providers: An Engineering Approach." Boca Raton, FL: CRC Press.

Chrissis, M. B., M. Konrad, and S. Shrum. 2011. *CMMI for Development: Guidelines for Process Integration and Product Improvement, 3rd ed.* Boston: Addison-Wesley.

IEEE. 2012. "IEEE Standard for Configuration Management in Systems and Software Engineering" (IEEE Std 828-2012)." doi:10.1109/IEEESTD.2012.6170935.

Loeliger, J., and M. McCullough. 2012. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. Sebastopol, CA: O'Reilly and Associates.

Loope, J. 2011. *Managing Infrastructure with Puppet*. Sebastopol, CA: O'Reilly and Associates.

Pilato, C., B. Collins-Sussman, and B. Fitzpatrick. 2008. *Version Control with Subversion*. Sebastopol, CA: O'Reilly and Associates.

Smart, J. F. 2011. *Jenkins: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates.

This page intentionally left blank

Glossary

abstract data type

A type that is defined by its operations rather than its representation. The representation is private and may only be accessed by the defined operations.

acceptance testing

Customer tests of a system to decide if it is adequate to meet their needs and so should be accepted from a supplier.

activity chart

A chart used by project managers to show the dependencies between tasks that have to be completed. The chart shows the tasks, the time expected to complete these tasks and the task dependencies. The critical path is the longest path (in terms of the time required to complete the tasks) through the activity chart. The critical path defines the minimum time required to complete the project. Sometimes called a PERT chart.

Ada

A programming language that was developed for the US Department of Defense in the 1980s as a standard language for developing military software. It is based on programming language research from the 1970s and includes constructs such as abstract data types and support for concurrency. It is still used for large, complex military and aerospace systems.

agile manifesto

A set of principles encapsulating the ideas underlying agile methods of software development.

agile methods

Methods of software development that are geared to rapid software delivery. The software is developed and delivered in increments, and process documentation and

bureaucracy are minimized. The focus of development is on the code itself, rather than supporting documents.

algorithmic cost modeling

An approach to software cost estimation where a formula is used to estimate the project cost. The parameters in the formula are attributes of the project and the software itself.

application family

A set of software application programs that have a common architecture and generic functionality. These can be tailored to the needs of specific customers by modifying components and program parameters.

application framework

A set of reusable concrete and abstract classes that implement features common to many applications in a domain (e.g. user interfaces). The classes in the application framework are specialized and instantiated to create an application.

application program interface (API)

An interface, generally specified as a set of operations, that allows access to an application program's functionality. This means that this functionality can be called on directly by other programs and not just accessed through the user interface.

architectural pattern (style)

An abstract description of a software architecture that has been tried and tested in a number of different software systems. The pattern description includes information about where it is appropriate to use the pattern and the organization of the components of the architecture.

architectural view

A description of a software architecture from a particular perspective.

availability

The readiness of a system to deliver services when requested. Availability is usually expressed as a decimal number, so an availability of 0.999 means that the system can deliver services for 999 out of 1000 time units.

B

A formal method of software development that is based on implementing a system by systematic transformation of a formal system specification.

bar chart (Gantt chart)

A chart used by project managers to show the project tasks, the schedule associated with these tasks and the people who will work on them. It shows the tasks' start and end dates and the staff allocations against a timeline.

black-box testing

An approach to testing where the testers have no access to the source code of a system or its components. The tests are derived from the system specification.

BPMN

Business Process Modeling Notation. A notation for defining workflows that describe business processes and service composition.

brownfield software development

The development of software for an environment where there are several existing systems that the software being developed must integrate with.

C

A programming language that was originally developed to implement the Unix system. C is a relatively low-level system implementation language that allows access to the system hardware and which can be compiled to efficient code. It is widely used for low-level systems programming and embedded systems development.

C++

An object-oriented programming language that is a superset of C.

C#

An object-oriented programming language, developed by Microsoft, that has much in common with C++, but which includes features that allow more compile-time type checking.

Capability Maturity Model (CMM)

The Software Engineering Institute's Capability Maturity Model, which is used to assess the level of software development maturity in an organization. It has now been superseded by CMMI, but is still widely used.

Computer-Aided Software Engineering (CASE)

The term that was invented in the 1980s to describe process of developing software using automated tool support. Virtually all software development is now reliant on tool support so the term 'CASE is no longer widely used.

CASE tool

A software tool, such as a design editor or a program debugger, used to support an activity in the software development process.

CASE workbench

An integrated set of CASE tools that work together to support a major process activity such as software design or configuration management. Now often called a programming environment.

change management

A process to record, check, analyze, estimate and implement proposed changes to a software system.

class diagram

A UML diagram types that shows the object classes in a system and their relationships.

client–server architecture

An architectural model for distributed systems where the system functionality is offered as a set of services provided by a server. These are accessed by client computers that make use of the services. Variants of this approach, such as three-tier client–server architectures, use multiple servers.

cloud computing

The provision of computing and/or application services over the Internet using a 'cloud' of servers from an external provider. The 'cloud' is implemented using a large number of commodity computers and virtualization technology to make effective use of these systems.

CMMI

An integrated approach to process capability maturity modeling based on the adoption of good software engineering practice and integrated quality management. It supports discrete and continuous maturity modeling and integrates systems and software engineering process maturity models. Developed from the original Capability Maturity Model.

COCOMO II

See Constructive Cost Modeling.

code of ethics and professional practice

A set of guidelines that set out expected ethical and professional behavior for software engineers. This was defined by the major US professional societies (the ACM and the IEEE) and defines ethical behavior under eight headings: public, client and employer, product, judgment, management, colleagues, profession and self.

Common Request Broker Architecture (CORBA)

A set of standards proposed by the Object Management Group (OMG) that defines distributed component models and communications. Influential in the development of distributed systems but no longer widely used.

component

A deployable, independent unit of software that is completely defined and accessed through a set of interfaces.

component model

A set of standards for component implementation, documentation and deployment. These cover the specific interfaces that may be provided by a component, component naming, component interoperation and component composition. Component models provide the basis for middleware to support executing components.

component-based software engineering (CBSE)

The development of software by composing independent, deployable software components that are consistent with a component model.

conceptual design

The development of a high-level vision of a complex system and a description of its essential capabilities. Designed to be understood by people who are not systems engineers.

configurable application system

An application system product, developed by a system vendor, that offers functionality that may be configured for use in different companies and environments.

configuration item

A machine-readable unit, such as a document or a source code file, that is subject to change and where the change has to be controlled by a configuration management system.

configuration management

The process of managing the changes to an evolving software product. Configuration management involves version management, system building, change management and release management.

Constructive Cost Modeling (COCOMO)

A family of algorithmic cost estimation models. COCOMO was first proposed in the early-1980s and has been modified and updated since then to reflect new technology and changing software engineering practice. COCOMO II is its latest instantiation and is a freely available algorithmic cost estimation model that is supported by open source software tools.

CORBA

See Common Request Broker Architecture.

control metric

A software metric that allows managers to make planning decisions based on information about the software process or the software product that is being developed. Most control metrics are process metrics.

critical system

A computer system whose failure can result in significant economic, human or environmental losses.

COTS system

A Commercial Off-the-Shelf system. The term COTS is now mostly used in military systems. See configurable application system.

CVS

A widely used, open-source software tool used for version management.

data processing system

A system that aims to process large amounts of structured data. These systems usually process the data in batches and follow an input-process-output model. Examples of data processing systems are billing and invoicing systems, and payment systems.

denial of service attack

An attack on a web-based software system that attempts to overload the system so that it cannot provide its normal service to users.

dependability

The dependability of a system is an aggregate property that takes into account the system's safety, reliability, availability, security, resilience and other attributes. The dependability of a system reflects the extent to which it can be trusted by its users.

dependability requirement

A system requirement that is included to help achieve the required dependability for a system. Non-functional dependability requirements specify dependability attribute values; functional dependability requirements are functional requirements that specify how to avoid, detect, tolerate or recover from system faults and failures.

dependability case

A structured document that is used to back up claims made by a system developer about the dependability of a system. Specific types of dependability case are safety cases and security cases.

design pattern

A well-tried solution to a common problem that captures experience and good practice in a form that can be reused. It is an abstract representation than can be instantiated in a number of ways.

digital learning environment

An integrated set of software tools, educational applications and content that is geared to support learning.

distributed system

A software system where the software sub-systems or components execute on different processors.

domain

A specific problem or business area where software systems are used. Examples of domains include real-time control, business data processing and telecommunications switching.

domain model

A definition of domain abstractions, such as policies, procedures, objects, relationships and events. It serves as a base of knowledge about some problem area.

DSDM

Dynamic System Development Method. Claimed to be one of the first agile development methods.

embedded system

A software system that is embedded in a hardware device e.g. the software system in a cell phone. Embedded systems are usually real-time systems and so have to respond in a timely way to events occurring in their environment.

emergent property

A property that only becomes apparent once all of the components of the system have been integrated to create the system.

Enterprise Java Beans (EJB)

A Java-based component model.

enterprise resource planning (ERP) system

A large-scale software system that includes a range of capabilities to support the operation of business enterprises and which provides a means of sharing information across these capabilities. For example, an ERP system may include support for supply chain management, manufacturing and distribution. ERP systems are configured to the requirements of each company using the system.

ethnography

An observational technique that may be used in requirements elicitation and analysis. The ethnographer immerses him or herself in the users' environment and observes their day-to-day work habits. Requirements for software support can be inferred from these observations.

event-based systems

Systems where the control of operation is determined by events that are generated in the system's environment. Most real-time systems are event-based systems.

extreme programming (XP)

A widely-used agile method of software development that includes practices such as scenario-based requirements, test-first development and pair programming.

fault avoidance

Developing software in such a way that faults are not introduced into that software.

fault detection

The use of processes and run-time checking to detect and remove faults in a program before these result in a system failure.

fault tolerance

The ability of a system to continue in execution even after faults have occurred.

fault-tolerant architectures

System architectures that are designed to allow recovery from software faults. These are based on redundant and diverse software components.

formal methods

Methods of software development where the software is modeled using formal mathematical constructs such as predicates and sets. Formal transformation converts this model to code. Mostly used in the specification and development of critical systems.

Gantt chart

See bar chart.

Git

A distributed version management and system building tool where developers take complete copies of the project repository to allow concurrent working.

GitHub

A server that maintains a large number of Git repositories. Repositories may be private or public. The repositories for many open-source projects are maintained on GitHub.

hazard

A condition or state in a system that has the potential to cause or contribute to an accident.

host-target development

A mode of software development where the software is developed on a separate computer from where it is executed. The normal approach to development for embedded and mobile systems.

iLearn system

A digital learning environment to support learning in schools. Used as a case study in this book.

incremental development

An approach to software development where the software is delivered and deployed in increments.

information hiding

Using programming language constructs to conceal the representation of data structures and to control external access to these structures.

inspection

See program inspection.

insulin pump

A software-controlled medical device that can deliver controlled doses of insulin to people suffering from diabetes. Used as a case study in this book.

integrated application system

An application system that is created by integrating two or more configurable application systems or legacy systems.

interface

A specification of the attributes and operations associated with a software component. The interface is used as the means of accessing the component's functionality.

ISO 9000/9001

A set of standards for quality management processes that is defined by the International Standards Organization (ISO). ISO 9001 is the ISO standard that is most applicable to software development. These may be used to certify the quality management processes in an organization.

iterative development

An approach to software development where the processes of specification, design, programming and testing are interleaved.

J2EE

Java 2 Platform Enterprise Edition. A complex middleware system that supports the development of component-based web applications in Java. It includes a component model for Java components, APIs, services, etc.

Java

A widely used object-oriented programming language that was designed by Sun (now Oracle) with the aim of platform independence.

language processing system

A system that translates one language into another. For example, a compiler is a language-processing system that translates program source code to object code.

legacy system

A socio-technical system that is useful or essential to an organization but which has been developed using obsolete technology or methods. Because legacy systems often perform critical business functions, they have to be maintained.

Lehman's Laws

A set of hypotheses about the factors that influence the evolution of complex software systems.

maintenance

The process of making changes to a system after it has been put into operation.

mean time to failure (MTTF)

The average time between observed system failures. Used in reliability specification.

Mentcare system

Mental Health Care Patient Management System. This is a system used to record information about consultations and treatments prescribed for people suffering from mental health problems. Used as a case study in this book.

middleware

The infrastructure software in a distributed system. It helps manage interactions between the distributed entities in the system and the system databases. Examples of middleware are an object request broker and a transaction management system.

misuse case

A description of a possible attack on a system that is associated with a system use case.

model-driven architecture (MDA)

An approach to software development based on the construction of a set of system models, which can be automatically or semi-automatically processed to generate an executable system.

model checking

A method of static verification where a state model of a system is exhaustively analyzed in an attempt to discover unreachable states.

model-driven development (MDD)

An approach to software engineering centered around system models that are expressed in the UML, rather than programming language code. This extends MDA to consider activities other than development such as requirements engineering and testing.

multi-tenant databases

Databases where information from several different organizations is stored in the same database. Used in the implementation of software as a service.

mutual exclusion

A mechanism to ensure that a concurrent process maintains control of memory until updates or accesses have been completed.

.NET

A very extensive framework used to develop applications for Microsoft Windows systems. Includes a component model that defines standards for components in Windows systems and associated middleware to support component execution.

object class

An object class defines the attributes and operations of objects. Objects are created at run-time by instantiating the class definition. The object class name can be used as a type name in some object-oriented languages.

object model

A model of a software system that is structured and organized as a set of object classes and the relationships between these classes. Various different perspectives on the model may exist such as a state perspective and a sequence perspective.

object-oriented (OO) development

An approach to software development where the fundamental abstractions in the system are independent objects. The same type of abstraction is used during specification, design and development.

object constraint language (OCL)

A language that is part of the UML, used to define predicates that apply to object classes and interactions in a UML model. The use of the OCL to specify components is a fundamental part of model-driven development.

Object Management Group (OMG)

A group of companies formed to develop standards for object-oriented development. Examples of standards promoted by the OMG are CORBA, UML and MDA.

open source

An approach to software development where the source code for a system is made public and external users are encouraged to participate in the development of the system.

operational profile

A set of artificial system inputs that reflect the pattern of inputs that are processed in an operational system. Used in reliability testing.

pair programming

A development situation where programmers work in pairs, rather than individually, to develop code. A fundamental part of extreme programming.

peer-to-peer system

A distributed system where there is no distinction between clients and servers. Computers in the system can act as both clients and servers. Peer-to-peer applications include file sharing, instant messaging and cooperation support systems.

People Capability Maturity Model (P-CMM)

A process maturity model that reflects how effective an organization is at managing the skills, training and experience of the people in that organization.

plan-driven process

A software process where all of the process activities are planned before the software is developed.

planning game

An approach to project planning based on estimating the time required to implement user stories. Used in some agile methods.

predictor metric

A software metric that is used as a basis for making predictions about the characteristics of a software system, such as its reliability or maintainability.

probability of failure on demand (POFOD)

A reliability metric that is based on the likelihood of a software system failing when a demand for its services is made.

process improvement

Changing a software development process with the aim of making that process more efficient or improving the quality of its outputs. For example, if your aim is to reduce the number of defects in the delivered software, you might improve a process by adding new validation activities.

process model

An abstract representation of a process. Process models may be developed from various perspectives and can show the activities involved in a process, the artifacts used in the process, constraints that apply to the process, and the roles of the people enacting the process.

process maturity model

A model of the extent to which a process includes good practice and reflective and measurement capabilities that are geared to process improvement.

program evolution dynamics

The study of the ways in which an evolving software system changes. It is claimed that Lehman's Laws govern the dynamics of program evolution.

program generator

A program that generates another program from a high-level, abstract specification. The generator embeds knowledge that is reused in each generation activity.

program inspection

A review where a group of inspectors examine a program, line by line, with the aim of detecting program errors. A checklist of common programming errors often drives inspections.

Python

A programming language with dynamic types, which is particularly well-suited to the development of web-based systems.

quality management (QM)

The set of processes concerned with defining how software quality can be achieved and how the organization developing the software knows that the software has met the required level of quality.

quality plan

A plan that defines the quality processes and procedures that should be used. This involves selecting and instantiating standards for products and processes and defining the system quality attributes that are most important.

rapid application development (RAD)

An approach to software development aimed at rapid delivery of the software. It often involves the use of database programming and development support tools such as screen and report generators.

rate of occurrence of failure (ROCOF)

A reliability metric that is based on the number of observed failures of a system in a given time period.

Rational Unified Process (RUP)

A generic software process model that presents software development as a four-phase iterative activity, where the phases are inception, elaboration, construction and transition. Inception establishes a business case for the system, elaboration defines the architecture, construction implements the system, and transition deploys the system in the customer's environment.

real-time system

A system that has to recognize and process external events in 'real-time'. The correctness of the system does not just depend on what it does but also on how quickly it does it. Real-time systems are usually organized as a set of concurrent processes.

reductionism

An engineering approach that relies on breaking down a problem to sub-problems, solving these sub-problems independently then integrating these solutions to create the solution to the larger problem.

reengineering

The modification of a software system to make it easier to understand and change. Reengineering often involves software and data restructuring and organization, program simplification and redocumentation.

reengineering, business process

Changing a business process to meet a new organizational objective such as reduced cost and faster execution.

refactoring

Modifying a program to improve its structure and readability without changing its functionality.

reference architecture

A generic, idealized architecture that includes all the features that systems might incorporate. It is a way of informing designers about the general structure of that class of system rather than a basis for creating a specific system architecture.

release

A version of a software system that is made available to system customers.

reliability

The ability of a system to deliver services as specified. Reliability can be specified quantitatively as a probability of failure on demand or as the rate of occurrence of failure.

reliability growth modeling

The development of a model of how the reliability of a system changes (improves) as it is tested and program defects are removed.

requirement, functional

A statement of some function or feature that should be implemented in a system.

requirement, non-functional

A statement of a constraint or expected behavior that applies to a system. This constraint may refer to the emergent properties of the software that is being developed or to the development process.

requirements management

The process of managing changes to requirements to ensure that the changes made are properly analyzed and tracked through the system.

resilience

A judgement of how well a system can maintain the continuity of its critical services in the presence of disruptive events, such as equipment failure and cyberattacks.

REST

REST (Representational State Transfer) is a style of development based around simple client/server interaction which uses the HTTP protocol for communications. REST is based around the idea of an identifiable resource, which has a URI. All interaction with resources is based on HTTP POST, GET, PUT and DELETE. Widely used for implementing low overhead web services (RESTful services).

revision control systems

See version control systems.

risk

An undesirable outcome that poses a threat to the achievement of some objective. A process risk threatens the schedule or cost of a process; a product risk is a risk that may mean that some of the system requirements may not be achieved. A safety risk is a measure of the probability that a hazard will lead to an accident.

risk management

The process of identifying risks, assessing their severity, planning measures to put in place if the risks arise and monitoring the software and the software process for risks.

Ruby

A programming language with dynamic types that is particularly well-suited to web application programming.

SaaS

See software as a service.

safety

The ability of a system to operate without behavior that may injure or kill people or damage the system's environment.

safety case

A body of evidence and structured argument from that evidence that a system is safe and/or secure. Many critical systems must have associated safety cases that are assessed and approved by external regulators before the system is certified for use.

SAP

A German company that has developed a well-known and widely-used ERP system. It also refers to the name given to the ERP system itself.

scenario

A description of one typical way in which a system is used or a user carries out some activity.

scenario testing

An approach to software testing where test cases are derived from a scenario of system use.

Scrum

An agile method of development, which is based on sprints – short development, cycles. Scrum may be used as a basis for agile project management alongside other agile methods such as XP.

security

The ability of a system to protect itself against accidental or deliberate intrusion. Security includes confidentiality, integrity and availability.

SEI

Software Engineering Institute. A software engineering research and technology transfer center, founded with the aim of improving the standard of software engineering in US companies.

sequence diagram

A diagram that shows the sequence of interactions required to complete some operation. In the UML, sequence diagrams may be associated with use cases.

server

A program that provides a service to other (client) programs.

service

See web service.

socio-technical system

A system, including hardware and software components, that has defined operational processes followed by human operators and which operates within an organization. It is therefore influenced by organizational policies, procedures and structures.

software analytics

Automated analysis of static and dynamic data about software systems to discover relationships between these data. These relationships may provide insights about possible ways to improve the quality of the software.

software architecture

A model of the fundamental structure and organization of a software system.

software as a service (SaaS)

Software applications that are accessed remotely through a web browser rather than installed on local computers. Increasingly used to deliver application services to end-users.

software development life cycle

Often used as another name for the software process. Originally coined to refer to the waterfall model of the software process.

software metric

An attribute of a software system or process that can be expressed numerically and measured. Process metrics are attributes of the process such as the time taken to complete a task; product metrics are attributes of the software itself such as size or complexity.

software process

The activities and processes that are involved in developing and evolving a software system.

software product line

See application family.

spiral model

A model of a development process where the process is represented as a spiral, with each round of the spiral incorporating the different stages in the process. As you move from one round of the spiral to another, you repeat all of the stages of the process.

state diagram

A UML diagram type that shows the states of a system and the events that trigger a transition from one state to another.

static analysis

Tool-based analysis of a program's source code to discover errors and anomalies. Anomalies, such as successive assignments to a variable with no intermediate use may be indicators of programming errors.

structured method

A method of software design that defines the system models that should be developed, the rules and guidelines that should apply to these models and a process to be followed in developing the design.

Structured Query Language (SQL)

A standard language used for relational database programming.

Subversion

A widely-used, open source version control and system building tool that is available on a range of platforms.

Swiss cheese model

A model of system defenses against operator failure or cyberattack that takes vulnerabilities in these defenses into account.

system

A system is a purposeful collection of interrelated components, of different kinds, which work together to deliver a set of services to the system owner and users.

system building

The process of compiling the components or units that make up a system and linking these with other components to create an executable program. System building is normally automated so that recompilation is minimized. This automation may be built in to the language processing system (as in Java) or may involve software tools to support system building.

systems engineering

A process that is concerned with specifying a system, integrating its components and testing that the system meets its requirements. System engineering is concerned

with the whole socio-technical system—software, hardware and operational processes—not just the system software.

system of systems

A system that is created by integrating two or more existing systems.

system testing

The testing of a completed system before it is delivered to customers.

test coverage

The effectiveness of system tests in testing the code of an entire system. Some companies have standards for test coverage e.g. the system tests shall ensure that all program statements are executed at least once.

test-driven development

An approach to software development where executable tests are written before the program code. The set of tests are run automatically after every change to the program.

TOGAF

An architectural framework, supported by the Object Management Group, that is intended to support the development of enterprise architectures for systems of systems.

transaction

A unit of interaction with a computer system. Transactions are independent and atomic (they are not broken down into smaller units) and are a fundamental unit of recovery, consistency and concurrency.

transaction processing system

A system that ensures that transactions are processed in such a way so that they do not interfere with each other and so that individual transaction failure does not affect other transactions or the system's data.

Unified Modeling Language (UML)

A graphical language used in object-oriented development that includes several types of system model that provide different views of a system. The UML has become a *de facto* standard for object-oriented modeling.

unit testing

The testing of individual program units by the software developer or development team.

use case

A specification of one type of interaction with a system.

use-case diagram

A UML diagram type that is used to identify use-cases and graphically depict the users involved. It must be supplemented with additional information to completely describe use-cases.

user interface design

The process of designing the way in which system users can access system functionality, and the way that information produced by the system is displayed.

user story

A natural language description of a situation that explains how a system or systems might be used and the interactions with the systems that might take place.

validation

The process of checking that a system meets the needs and expectations of the customer.

verification

The process of checking that a system meets its specification.

version control

The process of managing changes to a software system and its components so that it is possible to know which changes have been implemented in each version of the component/system, and also to recover/recreate previous versions of the component/system.

version control (VC) systems

Software tools that have been developed to support the processes of version control. These may be based on either centralized or distributed repositories.

waterfall model

A software process model that involves discrete development stages: specification, design, implementation, testing and maintenance. In principle, one stage must be complete before progress to the next stage is possible. In practice, there is significant iteration between stages.

web service

An independent software component that can be accessed through the Internet using standard protocols. It is completely self-contained without external dependencies. XML-based standards such as SOAP (Standard Object Access Protocol), for web service information exchange, and WSDL (Web Service Definition Language), for the definition of web service interfaces, have been developed. However, the REST approach may also be used for web service implementation.

white-box testing

An approach to program testing where the tests are based on knowledge of the structure of the program and its components. Access to source code is essential for white-box testing.

wicked problem

A problem that cannot be completely specified or understood because of the complexity of the interactions between the elements that contribute to the problem.

wilderness weather system

A system to collect data about the weather conditions in remote areas. Used as a case study in this book.

workflow

A detailed definition of a business process that is intended to accomplish a certain task. The workflow is usually expressed graphically and shows the individual process activities and the information that is produced and consumed by each activity.

WSDL

An XML-based notation for defining the interface of web services.

XML

Extended Markup Language. XML is a text markup language that supports the interchange of structured data. Each data field is delimited by tags that give information about that field. XML is now very widely used and has become the basis of protocols for web services.

XP

See Extreme Programming.

Z

A model-based, formal specification language developed at the University of Oxford in England.

This page intentionally left blank

Subject Index

A

abstraction level (reuse), 213
acceptability, 22, 347–48
acceptance testing, 77, 82, 249, 250–51, 252
accidents (mishaps), 343–44, 347
ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, 29–30
acquisition (procurement), 473, 553–54, 566–70
activities (software engineering activities), 20, 23, 44, 47–48, 54–61, 142, 298, 643–44. *See also* development; evolution; specification; validation
activity charts (planning), 678–80
activity diagrams (UML), 33–34, 47, 50, 56, 141, 143–44, 163
actuators, 218, 502, 613–14, 615
Ada programming language, 359
adaptors, 469, 482–83
additive composition, 481
Adobe Creative Suite, 27
aggregation, 153
agile methods, 45, 66, 72–100
  architectural design and, 168, 175
  change and, 76, 78, 91, 131–32
  change management and, 97, 748, 750
  configuration management (CM) for, 732, 742–43, 748, 750
  critical systems and, 75, 92, 96
  continuous integration, 742–43
  custom systems and, 90, 732
  customer involvement and, 76, 77, 91, 748, 750
  development team, 85, 90, 92–93
  documentation and, 73–75, 86, 89–90, 92–93, 175
  evolution and, 90, 261
  extreme programming (XP), 73, 77–84
  incremental development and, 45, 50, 73–74, 77
  large system complexity and, 93–96
  manifesto, 75–76, 77–78
  model-driven architecture (MDA) and, 162
  organizations and, 91, 97
  pair programming, 78, 83–84
  'people, not process' and, 76, 77, 91
  plan-driven approach *v.*, 45, 74–75, 91–93, 98
  principles of, 76
  process improvement and, 66
  project management and, 84–88, 643, 647, 661
  project planning, 91–93, 670, 680–83, 696
  quality management (QM), 714–16, 727
  refactoring, 51, 80–81
  risk management and, 647
  scaling, 88–97, 98
  simplicity of, 76, 78, 91
  Scrum approach and, 73, 78, 85–88, 96
  test first development, 59, 78, 81–83
  user stories for, 681–82
  user testing, 251
agile modeling, 50
Agile Scaling Model (ASM), 95
air traffic management (ATC) systems, 554–55, 569

Airbus 340 flight control system, 321–22, 340
AJAX programming, 28, 445, 512
algorithm error, 351–52
algorithmic cost modeling, 683, 684–86
alpha testing, 249
analysis systems, 25
Android, 219
Apache web server, 219
aperiodic stimuli, 613
Apollo 13 mission resilience, 409, 411, 416
application assessment (legacy systems), 269
application data, 262
application frameworks, 442, 443–46, 460
application layer, 292
application-level protection, 393–394
application programming interfaces (APIs), 39, 595–96
application security, 374–375
application software, 262
application system, 53, 438, 453–60
  COTS systems, 453
  ERP systems, 454–457
  reuse, 438, 442, 453–60
architectural description languages (ADLs), 175
architectural design, 57, 149, 167–195, 570–71, 595, 599–606
  block diagrams for, 170
  Booch's architecture catalog and, 170
  decisions, 171–73, 192
  4+1 view model, 173–74
  levels of abstraction, 169
  maintenance and, 172–73, 178
  model-driven architecture (MDA), 159–62
  non-functional requirements for, 169, 172–73
  object-oriented systems, 201–02
  patterns, 175–84, 192
  refactoring and, 168
  security and, 172, 388, 392–395
  structural models for, 149
  system development and, 570–71
  systems of systems (SoS), 595, 599–606
  views, 173–75, 192
architectural frameworks, 600–02
architectural patterns (styles), 172
  client-server architecture, 180–82, 501, 503–06, 517
  container systems, 603–05
  data-feed systems, 602–03
  distributed component systems, 501, 506–09, 517
  distributed systems, 175–84, 192, 501–12, 517
  embedded software and, 620–26, 634
  environmental control, 620, 623–25
  layered architecture, 177–79
  master-slave architecture, 501–02
  model-view-controller (MVC), 176–77
  multi-tier client-server architecture, 501, 505–06
  observe and react, 620, 621–23
  peer-to-peer (p2p) architecture, 501, 509–12, 517
  pipe and filter architecture, 182–84
  process pipeline, 620, 625–26
  real-time software, 620–26, 634
  repository architecture, 179–80
  security and, 172, 388, 392–95
  systems of systems (SoS), 602–606, 607
  trading systems, 605–06
  two-tier client-server architecture, 501, 503–05
Architecture Development Method (ADM), 601
architectures (software architectures)
  application, 184–91, 192
  architecture in the large, 169
  architecture in the small, 169
  defined, 192
  distributed, 171, 182
  fault-tolerant, 318–25
  industrial practice *v*., 170
  pipe and filter compiler, 190–91
  reference, 191
  self-monitoring, 320–22
Ariane 5 explosion, 296, 479, 480
arithmetic error, 351
as low as reasonably practical (ALARP) risks, 347
aspect-oriented software development, 442
Assertion checking, 360
assessment
  hazards for safety requirements, 345, 346–349
  security risk, 381–82
assets, 377, 378, 413, 414–415
assurance
  safety processes, 353–56
  security testing and, 402–04
ATMs (automated teller machines), 186–87, 315–16
attacks, 377, 378–79, 389, 413, 414–15, 494–95
attributes of software, 20, 22, 40
authentication, 413, 414, 416
automated management, 423–24
automated testing, 78, 81–83, 233–34, 242, 252
automatic static analysis, 359–60

availability
security and, 374, 375, 413
system availability, 172, 288, 309–12
availability metric (AVAIL), 313–314
avoidance
error discovery and, 300–01
fault, 308
hazard, 342, 351
strategies (risk management), 650
vulnerability, 378

B

B method, 49, 300, 301, 357
banking system, Internet, 505
baselines, 734, 735, 736
batch processing systems, 25
behavioral models, 154–59, 163
beta testing, 58, 60, 249–250
bidding (projects), 669, 671–72
bindings, 527–28
blackboard model, 180
block diagrams, 170, 199
Boehm's spiral process model, 48
Booch's software architecture catalog, 170
boundaries (system models), 141–42, 163, 199, 556–57
branching, 734, 739
broadcast (listener) models, 202
Brownfield systems, 94, 256
BSD (Berkeley Standard Distribution) license, 220
Bugzilla, 216
build system, 741–42
burglar alarm system, 614, 622, 629–31
business-critical system, 287
business process layer, 292
Business Process Modeling Notation (BPMN), 544–46
business process models, 544–46
businesses
activity diagrams (UML) for processes, 143–44
interrelated 4 R's approach, 426–27
legacy system evolution, 261–68
maintenance costs, 274–76, 279
modeling workflow, 67–68
open-source software and, 221
policies (rules), 262
process maturity models, 67–68
process reengineering, 276–78
processes, 262
rapid software development and, 73–74
requirements changes, 131
resilience and, 426–27
security and, 380–382
services, 534, 541–47, 548
social change and, 24
software systems, 24, 27, 45, 68, 267–68
system construction by composition, 543–44
system values, 267–68, 280
web-based applications, 27
workflow, 542, 543, 544–46

C

C and C++ programming languages, 197, 327, 330, 359, 360, 401, 444, 619
callbacks, 445
catalog interface design, 537–538
centralized systems, version management of, 735, 737
certification (software dependability), 294, 299, 302, 354, 355–56, 474, 477, 709–10
change, 61–65. *See also* process change
agile methods and, 73–74, 78, 90–91, 97
business and social needs, 24
cost effectiveness of, 133
cultural (social), 24, 97
customers and, 748–49
effects on software engineering, 27–28
extreme programming (XP) and, 78
implementation, 134, 259–60, 280
incremental delivery, 62, 64–65
plan-driven process and, 73
problem analysis and, 133
prototyping, 62–63
rapid software development for, 73–74
requirements management for, 111, 130–34
reuse, 27–28
rework for, 61, 73

change anticipation, 61
change control board (CCB), 748–49
change management, 97, 731, 745–50, 753
  agile methods and, 97, 748, 750
  change requests, 747–50
  dependability and, 299
  development environments and, 217
  requirements and, 111, 130–34
change proposals, 90, 258–59
change request form (CRF), 747–48
change tolerance, 61
characteristic error checking, 359–60
check array bounds, 330
checking requirements, 317
checklists, 403, 713–714
checksums, 745
circular buffer, 616–17
class diagrams, 141, 149–51, 163
class identification, 202–04
Cleanroom process, 230, 332
client-server architecture, 180–82, 428, 501, 503–06, 517
client-server systems, 499–501, 517
clouds, 25, 27, 532
COBOL code, 263
COCOMO II modeling, 276, 476, 686–96
  application composition model, 688–89
  cost drivers, 692
  early design model, 689–90
  post-architectural level, 692–94
  project duration and staffing, 694–96
  reuse model, 690–92
code coverage, 243–44, 252
code inspection and review, 83, 715
Code of Ethics and Professional Practice (software engineering), 29–30
codelines, 734, 735, 736, 739
collaborative systems, 588
collective ownership, 78
COM platform, 466
Common Intermediate Language (CIL), 470–71
communication
  data management layer and, 292
  message exchange, 496–97, 526–29, 537
  stakeholder, 169
communication latency, 218
compartmentalization, 399
competence, 28
completeness, 107, 129
complexity, 18, 93–96, 274–75, 278, 584–87, 606
  governance, 586–87, 588–90, 606
  large systems, 93–96
  maintenance prediction and, 274–75
  management, 585, 586–87, 587–90, 606
  reductionism for systems, 590–93, 606
  refactoring, 278
  scaling agile methods and, 93–96
  system releases, 751–52
  systems of systems (SoS), 584–87, 606
  technical, 585, 586–87, 590
compliance to software regulation, 294–95
component-based software engineering (CBSE), 442, 464–489
  component certification, 474, 477
  component management, 474, 476
  development for reuse, 473, 474–77
  development with reuse, 473, 477–80
  middleware and, 465, 472–73
  service-oriented software *v*., 466–67
component level (reuse), 214
components (software), 52–53, 188, 190, 295, 424, 465–73, 487, 526–29
  architectural design and, 172
  communications, 172, 218, 526–29
  composition, 480–86, 487
  defined, 465, 467, 487
  deployment, 471, 472–73
  design and selection of, 57, 424, 452
  external, 330–31
  implementation, 465, 466, 471–72, 475, 487
  incompatibility, 481–83
  interfaces, 208–209, 237–239, 465, 468–69
  measurement (analysis), 722–23
  models, 470–73, 487
  open-source, 220–21
  platforms for, 466–67
  remote procedure calls (RPCs) for, 470, 471
  reuse, 52–53, 212, 214, 221, 438–439, 452, 468, 487
  services *v*., 521
  service-oriented architectures (SOA), 526–29
  testing, 59, 232, 237–239
  timeouts, 330–31
  timing errors, 238–239
components (system), procurement (acquisition) of, 567–68

composition
of components, 480–86, 487
service systems and, 541–47
computation independent model (CIM), 159–61
computer science, software engineering *v*., 20, 23
concept reuse, 439
conceptual system design, 553, 563–66, 577, 594
conceptual views, 174, 192
concurrency, 491
confidence levels (verification), 228–29
confidentiality, 28, 374, 413
configurable application systems, 442, 454–457
configuration management (CM), 213, 215–216, 222, 730–55. *See also* change management
activities of, 215–16
agile methods and, 732, 742–43, 748, 750
architectural patterns, 175
change management, 731, 745–50, 753
design implementation and, 213, 215–16, 222
problem tracking, 216
release management, 216, 731, 750–53, 754
system building, 731, 740–45, 753
system integration and, 215–16
terminology for, 734
version management (VM), 215, 216, 731, 735–40, 753
configuration, software product lines, 451–52
ConOps document standard, 563
consistency, 107, 129, 652
constants, naming of, 331
construction phase (RUP), 46
consumer/producer processes (circular buffer), 616–17
container systems, 603–05
context models, 141–44, 163, 199–200
contingency plans, 650–51
continuous integration, 78, 742–43
control
application frameworks and, 445
cybersecurity, 413–414
inversion of, 445
safety-critical systems, 341–42
security, 377, 378–79
visibility of information, 325–26
control metrics, 717
controlled systems, 319
cooperative interaction patterns, 175
coordination services, 534, 548
CORBA (Common Object Request Broker Architecture), 466, 493, 507
cost/dependability curve, 290–91
cost drivers, 692
costs. *See also* estimation techniques
change analysis and, 133
COCOMO II modeling, 686–96
dependability and, 290–91
distributed systems, 495
effort, 669
fault removal, 308–09
formal verification, 357
maintenance/development, 274–76, 279, 280
overhead, 669
project planning, 669
safety engineering and, 357, 362–63
software engineering, 20
software reuse and, 214, 439
system failure, 286
COTS (commercial-off-the-shelf) systems, 453. *See also* application system reuse
critical systems, 287. *See also* safety-critical systems
agile methods and, 75
dependable processes for, 297
documentation for, 92, 96
failure of, 287, 303
formal methods for dependability of, 302
redundancy and, 295
types of, 287, 424
verification and validation costs, 290
cultural change, 97
customer involvement (agile methods), 76, 77, 91, 748, 750
customer testing, 59
customization, 471, 732–33
cybersecurity, 376, 412–416, 432

D

damage limitation, 342, 351
data clumping, 279
data collection systems, 25, 202
data flow diagrams (DFD), 154–55
data reengineering, 277
database design, 57

data-driven modeling, 154–55
data-feed systems, 602–03
data-mining system, 508
deadlines (real-time systems), 627
debugging, 58, 216, 232, 244
decentralized systems, 510–11, 517
Decorator pattern, 212
defect testing, 58, 227–28, 232
  debugging *v*., 58, 232
  performance, 248
  release testing, 248
deltas (storage management), 740
denial-of-service attacks, 289–90, 389, 423
Department of Defense Architecture Framework (DODAF), 601
dependability (software dependability), 26, 285–305
  activities for, 298
  assurance, 353–56, 402–04
  costs of, 290–91
  critical systems, 287, 290, 297, 302
  design considerations, 287, 295
  formal methods and, 299–302, 303
  functionality *v*., 286
  properties, 288–91
  redundancy and diversity, 295–97, 303
  reliability and, 288–90, 297, 303
  safety and, 288, 299
  security and, 22, 26, 288, 376–79
  sociotechnical systems, 291–95, 303
  specification and, 300–02
  system, 268, 286–91, 303
dependable programming guidelines, 325–31
deployment
  component model, 471, 472–73
  design for, 399–400
  service implementation and, 540–41
  system development and, 570
  systems of systems (SoS), 595, 597–99
  UML diagrams, 149, 218
deployment-time configuration, 451–52
derivation history, 750
design (software design), 44, 56–58, 69, 78, 196–225. *See also* architectural design; system design
  activity model (diagram), 56
  configuration management, 212, 215–16, 222
  for deployment, 399–400
  engineering programming and, 23, 44, 58
  guidelines, 396–401, 405
  implementation and, 47, 56–58, 69, 196–225
  interface, 57, 208–09, 222
  life-cycle phase, 47
  models, 123–208, 222
  object-oriented, 198–209, 222
  open-source development, 219–21, 222
  patterns, 209–12
  for recovery, 400–01
  for resilience, 424–32
  reuse and, 57, 212, 213–15
  service interfaces, 533, 536–40
  test-case, 234–37
  UML documentation, 197, 198–209
  user interface, 62
design-time configuration, 451–52
'desk' testing, 428
development
  agile techniques, 77–84, 88, 732
  customization stages, 732–33
  configuration management (CM) phases, 732–33
  engineering design and programming, 23, 44
  evolution and, 23, 60–61, 256–57, 280
  implementation stage, 56–58
  maintenance costs, 274–76, 279
  maintenance *v*., 60–61
  pair programming, 83–84
  plan-driven process, 59–60, 570
  professional software, 19–28
  refactoring, 51, 62, 80–81
  regulators for safety, 353
  reuse and, 52–54
  reuse for (CBSE process), 473, 474–77
  reuse with (CBSE process), 473, 477–80
  safety cases, 362–63
  safety-critical systems, 352–53
  services and, 541–47, 548
  sociotechnical systems, 291–295, 303
  software dependability and, 290
  spiral model for, 256–57
  system processes, 554, 570–74
  testing, 58–60, 81–83, 230–32
development team, 85, 90, 92–93
development testing, 231–42, 252
development view, 174, 192
digital art, 566
digital learning environment (iLearn), 38–39
  application programming interface (API), 39
  architecture (diagram), 38–39

elicitation of requirements, 118–20
layered architecture of, 179
photo sharing story, 118–20
services, 38–39
Virtual Learning Environment (VLE), 38
directed systems, 588
distributed architectures, 171
distributed component systems, 501, 506–09, 517
distributed development (Scrum), 88
distributed systems (software engineering), 490–519
advantages of, 491, 517
architectural design of, 171–72, 182
architectural patterns for, 175–84, 501–12, 517
attack defense, 494–95
client-server architecture, 180–82, 501, 503–06, 517
client-server systems, 499–501, 517
CORBA (Common Object Request Broker Architecture), 493, 507
design issues, 492–96, 517
interaction models, 496–97
middleware, 498–99, 517
openness, 491, 492, 493
quality of service (QoS), 492, 495
scalability, 491, 492, 494, 514, 515–16
software as service (SaS), 512–16, 517
version management of, 735, 737–39
diversity (software diversity)
application types, 24–25
dependability and, 295–97, 303
fault-tolerant architecture, 318, 322, 323–25
redundancy and, 318, 398
reliability and, 318, 322, 323–25, 336
risk reduction and, 398
software engineering, 24–27
documentation, 19, 40, 49, 56, 73–75, 92–93, 273
agile methods and, 73–75, 86, 89–90, 92–93, 126
architectural design and, 175
certification and, 294, 299, 302
change implementation, 260
maintenance and, 92, 273
organization of, 127–28
reader requirements, 103–04
safety cases, 361–67
software requirements (SRS), 126–29, 135
standards, 129, 706
system release, 741, 752–53
TDD and, 244
user requirements, 73, 126–27
domain-specific application systems, 438, 441, 446
duplicate code, 279
dynamic metrics, 720–21
dynamic model, 199, 205, 206, 222
dynamic perspective (RUP), 46
dynamic systems development method (DSDM), 73

E

e-commerce systems, 188–89
early design model, 689–90
Eclipse environment, 32, 216, 218, 219
efficiency, 22, 422–23
effort cost, 669
effort distribution, 272
egoless programming, 83
elaboration phase (RUP), 46
elicit stakeholder requirements, 450
elicitation/analysis for requirements, 55, 112–20, 134
embedded software systems, 25, 32, 634. *See also* real-time systems
architectural patterns and, 620–26, 634
design of, 217–18, 613–20
host-target development and, 217
real-time software engineering, 218, 610–37
simulators for, 217
stimulus/response model, 613–14, 634
timing analysis, 626–31
user testing, 251
emergency call log, 422–23
emergency repair process, 260–61
emergent properties, 558, 559–61, 577
encryption, 413
enduring requirements, 132
engineering, *see* software engineering; systems engineering
Enterprise Java Beans (EJB), 446, 466, 470, 507
Enterprise systems, 422, 552. *See also* ERP systems
entertainment systems, 25
environment assessment (legacy systems), 269
environmental adaptation, 271
environmental control pattern, 620, 623–25
environmental specialization (software product lines), 450

environments. *See also* IDEs
architectural patterns and, 176
business requirements changes, 131
context model for, 142–43
marketing, 229
software interaction and system failure, 293–94
work, 663
equipment layer, 292
equity trading system, 394–95
equivalence partitioning, 235–236
ERP (Enterprise Resource Planning) systems, 21, 184, 438, 442, 454–457
application frameworks, 446
architecture of, 455–456
configurable application reuse, 454–457
customer adaptation of, 438
system procurement and adaptation, 569
error-prone constructs, 308, 328–29
error tolerance, 289
errors
algorithmic, 351–52
arithmetic, 351
avoidance and discovery, 300–01
checking, 359–61
correction, 48
failure and fault *v*., 308
human, 307, 351–52, 418–21
safety engineering and, 359–61
specification, 324–25
static analysis for, 359–61
system, 307–09
timing, 238–39
estimation techniques (project planning), 682–86, 696
algorithmic cost modeling, 683, 684–86
COCOMO II model, 686–96
experience-based techniques, 683–84
software productivity and, 686
ethical/professional responsibility, 28–31, 40
ethnography technique, 116–18
evaluation, prototype phase of, 63
event-driven modeling, 156–58
evolution (software evolution), 69, 255–82
activity model (diagram), 61
agile technique and, 261
business costs and, 274–76, 279
development *v*., 60, 256–57, 280
engineer activities for, 20, 23, 44
legacy systems, 261–70, 280
life cycle, 257–58, 266
maintenance, 22, 60–61, 270–79, 280
processes, 258–61
program evolution dynamics, 271
refactoring and, 61, 78, 273, 278–79, 280
requirements changes, 131
servicing *v*., 257–58
software lifetime and, 256–57
software reengineering, 273, 276–78
spiral model of, 256–57
system evolution *v*., 575–76
exceptions
CBSE for reuse, 476–77
handlers for, 327–28
Executable UML (xUML), 162
execution time (real-time systems), 627
experience-based estimation, 683–84
experience-based testing, 403
explicitly defined process, 297
exposure, 377, 378, 379
external components, 330–31
external requirements, 109
extreme programming (XP), 73, 77–84
acceptance testing and, 77, 82
agile methods and, 73, 77–79
continuous integration and, 78, 96
pair programming, 78, 83–84
release cycle in, 77
story cards, 79–80
test first development, 78, 81–83, 242
user requirements, 73, 99

F

façade pattern, 211
failure propagation, 560–61
failures, *see also* system failure
definition *v*. judgment, 310
error and fault *v*., 308
hardware, 287
human errors and, 307, 351–52, 418–21
information loss, 286
operational, 287
safe state solutions to, 351–52
server safety *v*. privacy, 36

software, 18, 22, 26, 287, 308, 310, 340–41, 351–52
system failure costs, 286
fault (system faults), 307–09
avoidance, 308
costs of removal, 308–09
detection and correction, 308
error and failure *v*., 308
repair, 271
tolerance, 308
fault-tolerant architectures, 318–25, 491
distributed systems, 491
diversity of software, 323–25
N-version programming, 322–23
protection systems, 319–20
self-monitoring, 320–22
fault tree analysis, 349–51
feasibility studies, 54, 104
Federal Aviation Administration, 92, 290
federated systems, 589
film library, client-server architecture for, 182
firewalls, 413–14
flight control software, 296, 321–22, 340, 341
floating-point numbers, 329
formal (mathematical) models, 139
formal methods (software development), 49, 139, 299–302, 303, 356–58
B method, 49
dependability and, 299–302, 303
error avoidance and discovery from, 300–01
mathematical approach, 300, 301
model-checking, 300, 358–59
safety engineering, 356–59
security testing, 404
system models and, 139, 299–301
verification and, 300, 356–58
formal specifications, 109, 300–02
Fortify tool, 404
4 Rs model, 410–11, 414–15, 432
4+1 view model, 173–74
frameworks, 443–46, 600–02, 708–10
Free Software Foundation, 219
frequency (real-time systems), 627
fuel delivery system, 618–19
functional requirements, 105–07, 134, 312, 317–18, 335, 344
functional specialization (software product lines), 450
functionality, 286

G

'Gang of Four,' 209–12
General Public License (GPL), 220
generalization of structural models, 152–53, 205
generator-based reuse, 443
Git system, 216, 737, 740
GitHub, 476, 478
'glue code,' 466, 481, 487
GNU build system, 216
GNU General Public License (GPL), 220
Google Apps, 27
Google Code, 478
governance complexity, SoS, 586–87, 588–90, 606
graphical models, 140
graphical notations, 121
groups, *see* teamwork
growth modeling, 334
guideline-based testing, 234
guidelines
hiring, 661
dependable programming, 325–31
system security, 401–02, 405

H

handlers, exceptions, 327–28
hardware (system), 262
hardware failure, 287, 560–61
hazard-driven approaches, 342, 349–51, 368
hazards, 342, 343, 345–51
analysis of, 345, 349–51
assessment, 345, 346–49
avoidance, 342, 351
damage limitation, 342, 351
detection and removal, 342, 351
fault tree analysis, 349–51
identification of, 345–46
probability, 343
safety-critical system development, 342, 368
severity, 343
heterogeneity, software development and, 24
hierarchical composition, 480
hierarchical groups, 661–62

high-availability systems, 172, 218
honesty (people management), 653
host-target development, 213, 216–18, 222
HTML5 programming, 28, 445
http and https protocols, 530–31
human error, 307, 351–52, 418–21
human needs hierarchy, 653–54

I

IDEs (Interactive Development Environments), 53, 217
  ECLIPSE environment and, 218
  general-purpose, 218
  host-target development and, 216, 217–18, 222
  repository architecture for, 180
iLearn, 38–39, 567. *See also* digital learning environment
implementation (system implementation), 28, 47, 56–58, 69, 196–225
  components, 465, 466, 471–72, 475, 487
  configuration management, 212, 215–16
  design and, 56–58, 69, 196–225
  interface specification, 208–09
  life-cycle phase, 47
  host-target development, 213, 216–18
  open-source development, 219–21
  reuse and, 212, 213–215
  service deployment and, 540–41
  service-oriented software for, 28
  UML documentation, 197, 198–209
  unit testing and, 47
in-car information system, 522–24
inception phase (RUP), 46
inclusion (people management), 653, 657
incompatibility, component composition and, 481–83
incremental delivery, 46, 51, 62, 64–65, 76, 91
incremental development, 46, 50–51, 73–74, 77
incremental testing, 59, 242
incremental integration, 242
incremental planning, 78
information loss, 286
information systems, 32, 185–86, 187–89, 522–24
infrastructure security, 374, 375–76
inheritance, 152, 204, 209, 233, 722. *See also* generalization
input/output mapping, 310–11
inputs, validity checks of, 326–27, 399
inspections, 229–30, 239, 710–714. *See also* reviews
insulin pump control system, 32–34
  activity model of, 33, 155
  data-flow model (DMD) for, 155
  dependability properties for, 288–89
  failure in, 316–17
  functional reliability requirements, 317
  hardware components (diagram), 33
  hazards in, 346
  natural language specification for, 122
  non-functional reliability requirements, 316–17
  permanent software failure, 316
  risk classification for, 347–49
  risk reduction for, 351–52
  safety-critical system control, 341
  safety requirements for, 346–349, 351–52
  safe state, 351
  sequence diagrams for, 155
  software control of, 341
  software failure solutions, 351–52
  structured language specification for, 123–24
  tabular specification for, 124
  transient software failure, 316
issue-tracking systems, 746–47
integrated application systems, 442, 454
integration
  configuration and, 46, 52–54
  continuous, 78, 742–43
  system development and, 570
  system testing and, 48
  systems of systems (SoS), 595, 597–99
integrity, security and, 374, 413
intellectual property rights, 28
interacting workflows, 545–46
interaction models, 144–49, 163, 199–200, 496–97
  distributed systems, 496–97
  object-oriented design and, 199–200
  sequence diagrams, 146–49, 163
  use cases, 144–46, 163, 200
interactive applications, 25
interface design, 57, 208–09
interface misunderstanding, 238
interface misuse, 238

interfaces
application programming interfaces (APIs), 595–96
component, 208–09, 222, 237–239, 465, 468–69, 470–71
model specifications, 470–71
service design for, 533, 536–40, 596
specification, 208–09
systems of systems (SoS), 595–97
unified user interface (UI), 596–97
Internet banking system, 505
interviewing techniques, 115–16
intolerable risks, 347
inversion of control, 445
ISO 9001 standards framework, 708–10, 734
iteration planning, 680
iterative development/delivery, 65, 77, 98. *See also* agile methods
Iterator pattern, 212

J

Java programming language, 82, 152, 161, 197, 208, 218, 219, 327, 330, 359, 444
embedded systems development and, 619–20
interfaces, 208
program testing, 243
real-time systems development and, 619
Java Virtual Machine, 217
JavaMail library, 214
Jenkins, 743
JSON (Javascript Object Notation), 531
J2EE platform, 161, 466
JUnit, 59, 82, 217, 233, 243

L

language processing systems, 186, 189–91, 192
large-scale systems, 556
layered architecture, 177–79, 187–88, 192
layers
legacy systems, 262–64
sociotechnical systems, 292–93, 557–58
legacy systems, 261–70, 280, 540, 576
assessments, 269
business value of, 267–68, 280
component integration, 567
elements of, 262–63
management, 266–70
maintenance of, 263–64, 280
reengineering and, 276, 278
refactoring and, 279
replacement problems, 264–65
system evolution of, 546
wrapping, 278, 442, 540
Lehman's laws, 271
Lesser General Public License, GNU, 220
licensing, 220–21, 356
life cycles
application system reuse problems, 459–60
project planning stages, 668
software evolution, 257–58, 266
software model process, 45, 47–49
lifetimes, system evolution and, 575–76
Linux, 219, 398
logging user actions, 398
logical view, 174, 192
long methods, 279

M

maintainability, 22, 104, 169, 173, 198, 230, 266, 274, 275, 289, 494
maintenance (software maintenance), 22, 270
agile methods and, 90, 92
architectural design and, 172–73, 178
costs, 274–76, 279
development *v.*, 60–61
documentation and, 92, 273
legacy systems, 263–64
life-cycle phase, 48
prediction, 274–76
reengineering, 273, 276–78
refactoring, 278–79
software evolution and, 22, 263–64, 270–79
types of, 271, 280

management (software management), 26, 66–68, 84–88. *See also* configuration management; process improvement; project management; project planning; quality management; version management
agile methods, 84–88
automated, 423–24
CBSE process, 474, 476
coping with change, 63
planning, 132–33
process maturity method and, 66–68
real-time system processes, 632–34
requirements change, 130–34
resilience and, 421–24, 432
management complexity (SoS), 585, 586–87, 587–90, 606
manifesto, agile, 75–76, 77–78
marketing environment, 229
Mars exploration, 358
mathematical specifications, 121. *See also* formal methods
mean time to failures (MTTF), 313, 314
measurement. *See also* metrics
ambiguity in, 724–25
component analysis, 722–23
controller/predictor metrics, 717
quality management (QM) and, 716–26, 727
software analysis, 725–26, 727
software quality, 716–26, 727
mental health care system (Mentcare), 34–36
administrative reporting, 36
aggregation association in, 153
authentication procedures, 416
class diagrams for, 149–151
client-server architecture of, 428
context model of, 141–42
design risk assessment, 390–91
dose checking test case, 80
fail-secure approach, 397
functional requirements in, 106–07
generalization hierarchy and, 153
goals of, 35
individual care management, 35
key features of, 35–36
layered architecture pattern in, 179, 188
non-functional requirements in, 109–10
organization (diagram) of, 34
passwords, 400–101, 416
patient monitoring, 35
privacy and, 36
process model of involuntary detention, 143
release testing, 246, 247
requirements-based testing and, 246
resilience of, 289, 428–30
safety and, 36
safety-critical system control, 342
scenario in, 124–25
scenario testing and, 247
security of, 289, 377, 400–01
sequence diagrams for, 146–49
sociotechnical system for, 562–63
story cards and, 79–80
success criteria for, 562–63
system boundaries, 141–42
task cards and, 79–80
use case modeling and, 145–46
use cases for, 125–26
merging, 734, 739
message exchange, 496–97, 526–29, 537
message passing interfaces, 238
metrics
AVAIL, 243–314
control/predictor, 717
dynamic, 720–21
events, 717
non-functional requirements, 110
process measurement, 717–20
probability of failure on demand (POFOD), 313–14, 316
product, 720–22, 727
rate of occurrence of failures (ROCOF), 313–314
reliability, 313–14, 316
resource utilization, 717
software measurement and, 716–26, 727
static, 720–21
time, 717
Microsoft Office 360, 27
microwave oven scenario, 156–58
middleware, 217, 218, 446, 465, 472–73, 498–99
milestones (projects), 673, 674, 677–78, 696
minimization strategies (risk management), 650–51
mission-critical system, 287
MODAF, 600, 601
model checking, 300, 358–59, 368
model-driven architecture (MDA), 159–62
model-driven engineering (MDE), 158–59, 442

modeling systems, 25, 138–66
models, 45–54, 138–66. *See also* spiral models; UML (Unified modeling Language)
activity diagrams (UML) for, 33–34, 141, 143–44, 163
activity stages, 47–48, 142
agile approach and, 50, 162
algorithmic cost modeling, 683, 684–86
application architecture, 185
behavioral, 154–59, 163
class diagrams for, 149–50
COCOMO II, 276, 476, 686–96
component, 470–73, 487
context, 141–44, 163, 199–200
data-driven, 154–55
dynamic, 199, 205, 206, 222
event-driven, 156–57
formal (mathematical), 139, 300
generalization, 152–53, 205
incremental development, 46, 49–51
integration and configuration, 46, 52–54
interaction, 144–49, 163, 199–200, 496–97
ISO 9000 standards framework, 708–10, 734
object-oriented design, 199–200, 204–08
open-source licensing, 220–21
processes, 45–54, 68
project estimation, 682–96, 696
quality management (QM) and, 709–10, 719
real-time system design, 617–19
reliability growth, 334
RUP (Rational Unified Process), 46–47
reuse-based development, 52–54
sequence, 144, 146–49, 155, 163, 205, 206–07
spiral, 63, 256–57
state machine, 205, 207–08, 222, 617–18, 634
state-based, 156–158, 163
static, 205, 222
stimulus/response, 613–14, 634
structural, 149–54, 163, 199, 205
subsystem, 205–06
'Swiss cheese,' 420–21
of testing process, 230–31
UML (Unified Modeling Language), 33–34, 139, 140–41, 144–49, 713
use case, 125–26, 141, 144–46, 163, 200–01
model-view-controller (MVC) pattern, 176–77, 179, 444
monitoring projects, 651–52, 673
motivation (people management), 653–56
multi-tenancy, 514, 515, 516
multi-tier client-server architecture, 501, 505–06
MySQL, 219, 445

N

N-version programming, 322–23
namespaces, 528–29
natural language requirements, 121–22
nested technical and sociotechnical systems, 416–17
.NET framework, 161, 443, 446, 466, 470–71, 478, 507
non-deterministic properties, 561–62
non-functional requirements, 105, 107–11, 134, 169, 172–73, 312, 314–18, 547

O

object and function reuse, 438
object classes, 149–50, 202–04, 470
object constraint language (OCL), 208, 484–85
object level (reuse), 214
Object Management Group (OMG), 159
object-oriented metrics, 721–22
object-oriented systems
architectural design and, 201–02
class diagrams for, 149–50
class identification, 202–04
design, 198–209, 222
frameworks in, 444
interface specification, 208–09
system (design) models, 204–08
Unified Modeling Language (UML) and, 140, 198–209
use case model, 200–01
Objectory method, 125
observe and react pattern, 620, 621–23
Observer pattern, 210–11
on-site customer, 78
openness, distributed software, 491, 492, 493
open-source development, 219–21, 222, 738–39

operating system layer, 292
operating systems (real-time), 631–34, 635
operation and maintenance, 48
operation incompatibility, 481
operation incompleteness, 481
operation stage (systems), 554
operational failure, 287
operational processes, 421–24, 432
operational profiles, 334–35
operational security, 374, 376
operator reliability, 287, 560–61
Oracle, 21, 219
organizational design patterns, 175
organizational layers, 292, 557
organizational requirements, 108–09
organizational systems, 589
organizations and security, 380–82
overhead costs, 669
overspecification of reliability, 315

P

packing robot control system, 168
pair programming, 78, 83–84, 715
parameter definition, 452
parameter incompatibility, 481
parameter interfaces, 237
partition testing, 234–36
partner company software systems, 49
password checker, 392
passwords, 400–01, 413, 414, 416
path testing, 237
patient records system (PRS), 148–49
patterns, 175–84, 209–12, 442, 444
  application frameworks and, 444
  architectural, 175–84
  design, 209–12, 442
payment models, 547
peer-to-peer (p2p) architecture, 501, 509–12, 517
penetration testing, 403–04
People Capability Maturity Model (P-CMM), 656
people management, 652–56, 664
performance, 172, 248
periodic stimuli, 613
photo library, 483–85
physical view, 174, 192
pipe and filter architecture, 182–84, 191
plan-driven process, 45, 47, 50, 73, 570
  agile methods *v.*, 45, 74–75, 91–93, 98
  changing environment and, 73
  incremental development and, 50
  model processes, 47, 50
  project planning and, 672–75, 696
  scheduling and, 675–76
  system development and, 570
  testing (validation) phases, 59–60
  waterfall model, 47–48
planning game, 681–82
planning. *See also* project planning
  incremental, 78
  requirements management, 132–33
  risk, 650–51
  Scrum product backlog, 85, 86, 98
  test, 231
platform-independent model (PIM), 159–61
platform-level protection, 393–394
platform services, 472
platform specialization (software product lines), 450
platform-specific models (PSM), 160–61
plug-in architecture, 218
pointers, 308, 329
post-architectural level, 692–94
power supply failure, 627–28
practice perspective (RUP), 46
prediction, maintenance and, 274–76
predictor metrics, 717
PRISM model checker, 358
probability of failure on demand (POFOD), 313–14, 316
probability values, hazards, 343
problem tracking, 216
procedural interfaces, 238
process (software processes), 23, 26, 43–71
  activities, 40, 54–61
  agile approach, 45, 66
  analysis, 67, 112–20, 626–31
  assurance, 353–56
  design and implementation, 56–58
  emergency repair, 260–61
  engineer activities for, 20, 23, 44, 54–61
  evolution, 44, 60–61, 258–61
  improvement of, 65–68
  life cycles, 45, 47–49

management, 421–24, 432
maturity approach, 66–68
measurement, 66–67, 717–20
models, 45–54, 68
operational, 421–24, 432
plan-driven, 47–48
professional, 19–28, 45
prototype development, 62–63
quality (process-based), 65–68, 705
quality metrics, 717–20
review phases, 711–13
RUP (Rational Unified Process), 46–47
specification, 44, 54–56
standards, 45, 707, 708
validation, 44, 58–60
process change, 45, 69
agile manifesto and, 75–76
CBSE, 473–480
coping with, 61–65
evolution, 258–61
implementation, 259–60
for safety assurance, 353–56
software processes, 61–65, 67
urgent changes, 260
process improvement, 69
agile approach, 66
business values, 267–68
legacy system management, 266–70
process maturity approach, 66–68
reengineering, 276–78
refactoring, 278–79
software quality and, 65–68
software evolution and, 266–70, 276–79
process management, real-time systems, 632–34
process maturity approach, 66–68
process pipeline pattern, 620, 625–26
process requirements, 317
process specialization (software product lines), 450
process view, 174, 192
procurement (acquisition), 473, 553–54, 566–70, 577
producer/consumer pattern, 202
producer/consumer processes (circular buffer), 616–17
product
instance development, 450
quality metrics, 720–22, 727
requirements, 108–09
software types, 20–21, 24–26
standards, 706, 707
product architects (Scrum), 96
product backlog (Scrum), 85, 86
product owner (Scrum), 85
product risk management, 644–75, 646
professional software development, *see* development
program evolution dynamics, 271
program generators, 442
program inspections, 229–30, 239, 713–14. *See also* reviews
program libraries, 442
program modularization, 277
program structure improvement, 277
programmer/tester pairs, 231–32
programming. *See also* extreme programming
dependable guidelines, 325–31
egoless, 83
engineering design and, 23, 44, 58
real-time systems, 619–20
secure system guidelines, 401–02
techniques/activities, 26, 54–56
project management, 84–88, 641–66
activities, 643–44
agile methods and, 84–88, 643, 647, 661
differences from engineering, 642–43, 664
motivation and, 653–56
relationships with people, 652–56, 664
risk management, 644–52, 664
teamwork, 656–64
project planning, 92–93, 667–99
agile methods and, 670, 680–83, 696
bidding, 669, 671–72
COCOMO II cost modeling, 686–96
development team effectiveness, 92–93
duration and staffing, 694–96
estimation techniques, 682–86, 696
life cycle stages of, 668
milestones, 673, 674, 677–78, 696
plan-driven development and, 672–75, 696
process, 673–75
project costs, 669, 696
scaling agile methods for, 91–93
scheduling and, 675–80, 696
software pricing for, 670–72, 696
supplements, 673
user stories for, 681–82
project risk management, 644–45
Promela, 358

protection, 383
  assets, 380, 384, 390
  cybersecurity, 376, 414
  fault-tolerant architecture, 319–20
  layered architecture design, 393–95
  systems, 319–20, 414
prototyping (system prototyping), 62–63, 69, 117, 130
Python, 190, 197, 198, 327, 444

Q

quality management (QM), 299, 700–29
  agile development and, 714–16, 727
  configuration management (CM) and, 733
  documentation standards, 706
  reviews and inspections, 710–14, 727
  software development and, 701–02
  software measurement/metrics and, 716–26, 727
  software quality and, 703–05, 727
  software standards and, 706–10, 727
quality of service (QoS), 492, 495
quantitative reliability specifications, 314–15

R

range checks, 326
rapid software development, 73–74
rate of occurrence of failure (ROCOF), 313–14
reactive systems, 612
realism checks, 129
real-time systems, 205, 218, 610–37
  architectural patterns for, 620–26, 634
  design, 205, 613–20
  embedded systems, 218, 610–637
  modeling, 617–19, 634
  operating systems, 631–34, 635
  process management, 632–34
  programming, 619–20
  responsiveness, 611–12
  software engineering for, 610–37
  stimulus/response model, 613–14, 634
  timing analysis, 626–31, 635
reasonableness checks, 327
recognition, 410, 411, 414–15, 432
record-level protection, 393–394
recovery
  database integrity checking and, 430
  design for, 400–01
  requirements, 317
  resilience and, 411, 414–15, 430, 432
reductionism of complex systems, 590–93, 606
redundancy
  dependability and, 295–97, 303
  diversity and, 318, 398
  requirements, 317
reengineering (software reengineering), 273, 276–78, 280
refactoring, 51, 62, 78, 80–81, 83–84, 168, 278–79
  agile methods, 51, 80–81
  architectural design and, 168
  extreme programming (XP) methods, 78
  maintenance and, 278–79
  pair programming, 83–84
  software evolution, 273, 278–79, 280
reference architectures, 191
refinement-based development, 300
regression testing, 244
regulation and compliance (software), 294–95, 353
regulators, 294–95, 361, 362, 368
reinstatement, 411, 414–15, 432
release alignment (Scrum), 96
release management, 216, 731, 750–53, 754
release testing, 245–48
reliability, 309
  availability and, 309–12
  dependability and, 288–90, 297, 303, 336
  diversity and, 318, 322, 323–25, 336
  emergent properties, 560–61
  failure and, 18, 307–12, 560–61
  fault-tolerant architectures, 318–25
  functional requirements, 312, 317–18, 335
  growth modeling, 334
  human error, 307
  measurement of, 331–35
  metrics, 312–13, 332, 335
  non-functional requirements, 312, 314–18
  operational profiles, 334–35
  overspecification of, 315
  programming guidelines, 325–31
  requirements, 312–18, 335
  safety and, 340–41
  security and, 379

sociotechnical systems, 560–61
software, 18, 560–61
specification, 314–18
system error, 307–09
system fault, 307–09
systems, 18, 19, 22, 288–90, 297, 303, 306–38
statistical testing, 332–33, 336
remote method invocations (RMIs), 497
remote procedure calls (RPCs), 470, 471, 497
repairability, 289
repeatable process, 297, 303
replicated servers, 318
repository architectural pattern, 179–80, 190
repository cloning, 737–38
representation checks, 327
requirements, 102, 134
agile methods and, 55, 131–32
analysis and definition (life-cycle phase), 47
availability, 218
business changes, 131, 135
classification and organization of, 113
components, 218
discovery and understanding, 113, 115–18
documents (software specification), 103–04, 111, 114, 126–29, 135
elicitation and analysis of, 55, 112–20, 134
enduring, 132
engineering understanding of, 20, 23, 26
evolution, 131
functional, 105–07, 134, 317–18
hazard-based, 345
identification, 132
management, 132–134, 135
non-functional, 105, 107–11, 134, 314–17
notations for writing, 121
prioritization and negotiation of, 113
refinement, 53
reliability and, 312–18
reviews, 130
risk-based, 344, 345
safety, 344–52
specification, 55, 69, 102–03, 106–07, 110, 120–29, 135, 314–18, 344, 345
spiral model for, 572
software process, 44, 54–56
storage, 132
system, 102–03, 120–21
testing (requirements-based), 245–46
traceability, 132, 133
user, 102–03
validation, 55, 129–30, 135
volatile, 132
requirements engineering (RE), 69, 101–37
change management, 111, 130–34
documents for, 103–05
elicitation/analysis process, 112–20, 134
ethnography technique for, 116–18
feasibility studies, 54, 104
interviewing techniques for, 115–16
processes, 111–12, 134
software process activities, 44, 54–56
software documentation (SRS) for, 126–27
spiral model for, 112
system development and, 570
requirements partitioning, 571
research management systems, 448–49
resilience (system resilience), 288, 409, 408–34
activities, 410–11
automated management, 423–24
cybersecurity, 412–16, 432
dependability and, 288, 289
design for, 424–32
efficiency and, 422–23
engineering, 408–34
4 Rs model, 410–11, 414–15, 432
human error and, 418–21
interrelated business approach, 426–27
management, 421–24, 432
operational processes, 421–24, 432
security and, 288, 379
sociotechnical systems, 416–24
survivable systems analysis, 425–26
system failure and, 410–12
testing, 427–428
resistance, 410–11, 414–15, 432
resource management systems, 188–89, 192
resource sharing, 491
respect (people management), 652
restart capabilities, 329–330
restaurant interactions, 496–97
RESTful services, 524, 529–33, 544
reuse (software reuse), 26, 28, 46, 52–54, 169, 209–10, 212, 213–15, 222, 437–63, 474–480
application frameworks, 442, 443–46, 460
application system, 438, 453–60
approaches supporting, 441–43

reuse (*continued*)
architectural design and, 169
CBSE for, 473, 474–77
CBSE with, 473, 477–80
component selection and design, 57
components, 52–53, 212, 214, 221, 438–439, 452, 468, 487
costs of, 214, 439
design patterns, 209–10, 212, 442, 444
engineering applications of, 26, 28
generator-based, 443
implementation and, 212, 213–15
integration and configuration of, 52–54
integration problems, 459–60
landscape, 340–443
levels of, 213–14
object and function, 438
process model for, 52–53
software development tools, 53
software product lines, 442, 446–52
system features and, 46
reuse-based software engineering, 53–54, 438
reuse model, 690–92
reverse engineering, 277
reverse planning, 680
reviews, 130, 229, 239, 710–14
checklists, 713–14
code, 83, 715
hazard register for, 355
inspections and, 229, 710–14, 727
program inspections, 713–14
quality management (QM), 710–14, 727
requirements validation, 130
review process, 711–13
safety, 354, 355
verification and validation using, 229
rework, 49, 56, 61, 73, 75, 84, 129
risk
acceptable, 347–48
accidents (mishaps) and, 343–44, 347
analysis, 362, 648–49
as low as reasonably practical (ALARP), 347
defined, 343
redundancy and diversion for, 398
indicators, 652
intolerable, 347
ranking types of, 649
reduction, 351–52, 398
security assessment, 381–82, 405
triangle, 347–48
risk management, 644–52, 664
identification of risk, 647–48
planning process, 650–51
processes, 645–47
product risks, 644–45
project risks, 644–45
risk analysis and, 648–49
risk monitoring, 651–52
strategies for, 650–81
risk-based requirements specification, 344, 345
robot control system, 168
role replication (Scrum), 96
Ruby, 190, 444
RUP (Rational Unified Process), 46–47

S

safety, 339–72, 379
architectural design and, 172
assurance processes, 353–56
costs and, 357, 362–63
dependability and, 288, 299
engineering processes, 352–61
ethics and, 30–31
formal verification, 356–58
functional requirements, 344
hazard-driven requirements, 345, 368
hazards and, 342, 343, 345–351
model checking, 358–59, 368
regulation and compliance for, 294–95
regulators, 294–95, 361, 362
reliability and, 340–41
requirements, 344–52, 362
risks and, 343, 343–44, 347–48, 351–52
software certification, 355–56
static program analysis, 359–61, 368
terminology, 343
safety cases, 361–67, 368
development of, 362–63
organization of, 361–62
regulators for, 361, 362, 368
software safety arguments, 364–67
structured arguments, 363–364

safety-critical systems, 287, 340–44, 368
  certification of, 294, 302, 355–56
  control systems, 341–42
  dependability and, 294, 302
  development process and, 352–53
  error-prone constructs and, 329
  hazard-driven techniques, 342
  primary safety-critical software, 341
  process assurance and, 355–56
  regulation and compliance for, 294, 353
  risk triangle for, 347–48
  secondary safety-critical software, 341–42
  system failure and, 340–41
safety reviews, 355
SAP, 21
Sarbanes Oxley accounting regulations, 51
scalability, 491, 492, 494, 514, 515–16
scale, software development and, 24
scaling agile methods, 88–97, 98
scenarios
  elicitation of requirements from, 118–20
  testing, 246–47, 252
  use cases, 125–26
scheduling, 675–80, 696
  activity charts for, 678–80
  project planning and, 675–80, 696
  plan-driven projects, 675–76
  presentation (visualizing), 676–80
Scrum, 73, 78, 85–88, 96, 98
secure systems, 561
security, 24, 26, 373–407
  application, 374–375
  architectural design and, 172, 388, 392–95
  assurance, 402–04
  availability, 374, 375, 379
  checklist, 403
  confidentiality, 374
  controls, 377, 378–79
  dependability and, 22, 26, 288, 376–79
  design for, 374, 388–402, 405
  engineering, 373–407
  failure, 397
  guidelines, 396–401, 404
  infrastructure, 374, 375–76
  logging user actions, 398
  operational, 374, 376
  organizations and, 380–82
  policies, 396–97
  programming guidelines, 401–02
  protection, 380, 384, 390, 393–94, 395
  regulation and compliance for, 294–95
  reliability and, 379
  requirements, 382–88
  resilience and, 288, 379
  risk assessment, 381–82, 405
  safety and, 379
  system layers, 374–75
  terminology, 377–378
  testing, 402–04
  threats, 377, 378, 404
  trust and, 22, 24
  usability guideline, 397–98
  validation, 405
  vulnerability and, 377, 378, 391, 401
self-monitoring architecture, 320–22
SEMAT (software engineering methods and tools) initiative, 24
semicentralized P2P architecture, 511, 512
sensor-based data collection systems, 32
separation of concerns, 486
sequence diagrams, 141, 144, 146–49, 155, 163, 205, 206–07, 241
sequential composition, 480
server overload, 512–13
service engineering, 533–41
  candidate identification, 533–36
  implementation and deployment, 540–41
  interface design, 533, 536–40
  legacy systems and, 540
service information exchange (SOAP), 525–26, 531, 544
service-oriented architectures (SOAs), 513–14, 520–50
  approach, 522, 524
  components, 526–29
  message exchange, 526–29
  service interface, 528
  service protocols, 525
  software as service (SaS) *v.*, 513–14, 522
  standards, 525–26
  web applications, 524–29
  WSDL and, 526, 527–29
service-oriented software engineering, *see* service engineering; service-oriented architectures (SOAs); services
service-oriented systems, 442, 466–67, 526–33

service-to-service communication, *see* integrated services
services, 521
business, 534, 541–47, 548
classification of, 534, 548
communication and, 524–29
components, 521, 526–29
composition (construction) of, 541–47
coordination, 534, 548
incremental delivery and, 64–65
operation and maintenance for, 48
process models for, 544–46
reusable Web components, 52, 526–29
reuse of, 542
software development and, 541–47, 548
testing, 543, 546–47
utility, 534, 548
web-based, 27–28, 521
RESTful approach, 524, 529–33, 544
service information exchange (SOAP), 525–26, 531, 544
workflow, 542, 543, 544–46, 548
servicing, evolution *v*., 257–58
shared memory interfaces, 238
signatures, 744–45
simple design, 78
simplicity (agile methods), 76, 78, 91
simulation systems, 25
simulators, 217
size checks, 327
SLAM model checker, 358
small releases, 78
social change, business and, 24
social layer, 292
sociotechnical systems, 552, 577
complexity of, 556, 558–59
defensive layers, 419–20
emergent properties 544, 559–61, 577
environment and software interaction, 293–94
failure propagation, 560–61
human error and, 418–21
layers of, 292–93, 557
management, 421–24, 432
nested technical systems, 416–17
non-deterministic properties, 561–62
operational processes, 421–24, 432
organizational elements, 557–58
regulation and compliance, 294–95
resilience and, 416–24
success criteria, 562–63
systems engineering for, 556–59
software, 19, 20, 228
attributes, 20, 22
customized (bespoke), 21
efficiency, 22
engineering ethics, 28–31
failures, 18
generic products, 20–21
issues affecting, 24
lifetime, 256–57
product types, 20–21, 24–26
professional development, 19–28
regulation and compliance of, 294–95
system boundaries and characteristics, 26
software architecture catalog, Booch's, 170
software as service (SaS), 512–16, 517
configuration of, 514–15
multi-tenancy, 514, 515, 516
scalability, 514, 515–16
server overload and, 512–13
service-oriented architectures (SOAs) *v*., 513–14, 522
'software crisis', 19
Software Development Life Cycle (SDLC) model, 45
software development tools, 53
software diversity, 318, 322, 323–25, 336
software engineering, 19–23, 40, 92
activities for software process, 20, 23, 44
computer science *v*., 20, 23
diversity, 24–27
engineering discipline, 21–22
ethical responsibility and, 28–31, 40
formal verification, 356–58
fundamental notions in, 26, 40
Internet effect on, 20, 27–28
licensing for, 356
model checking, 358–59, 368
model-driven engineering (MDE), 158–59
product development and, 20–21
reuse-based, 53–54, 438
safety processes, 352–61
static program analysis, 359–61, 368
systems engineering *v*., 20, 23, 40, 554
web-based systems, 27–28
Software Engineering Institute (SEI), 67

software measurement/metrics, 716–26, 727
software platform, 57
software pricing, 670–72, 696
software product lines, 442, 446–52
software quality attributes, 704
software requirements specification (SRS), 126–29
software safety arguments, 364–67
source code translation, 277
SourceForge, 476, 478
space shuttle (U.S.) system, 319
specialization, software product lines, 450
specifications (software specifications), 20, 54–56, 208–09, 300–02
  availability, 313
  engineering definition and constraints, 23
  functional requirements, 106–07
  graphical notations, 121
  dependability and, 300–02
  design interface, 208–09
  errors, 324–25
  formal techniques, 300–02
  hazard-driven safety requirements, 345
  management of, 26
  natural language requirements, 121–22
  non-functional requirements, 110
  problem analysis and, 133
  reliability metrics, 313–14
  risk-based requirements, 344, 345
  safety requirements and, 344–45
  software process, 44, 54–56
  SRS document, 126–29
  structured natural language requirements, 121, 122–24
  system failure and, 310
  system requirements, 102–03, 120–29, 135
  use cases, 125–26
  user requirements, 102–03, 120, 135
speculative generosity, 279
SPIN model checker, 358
spiral models, 48, 112, 256–57, 572
sprint (Scrum), 85, 86–87
SQL (Structured Query Language), 218, 399, 401, 445, 505
stable domain abstractions, 475
staff allocation charts, 678, 680
stakeholders, 103–04, 107, 112–16
stand-alone applications, 25
standards
  documentation, 706
  ISO 9000 standards framework, 708–10, 734
  process, 45, 707, 708, 734
  product, 706, 707
  quality management (QM) and, 706–10, 727
  software, 706–10, 727
  service-oriented architectures (SOAs), 524, 525–26
  value of, 707–08
  web service, 525–26
state diagrams (UML), 141, 163, 205, 207–08
state machine models, 205, 207–08, 222, 617–18
state-based modeling, 156–58
static analyzers, 217
static metrics, 720–21
static models, 143, 205, 222
static perspective (RUP), 46
static program analysis, 359–61, 368
statistical testing, 332–33, 336
stimulus/response (embedded systems) model, 613–14, 634
storage management, 132, 740
stories, elicitation of requirements from, 118–20
story cards, 79–80, 99. *See also* user stories
stress testing, 248
structural models, 149–54, 163, 199, 205
structured arguments, 363–64
structured natural language requirements, 121, 122–24
subsystem engineering, 571, 573
subsystem faults, 573
subsystem model, 205–06
Subversion system, 216, 735
support environment, 32
support services, 472
support software, 262
survivable systems analysis, 425–26
sustainable pace, 78
'Swiss cheese' model, 420–21
switch (case) statements, 279
system availability, *see* availability
system boundaries, 141–42, 163, 199, 556–57
system building, 731, 740–45, 753
system construction by composition, 543–44
system design
  actuator control processes, 613–14, 615
  embedded systems, 217–18, 613–20

system design (*continued*)
host-target development, 213, 216–18, 222
modeling, 617–19
producer/consumer processes, 616–17
programming, 619–20
real-time systems, 205, 613–20
risk assessment, 389–92
security systems, 388–402, 405
stimulus response model, 613–14
system error, 307–09
system failure, 307
acceptance of, 410
availability and, 309–12
costs of, 286
critical systems, 287, 290, 297, 302, 340–41
dependability and, 22, 268, 286–91, 303
error and fault *v*., 308
hardware failure and, 287
human errors and, 287, 351–52
nondeterminism and, 560–61
reliability and, 307–12, 560–61
reparability and, 289
resilience and, 410–12, 420–21
safety-critical systems, 340–41
security and, 22, 268, 397
sociotechnical, 560–61
software failures and, 287, 340–41
specifications and, 310
'Swiss cheese' model of, 420–21
types of, 287
system fault, 307–09
system infrastructure frameworks, 446
system integration, 215–16
system level (reuse), 214
system modeling, *see* models
system of system coalitions, 589
system output, 268
system requirements, 52, 102–03
system reuse, 438
system selection, 594–95
system testing, 48, 59, 231–32, 240–42
system versions, 323–325
system vision document, 565–66
systems (software systems). *See also* distributed systems; embedded software systems; systems of systems (SoS)
activity models (diagram), 60, 61
agile methods for, 93–96
analysis for architectural design, 169
case study types, 31–32
complexity of, 18, 93–96, 274–75, 278, 552–53, 558–59
cost effectiveness of, 22–23
dependability, 268, 286–91, 303
engineering fundamentals for, 26, 40
large-scale, 93–94, 556
modeling, 25, 138–166
sociotechnical, 291–95, 303, 556–63
software design and, 47
specification requirements, 120–29
state representation, 155
systems of systems (SoS) *v*., 581–82
types of, 18, 20–21, 24–26, 32, 40, 552
systems engineering, 20, 23, 40, 551–79
conceptual design, 553, 563–66, 577
development processes, 570–74, 577
enterprise systems, 552
lifetimes and, 575–76
range of disciplines, 554–55
sociotechnical systems, 552, 556–63, 577
software engineering *v*., 20, 23, 40, 554
spiral model for requirements, 572
stages of, 553–54
system evolution, 575–76
system procurement (acquisition), 453–54, 566–70, 577
technical computer-based systems, 552
systems of systems (SoS), 25, 256, 442, 556, 580–609
architectural design, 595, 599–606, 607
classification of systems, 587–90, 606
container systems, 603–05
data-feed systems, 602–03
deployment and integration of, 595, 597–99
engineering, 593–99
governance complexity, 586–87, 588–90, 606
interface development, 595–97
large-scale systems, 556
management complexity, 585, 586–87, 587–90, 606
reductionism, 590–93, 606
software systems, 582
system complexity, 584–87, 606
system *v*., 581–82
technical complexity, 585, 586–87, 590
trading systems, 605–106

T

tabular specification, 124
task cards, 79–80, 82. *See also* user stories
teamwork, 656–64
  development team, 85, 90, 92–93
  group cohesion, 658
  group communication, 662–64
  group member selection, 659–60
  group organization, 660–62
  hierarchical groups, 661–62
  hiring people, 661
  physical work environment and, 663
technical complexity, SoS, 585, 586–87, 590
technical computer-based systems, 552
test cases, 130, 234–37, 252
test-driven development (TDD), 242–45
test-first development, 59, 78, 81–83, 252
test planning, 231
testing (software testing), 58–60, 226–54, 402–04, 427–28
  acceptance, 77, 82, 249, 250–51, 252
  agile methods for, 59, 78, 81–83, 251
  alpha, 249
  assurance and, 402–04
  automated, 78, 81–83, 233–34, 242, 252
  beta, 58, 60, 249–250
  choosing test cases, 234–37, 252
  component testing, 59, 232, 237–39
  customer, 58, 59
  debugging *v*., 58, 232, 244
  defect, 58, 227–28, 232, 245, 248
  development and, 59–60, 81–83, 570
  development testing, 231–42, 252
  goals of, 227
  incremental approach, 59
  inspections *v*., 229–30
  model of, 230–31
  penetration, 403–04
  plan-driven phases, 59–60
  process, 58–60
  release testing, 245–48
  reliability and, 332–33, 336
  resilience, 427–428
  security, 402–04
  services, 543, 546–47
  stages in, 59, 231
  statistical, 332–33, 336
  system, 59, 232, 240–42
  test-driven development (TDD), 242–45
  tool-based analysis, 404
  unit testing, 47, 232–37
  user testing, 249–51
  validation, 58–60, 227–29
threats, 377, 378, 404, 413, 414–15
timeouts, 330–31
timestamps, 744
timing analysis, 626–31, 635
timing errors, 238–39
TOGAF, 600, 601
tool-based analysis, 404
tool support, 132, 743, 744, 746
traceability (requirements), 132, 133
trading systems, 605–06
transaction-based applications, 25
transaction processing systems, 185, 186–87, 192
transition phase (RUP), 46–47
triple modular redundancy (TMR), 322
trust, security and, 22, 24
two-tier client-server architecture, 501, 503–05

U

UML (Unified Modeling Language), 140
  activity diagrams, 33–34, 141, 143–44
  architectural design and, 139, 175, 205
  behavioral models, 155–57
  business processes and, 143–44
  class diagrams, 141, 149–50
  component interface diagram, 469
  deployment diagrams, 149, 218
  diagram types, 139, 140–41, 205
  event-driven, 156–57
  executable (xUML), 162
  generalization and, 152
  interaction models, 144–49
  object oriented metrics and, 721
  object-oriented systems and, 140, 198–209
  package symbol, 37
  sequence diagrams, 141, 146–49, 155, 163, 205, 206–07
  state diagrams, 141, 205, 207–08

UML (*continued*)
subsystem models, 205–06
system modeling using, 139, 140
use cases, 125–26, 141, 144–46, 163, 205
workflow models, 143–44, 544
unified user interface (UI), 596–97
Uniform Resource Locator (URL), 530–32, 539
unit testing, 47, 231, 232–37
Universal Description, Discovery, and Integration (UDDI), 526
Universal Resource Identifiers (URIs), 471, 527
Unix systems, 183, 401
urgent changes, 260
usability
error tolerance, 289
patterns, 175
requirements, 109–10
security guideline, 397–98
usage, component models and, 471
use cases, 125–26, 141, 144–46
interaction models, 144–46, 163, 200–01
requirements specification and, 125–26
testing, 240–41
UML diagram models, 141
user access, 392
user actions, logging, 398
user-defined error checking, 360
user expectations, 228–29
user interface design, 62
user requirements, 55, 73–74, 102–03
user stories, 79–80, 82, 86, 247, 681–82
conceptual design and, 565–66
project planning (agile method) with, 681–82
task cards, 79–80
user testing, 249–51
utility services, 534, 548

V

V & V (verification and validation), 58, 227–29, 356. *See also* testing; validation
V-model, 60
vacation package workflow, 542, 544–45
validation (software validation), 20, 69, 58–60
engineering activities for, 23, 44
requirements, 55, 129–30, 135
testing, 58–60, 227–29
verification *v.*, 227–29
validity checks, 129, 326–27, 399
vehicle dispatcher system, 448–49
velocity (Scrum), 85
verifiability, 129
verification (software verification)
cost effectiveness of, 357
formal methods and, 300, 356–59
goal of, 228
levels of confidence, 228–29
model checking, 300, 358–59
safety engineering, 356–59
validation *v.*, 227–29
version control (VC) systems, 731, 735, 753
version management (VM), 215, 216, 731, 735–40, 753
vertical software packages, 20
views, architectural, 173–175, 192
Virtual Learning Environment (VLE), 38
virtual systems, 588
visibility of information, 325–26
volatile requirements, 132
VOLERE requirements engineering method, 123–24
vulnerability, 377, 378, 391, 401, 402

W

waterfall model, 45, 47–49
weather information database, 531–32
weather stations, *see* wilderness weather stations
web application frameworks (WAFs), 444
web-based systems, 27–28
web services, 27, 52, 521, 524–33. *See also* services; WSDL
browser development, 27, 521
business process model and, 544–46, 548
business, 534, 541–47, 548
classification of, 534, 548
clouds, 27, 532
components for, 526–29
composition (construction) of, 541–47
coordination, 534, 548
defined, 27, 521
http and https protocols, 530–31
interactive transaction-based applications, 25

interfaces, 28, 528
resource operations, 530
RESTful approach and, 529–33, 544
reusable components as, 52, 526–29, 542
service-oriented architecture (SOA) and, 524–29
SOA approach, 524
software development and, 541–47, 548
standards, 525–26
testing, 543, 546–47
utility, 534, 548
WSDL interface, 528
'wicked' problems, 130–31, 286, 301
wilderness weather stations, 36–38
architectural design of, 201–02
availability and reliability of, 289
'collect weather data' sequence chart for, 241
context model for, 199
data collection (sequence diagram) in, 206
data collection system architecture in, 202
data management and archiving system, 36
environment of, 36–37
high-level architecture of, 201
interface specification, 208–09
object class identification, 202–04
object interface of, 233
objects, 203–04
sequence diagram for, 241
sociotechnical system of, 291–92
state diagram, 207–08
station maintenance system, 37
system testing, 240–41
use case model for, 200–01
work environments, 663
work flow representation (UML), 143–44
workflow, 83, 452, 542, 543, 544–46, 548
wrapping, legacy system, 278, 442, 540
WS-BPEL, 525, 526, 544, 546
WSDL (Web Service Definition Language), 526, 527–29, 537, 540, 544
message exchange, 527–29, 537
model elements, 527–28
service deployment and, 540
web service interface, 528

X

XML, 470, 525, 527–529
language processing, 186, 189, 191, 470, 544
namespaces, 528–29
service descriptions, 528–29
web services and, 525
WS-BPEL workflow models, 544, 546
WSDL message exchange, 527–29
XML-based protocols, 521

This page intentionally left blank

Author Index

A

Abbott, R., 202, 224
Abdelshafi, I., 87, 100
Abrial, J. R., 49, 71, 300, 304, 357, 370
Abts, C., 459, 460, 462, 594, 608, 684, 688, 691, 694, 699
Addy, E., 476, 489
Aiello, B., 731, 754, 755
Alexander, C., 209, 224
Alford, M., 552, 579
Ali Babar, M., 169, 194
Allen, R., 459, 460, 463
Ambler, S. W., 89, 95, 98, 99, 140, 162, 165
Ambrosio, A. M., 341, 372
Amelot, A., 300, 304
Anderson, E. A., 300, 305
Anderson, R. J., 495
Anderson, R., 376, 402, 405, 406
Andrea, J., 244, 254
Andres, C., 98, 680, 699
Appleton, B., 175, 194, 754
Arbon, J., 252
Arisholm, E., 84, 99
Armour, P., 696
Arnold, S., 552, 579
Ash, D., 275, 282
Atlee, J. M., 135
Avizienis, A. A., 286, 303, 304, 323, 338

B

Badeau, F., 300, 304
Balcer, M. J., 162, 165
Ball, T., 300, 305, 358, 361, 370
Bamford, R., 709, 729, 734, 755
Banker, R. D., 275, 282
Basili, V. R., 73, 100
Bass, B. M., 655, 666
Bass, L., 169, 170, 175, 192, 194
Baumer, D., 446, 462
Baxter, G., 559, 579
Bayersdorfer, M., 221, 224
Beck, K., 71, 77, 80, 98, 99, 100, 203, 224, 242, 254, 279, 282, 680, 699
Beedle, M., 71, 85, 100
Behm, P., 356, 371
Belady, L., 271
Bell, R., 347
Bellouiti, S., 87, 100
Bennett, K. H., 257, 282
Benoit, P., 356, 371
Bentley, R., 125, 137
Berczuk, S. P., 175, 194, 754
Bernstein, A. J., 186, 195
Bernstein, P. A., 498, 519
Berry, G., 612, 637
Bezier, B., 235, 254
Bicarregui, J., 300, 302, 303, 305
Bird, J., 280

Bird, J., 90, 100
Bishop, P., 361, 371
Bjorke, P., 563, 579
Blair, G., 491, 517, 519
Bloomfield, R. E., 361, 371
Bochot, T., 300, 305, 358, 371
Boehm, B. W., 40, 45, 48, 71, 98, 227–28, 254, 459, 460, 462, 594, 608, 649, 666, 683, 684, 687, 688, 691, 694, 695, 697, 699
Bollella, G., 619, 637
Booch, G., 140, 165, 166, 170, 193, 194
Bosch, J., 169, 173, 180, 194
Bott, F., 31, 42
Bounimova, E., 300, 305
Brambilla, M., 139, 159, 163, 165
Brant, J., 80, 100, 279, 282
Brazendale, J., 347
Brereton, P., 517
Brilliant, S. S., 324, 338
Brook, P., 552, 579
Brooks, E. P., 665
Brown, A. W., 98, 684, 688, 699
Brown, L., 376, 405, 407
Bruno, E. J., 619, 637
Budgen, D., 517
Burns, A., 619, 631, 634, 635, 637
Buschmann, F., 175, 194, 195, 209, 224, 225
Buse, R. P. L., 726, 729

C

Cabot, J., 139, 159, 163, 165, 488
Calinescu, R. C., 300, 305, 583, 592, 609
Carollo, J., 252
Cha, S. S., 349, 371
Chapman, C., 220, 225
Chapman, R., 300, 305, 404, 406
Chaudron, M. R. V., 175, 195
Checkland, P., 559, 579
Chen, L., 169, 194
Cheng, B. H. C., 135
Chidamber, S., 721, 729
Chrissis, M. B., 67, 71, 734, 755
Christerson, M., 125, 137, 144, 165
Chulani, S., 684, 688, 699
Clark, B. K., 683, 684, 688, 691, 694, 699
Cleaveland, R., 371
Clements, P., 169, 170, 175, 192, 194
Cliff, D., 583, 592, 609
Cloutier, R., 563, 579
Cohn, M., 680, 697, 699
Coleman, D., 275, 282
Collins-Sussman, B., 216, 225, 735, 755
Connaughton, C., 727
Conradi, R., 69
Cook, B., 300, 305
Cooling, J., 627, 637
Coplien, J. O., 175, 194
Coulouris, G., 491, 517, 519
Councill, W. T., 467, 489
Crabtree, A., 117, 137
Cranor, L., 398, 406
Crnkovic, I., 487, 488
Cunningham, W., 84, 100, 203, 224
Curbera, F., 544, 550
Cusamano, M., 231, 254

D

Daigneau, R., 548
Dang, Y., 719, 726, 729
Datar, S. M., 275, 282
Davidsen, M. G., 272, 282
Davis, A. M., 102, 137
Deemer, P., 88, 100
Dehbonei, B., 356, 371
Deibler, W. J., 709, 729, 734, 755
Delmas, D., 356, 372
Delseny, H., 356, 357, 372
DeMarco, T., 665
den Haan, J., 159, 165
Devnani-Chulani, S., 688, 691, 694, 699
Dijkstra, E. W., 227, 254
Dipti, 282
Dollimore, J., 491, 517, 519
Douglass, B. P., 299, 305, 617, 620, 637
Duftler, M., 544, 550
Dunteman, G., 655, 666
Duquenoy, P., 31, 42
Dybä, T., 69, 84, 99

E

Ebert, C., 611, 635, 637
Edwards, J., 507, 519
El-Amam, K., 721, 729
Ellison, R. J., 425, 432, 434
Erickson, J., 140, 165
Erl, T., 526, 534, 548, 550
Erlikh, L., 256, 282

F

Fagan, M. E., 230, 254, 713, 729
Fairley, R. E., 563, 579
Faivre, A., 356, 371
Fayad, M. E., 446, 462
Fayoumi, A., 602, 609
Feathers, M., 280
Fielding, R., 530, 550
Firesmith, D. G., 383, 406
Fitzgerald, J., 300, 302, 303, 305, 735, 755
Fitzpatrick, B., 216, 225
Fogel, K., 222
Fowler, M., 80, 100, 279, 282
Fox, A., 517
Frank, E., 726, 729
Freeman, A., 28, 42

G

Gabriel, R. P., 581, 583, 607, 609
Gagne, G., 616, 637
Galin, D., 727
Gallis, H., 84, 99
Galvin, P. B., 616, 637
Gamma, E., 175, 194, 209, 210, 222, 225, 444, 463
Garfinkel, S., 398, 406
Garlan, D., 172, 175, 191, 192, 195, 459, 460, 461, 463
Gokhale, A., 443, 445, 463
Gotterbarn, D., 29, 40, 42
Graydon, P. J., 362, 371
Gregory, G., 82, 100, 233, 243, 254
Griss, M., 443, 463, 478, 489
Gryczan, G., 446, 462

H

Hall, A., 300, 305, 404, 406
Hall, E., 644, 666
Hall, M. A., 726, 729
Hamilton, S., 358, 371
Han, S., 719, 726, 729
Harel, D., 156, 165, 617, 637
Harford, T., 726, 729
Harkey, D., 507, 519
Harrison, N. B., 175, 194
Hatton, L., 325, 338
Heimdahl, M. P. E., 300, 305
Heineman, G. T., 467, 489
Helm, R., 175, 194, 209, 210, 222, 225, 444, 463
Henney, K., 175, 194, 209, 224
Heslin, R., 663, 666
Hitchins, D., 581, 608
Hnich, B., 487
Hofmeister, C., 174, 195
Holdener, A. T., 28, 42, 445, 463, 512, 519
Hollnagel, E., 409, 417–18, 434
Holtzman, J., 552, 579
Holzmann, G. J., 336, 358, 371
Hopkins, R., 94, 100, 256, 282
Horowitz, E., 683, 684, 688, 699
Howard, M., 405
Hudepohl, J. P., 360, 372
Hull, R., 151, 165
Humphrey, 67
Humphrey, W., 702, 713, 729
Hutchinson, J., 162, 165

I

Ince, D., 709, 729

J

Jackson, K., 552, 579
Jacobson, I., 24, 41, 42, 125, 137, 140, 144, 165, 166, 443, 463, 478, 489
Jain, P., 175, 195, 209, 225
Jeffrey, R., 727
Jeffries, R., 81, 84, 100, 140, 165, 242, 254
Jenkins, K., 94, 100, 256, 282
Jenney, P., 404, 407
Jhala, R., 300, 305, 358, 371
Joannou, D., 602, 609
Johnson, D. G., 31, 42
Johnson, R., 175, 194, 209, 210, 222, 225, 444, 463
Jones, C., 280, 611, 635, 637
Jones, T. C., 256, 282
Jonsson, P., 125, 137, 144, 165, 443, 463, 478, 489
Jonsson, T., 487

K

Kaner, C., 246, 254
Kawalsky, R., 602, 609
Kazman, R., 169, 170, 175, 192, 194
Keen, J., 583, 592, 609
Kelly, T., 583, 592, 609
Kemerer, C. F., 275, 282, 721, 729
Kennedy, D. M., 563, 579
Kerievsky, J., 279, 282
Kessler, R. R., 84, 100
Khalaf, R., 544, 550
Kifer, M., 186, 195
Kilner, S., 90, 100
Kindberg, T., 491, 517, 519
King, R., 151, 165
Kircher, M., 175, 195, 209, 225
Kitchenham, B., 718, 727, 729
Kiziltan, Z., 487
Klein, M., 581, 583, 607, 609
Kleppe, A., 485, 489
Knight, J. C., 324, 338, 362, 371
Knoll, R., 446, 462
Koegel, M., 161, 165
Konrad, M., 67, 71, 734, 755
Kopetz, H., 635
Korfiatis, P., 563, 579
Koskela, L., 59, 71
Koskinen, J., 275, 282
Kotonya, G., 473, 489
Kozlov, D., 275, 282
Krogstie, J., 272, 282
Krutchen, P., 46, 71, 173, 175, 195
Kuehl, S., 552, 579
Kumar, Y., 280
Kwiatkowska, M. Z., 300, 305, 358, 371, 583, 592, 609

L

Lamport, L., 495
Landwehr, C., 286, 303, 304
Lane, A., 392, 407
Lange, C. F. J., 175, 195
Laprie, J. C., 286, 303, 304, 409, 434
Larman, C., 73, 100, 222
Larsen, P.G., 300, 302, 303, 305
Lau, K-K., 466, 470, 487, 489
Laudon, K., 31, 42
LeBlanc, D., 405
Ledinot, E., 357, 371
Lee, E. A., 612, 637
Leffingwell, D., 95, 100
Lehman, M., 271
Leme, F., 82, 100, 233, 243, 254
Leveson, N. G., 324, 338, 368
Leveson, N. G., 349, 371
Levin, V., 300, 305, 358, 361, 370
Lewis, B., 521, 550
Lewis, P. M., 186, 195
Leymann, F., 532, 550
Lichtenberg, J., 300, 305
Lidman, S., 41, 42
Lientz, B. P., 256, 282
Lilienthal, C., 446, 462
Linger, R. C., 230, 254, 332, 338, 425, 432, 434
Lipson, H., 425, 434
Lister, T., 665

Loeliger, J., 216, 225, 735, 755
Lomow, G., 524, 550
Longstaff, T., 425, 432, 434
Loope, J., 753, 755
Lopes, R., 359, 371
Lou, J-G., 719, 726, 729
Lovelock, C., 521, 550
Lowther, B., 275, 282
Lutz, R. R., 238, 254, 371
Lutz, R. R., 340, 371
Lyu, M. R., 336, 338

M

Madachy, R., 683, 684, 688, 699
Madeira, H., 341, 372
Maier, M. W., 582, 583, 588, 589, 599–600, 607, 609
Majumdar, R., 300, 305, 358, 371
Marciniak, J. J., 69
Markkula, J., 275, 282
Marshall, J. E., 663, 666
Martin, D., 117, 137, 175, 195
Martin, R. C., 244, 254
Maslow, A. A., 383, 666
Massol, V., 82, 100, 233, 243, 254
McCay, B., 552, 579
McComb, S. A., 563, 579
McConnell, S., 713, 729
McCullough, M., 216, 225, 735, 755
McDermid, J., 583, 592, 609
McDougall, P., 510, 519
McGarvey, C., 300, 305
McGraw, G., 333, 338, 396, 407
McMahon, P. E., 41, 22
Mead, N. R., 425, 434
Mejia, F., 356, 371
Mellor, S. J., 159, 162, 165
Melnik, G., 81, 100, 242, 254
Menzies, T., 719, 725, 726, 727, 729
Meunier, R., 175, 194, 209, 225
Meyer, B., 485, 489
Meynadier, J-M., 356, 371
Miers, D., 544, 550
Mili, A., 476, 489
Mili, H., 476, 489
Miller, K., 29, 40, 42
Miller, S. P., 300, 305
Mitchell, R. M., 263, 282
Monate, B., 357, 371
Monk, E., 455, 463
Moore, A., 425, 432, 434
Morisio, M., 453, 460, 461, 463
Mostashari, A., 563, 579
Moy, Y., 357, 371
Mulder, M., 87, 100
Musa, J. D., 334, 338
Muskens, J., 175, 195

N

Nagappan, N., 360, 372
Nascimento, L., 461
Natarajan, B., 443, 445, 463
Naur, P., 19, 42
Newcomer, E., 524, 550
Ng, P-W., 41, 42
Nii, H. P., 180, 195
Nord, R., 174, 195
Norman, G., 358, 371
Northrop, L., 581, 583, 607, 609
Nuseibeh, B., 169, 194

O

O'Hanlon, C., 274, 282
Ockerbloom, J., 459, 460, 463
Oliver, D., 552, 579
Oman, P., 275, 282
Ondrusek, B., 300, 305
Opdahl, A. L., 386, 407
Opdyke, W., 80, 100, 279, 282
Oram, A., 510, 517, 519
Orfali, R., 507, 519
Ould, M., 644, 666
Overgaard, G., 125, 137, 144, 165
Owens, D., 552, 579

P

Paige, R., 583, 592, 609
Paries, J., 432, 434
Parker, D., 358, 371
Parnas, D., 296, 302, 305
Patel, S., 162, 166
Patterson, D., 517
Pautasso, C., 532, 550
Perrow, C., 342, 343, 371
Pfleeger, C. P., 376, 377, 407
Pfleeger, S. L., 376, 377, 407
Pilato, C., 216, 225, 735, 755
Pooley, R., 126, 137, 163
Poore, J. H., 230, 254, 332, 338
Pope, A., 466, 489, 493, 519
Prowell, S. J., 230, 254, 332, 338
Pullum, L., 318, 336, 338

Q

Quinn, M. J., 40

R

Rajamani, S. K., 300, 305, 358, 361, 370
Rajlich, V. T., 257, 282
Randell, B., 19, 42, 286, 303, 304, 307, 338
Ray, A., 368
Rayhan, S., 727
Raymond, E. S., 219, 225
Reason, J., 418, 420–21, 434
Regan, P., 358, 371
Reifer, D., 684, 688, 699
Richardson, L., 531, 550
Riehle, D., 446, 462
Rittel, H., 130, 137, 562, 579, 592, 609
Ritter, G., 637
Roberts, D., 80, 100, 279, 282
Robertson, J., 123, 135, 137
Robertson, S., 123, 135, 137
Rodden, T., 125, 137
Rodriguez, A., 548
Rogerson, S., 29, 40, 42
Rohnert, H., 175, 194, 195, 209, 225
Rosenberg, F., 544, 550
Rouncefield, M., 162, 165
Royce, W. W., 47, 71, 98, 687, 699
Rubin, K. S., 78, 85, 98, 100, 680, 699
Ruby, S., 531, 550
Rumbaugh, J., 140, 165, 166
Ryan, P., 754

S

Sachs, S., 731, 754, 755
Sakkinen, M., 275, 282
Sametinger, J., 470, 472, 489
Sami, M., 69
Sanderson, D., 516, 519
Sarris, S., 445, 463, 512, 519
Sawyer, P., 125, 137
Scacchi, W., 69
Schatz, B., 87, 100
Schmidt, D. C., 175, 194, 195, 209, 224, 225, 443, 445, 446, 462, 463, 581, 583, 607, 609
Schneider, S., 357, 371
Schneier, B., 384, 396, 407
Schoenfield, B., 392, 407
Schuh, P., 754
Schwaber, K., 71, 85, 100
Scott, J. E., 456, 463
Scott, K., 159, 165
Selby, R. W., 231, 254, 683, 699
Shaw, M., 172, 175, 191, 192, 195
Shimeall, T. J., 349, 371
Shou, P. K., 296, 305
Shrum, S., 67, 71, 734, 755
Siau, K., 140, 165
Silberschaltz, A., 616, 637
Sillitto, H., 578, 596, 600, 609
Silva, N., 341, 359, 371, 372
Sindre, G., 386, 407
Sjøberg, D. I. K., 69, 84, 99
Smart, J. F., 743, 755
Snipes, W., 360, 372
Sommerlad, P., 175, 194, 209, 225
Sommerville, I., 117, 135, 137, 175, 195, 461, 559, 579, 583, 592, 607, 609
Soni, D., 174, 195

Souyris, J., 356, 372
Spafford, E., 399, 401, 407
Spence, I., 41, 42
St. Laurent, A., 220, 225
Stafford, J., 488
Stahl, T., 159, 166
Stal, M., 205, 209, 225
Stallings, W., 376, 405, 407, 616, 637
Stapleton, J., 71, 100
Steece, B., 684, 688, 699
Stevens, P., 126, 137, 163
Stevens, R., 552, 579, 583, 609
Stewart, J., 574, 579
Stoemmer, P., 697
Storey, N., 349, 372
Strunk, E. A., 362, 371
Suchman, L., 117, 137
Swanson, E. B., 256, 282
Swartz, A. J., 294, 305
Szyperski, C., 467, 474, 487, 488, 489

T

Tahchiev, P., 82, 100, 233, 243, 254
Tanenbaum, A. S., 491, 519
Tavani, H. T., 31, 42
Thayer, R. H., 552, 579
Thayer, R. H., 563, 579
Tian, Y., 602, 609
Torchiano, M., 453, 460, 461, 463
Torres-Pomales, W., 318, 338
Trammell, C. J., 230, 254, 332, 338
Trimble, J., 299, 305
Tully, C., 552, 579
Turner, M., 517
Turner, R., 45, 71
Twidale, M., 125, 137

U

Ulrich, W. M., 276, 282
Ulsund, T., 69
Ustuner, A., 300, 305

V

Valeridi, R., 697
van Schouwen, J., 296, 305
Van Steen, M., 491, 519
van Vliet, M., 87, 100
Vandermerwe, S., 521, 550
Veras, P. C., 341, 372
Vicente, D., 359, 371
Viega, J., 396, 405, 407
Vieira, M., 341, 372
Villani, E., 341, 372
Viller, S., 117, 137
Virelizier, P., 300, 305, 358, 371
Vlissides, J., 175, 194, 209, 210, 222, 225, 444, 463
Voas, J., 333, 338
Voelter, M., 159, 166
Vogel, L., 32, 42, 218, 225
Vouk, M. A., 360, 372

W

Waeselynck, H., 300, 305, 358, 371
Wagner, B., 455, 463
Wagner, L. G., 300, 305
Wallach, D. S., 512, 519
Wang, Z., 466, 470, 487, 489
Warmer, J., 485, 489
Warren, I., 266, 282
Webber, M., 130, 137, 562, 579, 592, 609
Weils, V., 356, 357, 358, 372
Weinberg, G., 83, 100
Weiner, L., 203, 225
Weinreich, R., 470, 472, 489
Weise, D., 159, 165
Wellings, A., 619, 631, 634, 635, 637
Westland, C., 683, 699
Whalen, M. W., 300, 305
Wheeler, D. A., 396, 407
Wheeler, W., 52, 71, 473, 489
White, J., 52, 71, 473, 489
White, S. A., 544, 550
White, S., 552, 579
Whittaker, J. A., 237, 242, 252, 254

Whittle, J., 162, 165
Wiels, V., 300, 305, 371
Wilkerson, B., 203, 225
Willey, A., 552, 579
Williams, L., 84, 100, 360, 372
Williams, R., 574, 579
Wimmer, M., 139, 159, 163, 165
Wirfs-Brock, R., 203, 225
Witten, I. H., 726, 729
Woodcock, J., 300, 302, 303, 305
Woods, D., 432, 434
Wreathall, J., 432, 434
Wysocki, R. K., 665

X

Xie, T., 719, 726, 729

Y

Yacoub, S., 476, 489
Yamaura, T., 252

Z

Zelkowitz, M., 637
Zhang, D., 719, 726, 729
Zhang, H., 719, 726, 729
Zhang, Y., 162, 166
Zheng, J., 360, 372
Zimmermann, O., 532, 550
Zimmermann, T., 719, 725, 726, 727, 729
Zullighoven, H., 446, 462
Zweig, D., 275, 282

SOFTWARE ENGINEERING

CO3001

LECTURE 2&3 – REQUIREMENTS ENGINEERING

Anh Nguyen Duc
Jingyue Li
Daniela Soares Cruzes

NTNU

Adapted from https://iansommerville.com/software-engineering-book/slides/

TOPICS COVERED

- ✓ Requirements engineering & Requirements
- ✓ Requirements elicitation
- ✓ Requirements specification
- ✓ Requirements validation
- ✓ Requirements management
- ✓ Goal oriented requirement engineering
- ✓ Quality of requirements
- ✓ Non-functional requirements
- ✓ Requirement traceability

TOPICS COVERED

- ✓ Requirements engineering & Requirements
- ✓ Requirements elicitation
- ✓ Requirements specification
- ✓ Requirements validation
- ✓ Requirements management
- ✓ Goal oriented requirement engineering
- ✓ Quality of requirements
- ✓ Non-functional requirements
- ✓ Requirement traceability

REQUIREMENTS

EFFECTS OF INADEQUATE REQUIREMENTS DEVELOPMENT – AIRBUS

- ✓ **Requirement**: *Reverse thrust may only be used <u>when the airplane is landed</u>.*
- ✓ **Translation**: *Reverse thrust may only be used <u>while the wheels are rotating</u>.*
- ✓ **Implementation**: *Reverse thrust may only be used <u>while the wheels are rotating fast enough</u>.*

- **Situation**: Rainstorm – aquaplaning
- **Result**: Crash due to overshooting the runway!
- **Problem**: Erroneous in the requirement phase

The Ariane 5 accident – 1

- Single root cause failure!
- **The ”bug”: attitude deviation** stored as 2-byte integer (max value 65,535) instead of 4-byte (max value 4,294,967,295)
- SW module was reused from Ariane 4
- Insufficient V&V of detailed requirements: larger attitude deviation tolerated in Ariane 5 than in Ariane 4
- Ariane 5 production cost 10 years and $7 billion; luckily, no victims because it was unmanned.

65,53**5** = 00000000 00000000 11111111 11111111

65,53**6** = 00000000 00000001 00000000 00000000

OTHER SOFTWARE RELATED ACCIDENTS & INCIDENTS

- **Accidents & incidents**
  - Alarm flooding, **power distribution** failure, BP Grangemouth Scotland, 29th May - 10th June 2000
  - failure in a data bus + faulty logic in the software => engine power loss, **Airbus** A340-642, 2005
  - failed accelerometer + software bug => Faulty airspeed metering, **Boeing** 777-200, 2005
  - Software bug => shutdown of radio communication between **ATC and aircraft**, 2004. disrupted 800 flights
  - Safety-related software flaws => recall of 200,000 **pacemakers** in 1990-2000
  - **radiotherapy machines** attacked by computer **viruses**, 2005
  - Buggy software incorporate computer + connection between corporate and control systems networks => shutdown of the **nuclear power plant**, USA 2008
- Many software failures are actually ”bugs” in the **detailed requirement specifications,** i.e., poor understanding of the very detailed requirements

Korean Air 747 in Guam, 200 deaths (1997): incorrect configuration of ”minimum altitude” warning system

How the customer explained it
How the project leader understood it
How the engineer designed it
How the programmer wrote it
How the sales executive described it
How the project was documented
What operations installed
How the customer was billed
How the helpdesk supported it
What the customer really needed

REQUIREMENTS ENGINEERING

- ✓ The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.

WHAT IS A REQUIREMENT?

Requirement engineering = establishing the ***services*** that the customer requires from a system and the ***constraints*** under which it operates and is developed.

- Requirement = the descriptions of
  - the system services
  - and constraints
- It may range
  - from a high-level abstract statement
  - to a detailed mathematical functional specification.
- May serve a dual function
  - The basis for a bid for a contract - must be open to interpretation;
  - The basis for the contract itself - must be in detail;

STAKEHOLDERS

- Any person or organization who is affected by the system in some way and so who has a legitimate interest
- Stakeholder types
  - End users
  - System managers
  - System owners
  - External stakeholders

FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS

- Functional requirements
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  - May state what the system should not do.
- Non-functional requirements
  - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - Often apply to the system as a whole rather than individual features or services.

  - Domain requirements
    - Constraints on the system from the domain of operation

NON-FUNCTIONAL REQUIREMENTS IMPLEMENTATION

- Non-functional requirements may be more critical than functional requirements.
  - If these are not met, the system may be useless.
- Non-functional requirements may affect the overall architecture of a system
  - rather than the individual components
  - cross-cutting concern
- A single non-functional requirement
  - may generate a number of related functional requirements
  - and may also generate requirements that restrict existing requirements.

NON-FUNCTIONAL REQUIREMENTS

- Define system properties and constraints
  - Properties: reliability, response time and storage requirements.
  - Constraints: I/O device capability, system representations, etc.

- Non-functional requirements may be more critical than functional requirements.
  - If these are not met, the system may be useless.

NON-FUNCTIONAL REQUIREMENTS IMPLEMENTATION

- Non-functional requirements may affect the overall architecture of a system
  - rather than the individual components
  - cross-cutting concern

- A single non-functional requirement
  - may generate a number of related functional requirements
  - and may also generate requirements that restrict existing requirements.

REQUIREMENTS ENGINEERING PROCESS

REQUIREMENTS ENGINEERING PROCESSES

- Processes to “generate” all require
- Generic activities common to all pr
  - Requirements elicitation;
  - Requirements analysis;
  - Requirements validation;
  - Requirements management.
- In practice, RE is an iterative activi

"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."

The RE process:

A SPIRAL VIEW OF THE REQUIREMENTS ENGINEERING PROCESS

A AGILE VIEW OF THE REQUIREMENTS ENGINEERING PROCESS

Iterative Enhancement Model

REQUIREMENT ELICITATION

REQUIREMENTS ELICITATION AND ANALYSIS

- ~ requirements elicitation or requirements discovery.

- Work with customers to find out:
  - the application domain, the services and the operational constraints (system performance, hardware constraints, etc.).
- May involve
  - end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called stakeholders.

PROBLEMS OF REQUIREMENTS ELICITATION

- ✓ Stakeholders don’t know what they really want.
- ✓ Stakeholders express requirements in their own terms.
- ✓ Different stakeholders may have conflicting requirements.
- ✓ Organisational and political factors may influence the system requirements.
- ✓ The requirements change during the analysis process.
  - New stakeholders may emerge and the business environment change.

THE REQUIREMENTS ELICITATION AND ANALYSIS PROCESS

REQUIREMENTS DISCOVERY

- To gather information about the required and existing systems and distil the user and system requirements from this information.

- Main concerns:
  - Stakeholders
  - Discovery techniques/approaches/…

DISCOVERY TECHNIQUE - INTERVIEWING

- Part of most RE processes.
- Types of interview
  - Closed vs Open => mixed?
- Be effective
  - Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
  - Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.

DISCOVERY TECHNIQU

- Observational technique
  - used to understand operat
    derive support requiremen
- How
  - A social scientist spends a considerable time observing and analysing how people actually work.
  - People do not have to explain or articulate their work.
  - Social and organisational factors of importance may be observed.

STORIES AND SCENARIOS

- ✓ Scenarios and use
  of how a system c
- ✓ Stories and scenar
  system may be us
- ✓ Because they are
  stakeholders can r
  comment on their
  story.

REQUIREMENT ELICITATION TECHNIQUES

	Interviews	Workshops	Focus groups	Observations	Questionnaires	System interface analysis	User interface analysis	Document analysis
Mass-market software	x		x		x			
Internal corporate software	x	x	x	x		x		x
Replacing existing system	x	x		x		x	x	x
Enhancing existing system	x	x				x	x	x
New application	x	x				x		
Packaged software implementation	x	x		x		x		x
Embedded systems	x	x				x		x
Geographically distributed stakeholders	x	x			x			

Suggested elicitation techniques by project characteristics

REQUIREMENTS SPECIFICATION

REQUIREMENTS SPECIFICATION

- The process of writing down the user and system requirements in a requirements document.

- Notes:
  - User requirements have to be understandable by end-users and customers who do not have a technical background.
  - System requirements are more detailed requirements and may include more technical information.
  - The requirements may be part of a contract for the system development

WAYS OF WRITING A SYSTEM REQUIREMENTS SPECIFICATION

Notation	Description
Natural language	Sentences in natural language. Each sentence should express one requirement.
Structured natural language	Natural language statements on a standard form or template
Design description languages	Uses a language like a programming language, but with more abstract features
Graphical notations	Graphical models, supplemented by text annotations (best for functional requirements); UML use case and sequence diagrams are commonly used.
Mathematical specifications	Based on mathematical concepts such as finite-state machines or sets; Can reduce the ambiguity but hard to understand (and hard to check manually)

NATURAL LANGUAGE SPECIFICATION

- Used for writing requirements because it is expressive, intuitive and universal.
  - The requirements can be understood by users and customers.

- Problems
  - Lack of clarity: Precision is difficult without making the document difficult to read.
  - Requirements confusion: Functional and non-functional requirements tend to be mixed-up.
  - Requirements amalgamation: Several different requirements may be expressed together.

An insulin pump control system

An insulin pump is a medical system that simulates the operation of the pancreas (an internal organ). The software controlling this system is an embedded system, which collects information from a sensor and controls a pump that delivers a controlled dose of insulin to a user.

People who suffer from diabetes use the system. Diabetes is a relatively common condition where the human pancreas is unable to produce sufficient quantities of a hormone called insulin. Insulin metabolises glucose (sugar) in the blood. The conventional treatment of diabetes involves regular injections of genetically engineered insulin. Diabetics measure their blood sugar levels using an external meter and then calculate the dose of insulin that they should inject.

EXAMPLE REQUIREMENTS FOR THE INSULIN PUMP SOFTWARE SYSTEM

- ✓ Req 3.2. The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes.

- ✓ Req 3.6. The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1.

STRUCTURED SPECIFICATIONS

- Writing on a standard form or template:
  - Name
  - Inputs, outputs
  - The information needed for the computation
  - Action
  - Pre and post conditions (if appropriate)
  - The side effects (if any)

- This works well for some types of requirements e.g. requirements for embedded control system

A
RE

Insulin Pump/Control Software/SRS/3.3.2	
Function	Compute insulin dose: safe sugar level
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.
Inputs	Current sugar reading (r2); the previous two readings (r0 and r1).
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose—the dose in insulin to be delivered
Destination	Main control loop.
Action	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.
Requirements	Two previous readings so that the rate of change of sugar level can be computed
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin

TABULAR SPECIFICATION

- Particularly useful when you have to define a number of possible alternative courses of action.
- Example:

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 – r1) < (r1 – r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing ((r2 – r1) ≥ (r1 – r0))	CompDose = round ((r2 – r1)/4) **If** rounded result = 0 **then** CompDose = MinimumDose

USE CASES

- Use-cases are a kind of scenario
  - identify the actors in an interaction and which describe the interaction itself
  - Included in the UML
- A set of use cases should describe all possible interactions with the system.
- UML sequence diagrams may be used to add detail to use-cases
  - show the sequence of event processing in the system

USE CASE DIA

- Actors
- Use cases
- Association
- System boundary boxes

Partial use case diagram for the Chemical Tracking System (CTS)

NTNU

USE CASE TEMPLATE AND EXAMPLE

ID and Name:	UC-4 Request a Chemical		
Created By:	Lori	Date Created:	8/22/13
Primary Actor:	Requester	Secondary Actors:	Buyer, Chemical Stockroom, Training Database
Description:	The Requester specifies the desired chemical to request by entering its name or chemical ID number or by importing its structure from a chemical drawing tool. The system either offers the Requester a container of the chemical from the chemical stockroom or lets the Requester order one from a vendor.		
Trigger:	Requester indicates that he wants to request a chemical.		
Preconditions:	PRE-1. User's identity has been authenticated. PRE-2. User is authorized to request chemicals. PRE-3. Chemical inventory database is online.		
Postconditions:	POST-1. Request is stored in the CTS. POST-2. Request was sent to the Chemical Stockroom or to a Buyer.		

USE CASE TEMPLATE AND EXAMPLE (CONT')

ID and Name:	UC-4 Request a Chemical
Normal Flow:	**4.0 Request a Chemical from the Chemical Stockroom** 1. Requester specifies the desired chemical. 2. System lists containers of the desired chemical that are in the chemical stockroom, if any. 3. System gives Requester the option to View Container History for any container. 4. Requester selects a specific container or asks to place a vendor order (see 4.1). 5. Requester enters other information to complete the request. 6. System stores the request and notifies the Chemical Stockroom.
Alternative Flows:	**4.1 Request a Chemical from a Vendor** 1. Requester searches vendor catalogs for the chemical (see 4.1.E1). 2. System displays a list of vendors for the chemical with available container sizes, grades, and prices. 3. Requester selects a vendor, container size, grade, and number of containers. 4. Requester enters other information to complete the request. 5. System stores the request and notifies the Buyer.
Exceptions:	**4.1.E1 Chemical Is Not Commercially Available**

DATA FLOW DIAGRAM

- ✓ How data moves through a system

NTNU

DECISION TABLE/DECISION TREE

- ✓ Complex logics

Requirement Number					
Condition	**1**	**2**	**3**	**4**	**5**
User is authorized	F	T	T	T	T
Chemical is available	—	F	T	T	T
Chemical is hazardous	—	—	F	T	T
Requester is trained	—	—	—	F	T
Action					
Accept request			X		X
Reject request	X	X		X	

STATE-TRANSITION DIAGRAM

- ✓ A set of complex state changes in natural language creates a high probability of overlooking a permitted state change or including a disallowed change.

THE SOFTWARE REQUIREMENTS DOCUMENT

- The software requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

REQUIREMENT VALIDATION

REQUIREMENTS VALIDATION

- ✓ Requirements error can be costly !

Cost to Fix Requirements Error (in Ratios)			
Development Discipline/Phase	**Composite of Studies (NASA)***	**NASA Software#**	**Davis' Composite+**
Requirements	1 (baseline)	1 (baseline)	1 (baseline)
Design	5x	8x	2.5x – 5x
Code	10x	16x	5x – 10x
Test	50.5x	21x	Unit Test: 10x – 20x Acceptance Test: 25x – 50x
Post-Deployment	n/a	29x	100x – 200x

* NASA reviewed McGibbon (2003), Pavlina (2003), Cigital (2003), Rothman (2002), Hoffman (2001), Rothman (2000), and Boehm (1981) and assessed the median of their data, presented here.
# NASA analyzed its own software projects to assess the relevance of reviewed studies.
# Leffingwell & Widrig reviewed GTE, TRW, IBM, and Davis (2003), who himself reviewed several studies. The results of Davis' work are presented here.

REQUIREMENTS CHECKING

- Validity.
  - Does the system provide the functions which best support the customer’s needs?
- Consistency.
  - Are there any requirements conflicts?
- Completeness.
  - Are all functions required by the customer included?
- Realism.
  - Can the requirements be implemented given available budget and technology
- Verifiability.
  - Can the requirements be checked?

REQUIREMENTS VALIDATION TECHNIQUES

- ✓ Requirements reviews
  - Systematic manual analysis of the requirements.
- ✓ Prototyping
  - Using an executable model of the system to check requirements.
- ✓ Test-case generation
  - Developing tests for requirements to check testability.

REQUIREMENTS MANAGEMENT

"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."

CHANGING REQUIREMENTS

- The business and technical environment of the system always changes after installation.
- The people who pay for a system and the users of that system are rarely the same people.
- Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.

REQUIREMENTS MANAGEMENT

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- New requirements emerge as a system is being developed and after it has gone into use.
- You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements.

REQUIREMENTS MANAGEMENT PLANNING

- ✓ Establishes the level of requirements management detail that is required.

- ✓ Requirements management decisions:
  - Requirements identification
  - A change management process
  - Traceability policies
  - Tool support

REQUIREMENTS CHANGE MANAGEMENT

- Deciding if a requirements change should be accepted
  - Problem analysis and change specification
  - Change analysis and costing
  - Change implementation

RESPONSE TO CHANGE: AGILE APPROACH!

SUMMARY

- ✓ Requirements: what the system should do and constraints on its operation and implementation.
- ✓ Functional requirements = the services
- ✓ Non-functional requirements = constraints (development & use)
  - apply to the system as a whole.
- ✓ The software requirements document (i.e. SRS) is an agreed statement of the system requirements.
- ✓ The RE process is an iterative process
  - requirements elicitation, specification and validation.

GOAL ORIENTED REQUIREMENT ENGINEERING

"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."

WHAT IS GORE [3]?

- GORE is approach and modeling method to
  - Identify goals and sub-goals of software systems
  - Identify who and how to achieve the goals
  - Identify domain hypothesis and conflicts
- Similar principle to use cases and user stories
- Can be applicable to cyber-physical systems and user-system interaction systems

PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- ✓ Anticipating obstacles
- ✓ Identifying domain hypothesis
- ✓ Handling conflicts

GOAL MODEL

- A goal is an objective the system under consideration should achieve
- Goal is condition or situation that can be achieved or not. Goal is used to define the functional requirements of the system.
- Goals provide a precise criterion for sufficient completeness of a requirements specification

GOAL MODEL

- Hierarchy and relationship of goals
- Goals can be of different types

GOAL MODEL USING BEHAVIOR AND SOFT GOAL TYPES

- Soft goal
  - Satisfaction cannot be established in a clear-cut sense
  - Useful for comparing alternative goal refinements
- Hard/behavioral goal
  - Satisfaction can be established through verification

GOAL MODEL USING BEHAVIOR GOAL TYPES

Informal temporal patterns

Achieve[TargetCondition]
Cease[TargetCondition]

Maintain[GoodCondition]
Avoid[BadCondition]

Improve[TargetCondition]
Increase[TargetQuantity]
Reduce[TargetQuantity]
Maximise[ObjectiveFunction]
Minimise[ObjectiveFunction]

- Achieve: Some target property to be eventually satisfied
- Maintain: Some target property to be permanently satisfied
- Avoid: Some target property to be permanently denied
- …

PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- ➢ Anticipating obstacles
- ➢ Identifying domain hypothesis
- ➢ Handling conflicts

ELICITING GOALS

Each sub-goal is refined into finer-grained goals until we reach a software requirement associated with a single agent and expectation (assumption) on the environment.

CASE STUDY: TRAIN TRANSPORT SAFETY

NTNU

SOFT AND BEHAVIOR GOALS

ELICITING NEW GOALS THROUGH “HOW” QUESTIONS

CONTINUE ELICITING NEW GOALS THROUGH "HOW" QUESTIONS

NTNU

GOALS REFINEMENT

Mean-ends **relationship** ( )
Decomposition **relationship** ( )
Contribution **relationship**( )
Dependency **relationship** ( )

ELICITING NEW GOALS THROUGH “WHY” QUESTIONS

- ✓ High level goals are identified by asking WHY questions

AATC SOFT GOALS

How

Why

LINKS:

+ : Contribute
- : Conflict
Double line : OR
Black circle : AND
… : More sub-goals

++ : Contribute strongly
-- : Conflict strongly

NTNU

PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- ➢ Anticipating obstacles
- ➢ Identifying domain hypothesis
- ➢ Handling conflicts

OBJECT MODEL

- Agent: independent active object, e.g.,
  - Speed/Acceleration control system
  - On-board train controller
  - Tracking systems

- Agent has entities
  - E.g., tracking systems have several sensors

AGENT RESPONSIBILITY MODEL

- A **goal** has to be placed under the responsibility of an **agent**.

NTNU

PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- ➢ Anticipating obstacles
- ➢ Identifying domain hypothesis
- ➢ Handling conflicts

OPERATION MODEL

- ✓ Operation is input-output relation over objects
- ✓ Defines state transitions
- ✓ Characterized by pre-, post-, and trigger conditions

OPERATION MODEL – TRAIN CONTROL SYS.

- ✓ Operation: SendCommandMessage
- ✓ Input: Train ID
- ✓ Output: Command to the train
- ✓ Pre-condition: Message not sent
- ✓ Post-condition: Message sent to correct train

State transition

PROCESSES TO CREATE GORE MODELS

- ✓ Initial goal identification
- ✓ Eliciting new goals through WHY and HOW questions
- ✓ Identifying agents and responsibility assignments
- ✓ Identifying operations
- Anticipating obstacles
- Identifying domain hypothesis
- Handling conflicts

ANTICIPATING OBSTACLES

- Obstacles (Possible failures that may negatively impact goal achievement)
  - CommandMsgNotIssued
  - CommandMsgIssuedLate
  - CommandMsgSentToWrongTrain

IDENTIFY DOMAIN HYPOTHESIS

- **Domain property**: A descriptive statement about the problem world.
- **Domain invariants:** Should hold invariably, regardless of how the system behaves. This usually corresponds to some physical laws. E.g. *The light is either on or off*.
- **Domain hypothesis**: To satisfy a service request, an infrastructure/environment to perform the service must be available.

IDENTIFYING DOMAIN HYPOTHESIS

✓ Domain hypothesis related to the goal: *Every track segment leading to a switch is ended with a gate*.

HANDLING GOAL CONFLICTS

CAN GORE HELP DERIVE BETTER REQUIREMENTS ?

- ✓ "Requirements definition must say why a system is needed, based on current or foreseen conditions, which may be internal operations or an external market. It must say what system features will serve and satisfy this context. And it must say how the system is to be constructed. [1]"

Goal models

Agent models

Responsibility models

Operation models

Domain hypotheses

SUMMARY (CONT.)

- ✓ Requirements elicitation and analysis = iterative process
  - requirements discovery, classification and organization, negotiation and requirements documentation.
- ✓ Techniques for requirements elicitation
  - interviews, scenarios, use-cases and ethnography, etc.
- ✓ Requirements validation = checking the requirements
  - for validity, consistency, completeness, realism and verifiability.
- ✓ Business, organizational and technical changes inevitably
  - => changes to the requirements for a software system.
- ✓ Requirements management = managing and controlling the requirement changes.

- Electronic Healthcare Record (EHR) system is a repository of patient data in digital form, stored and exchanged securely, and accessible by multiple authorised users. As a normal flow, medication will be first prescribed by authorized nurses to a patient, after that the medication is dispensed and then taken by the patient. EHR ServerRX deployed in a London hospital, required all the patients to wear a bracelet ID for identification purposes. Nurses had to scan this bracelet before dispensing any medicine in order to reduce the risk of administering the medication to the wrong patient.
- Describe at least three sub goals to the goals
- https://app.diagrams.net/#G1pL-o8b58wxjIz1BakfxWQo-B6EWvYL8W

EXERCIS

GOAL MODEL
Achieve [Prescribed Medication Taken By Patient]
Achieve [Patient ID Verified]
Achieve [Dispensed Medication Taken By Patient]
Achieve [Prescribed Medication Dispensed When Patient ID Verified]

Maintain [Medication in stock]
Avoid [Medicine Overdose]
Achieve [Prescribed Medication Taken By Patient]
Achieve [Patient ID Verified]
Achieve [Prescribed Medication Dispensed when Patient ID Verified]
Achieve [Dispensed Medication Taken By Patient]
Nurse
Patient Wears Identification Bracelet
Achieve [Patient Identification Bracelet Scanned]
Achieve [Prescribed Medication In Patient Drawer]
Achieve [Patient Drawer Opened When Patient Bracelet Scanned
Nurse
Nurse
Electronic Dispensing System

QUALITY OF REQUIREMENTS

"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."

EXAMPLE OF NIGHTMARE REQUIREMENTS

- ✓ The system shall perform at the maximum rating at all times except that in emergencies it shall be capable of providing up to 125% rating unless the emergency condition continues for more than 15 minutes in which case the rating shall be reduced to 105% but in the event that only 95% can be achieved then the system shall activate a reduced rating exception and shall maintain the rating within 10% of the stated values for a minimum of 30 minutes.

More than one requirement in a paragraph
Rambling: like a novel
Vague terms: emergencies

ANOTHER EXAMPLE OF NIGHTMARE REQUIREMENTS

- ✓ The system shall provide general word processing facilities which shall be easy to use by untrained staff and shall run on a thin Ethernet Local Area Network wired into the overhead ducting with integrated interface cards housed in each system together with additional memory if that should be necessary.

More than one requirement in a paragraph
Rambling: like a novel
Let-out clauses: if that should be necessary
Vague words: be easy to use, additional memory

CHARACTERISTICS OF GOOD SOFTWARE REQUIREMENTS SPECIFICATION*

- Complete
- Unambiguous
- Consistent
- Correct
- ✓ Verifiable
- ✓ Traceable
- ✓ Ranked for importance and/or stability
- ✓ Modifiable

* IEEE Std 830-1998

COMPLETENESS

- ✓ Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations (IEEE Std).

- ✓ All possible situations must be covered

  "If X then….", "If Y then…." Must also consider what will happen "If neither X nor Y…"

COMPLETENESS (CONT')

- Automatic door opener

  If the door is closed and a person is detected, then send signal Open_Door.

  If no person is detected after 10 sec., send signal Close_Door.

- What are missing?

UNAMBIGUOUS

- ✓ A software requirement is unambiguous, if and only if, every requirement stated therein has only one interpretation (IEEE Std).
- Ambiguity: Requirement with terms or statements that can be interpreted in different ways.

UNAMBIGUOUS (CONT')

- ✓ Requirement of autonomous cruise control (ACC) system is:

The maximum speed of a vehicle in a busy road shall be 10 mph.

Ambiguity: What type of vehicle do you mean?
What is busy road?

CONSISTENT

- ✓ A software requirement is internally consistent if, and only if, no subset of individual requirements described in is conflict.

CONSISTENT (CONT')

- Two requirements of the PROFIBUS network are in conflict
  - The PROFIBUS shall have a short reaction time of 60ms
  - PROFIBUS used in hazardous area shall have a low power dissipation of 3.6W

Conflict: Fast PROFIBUS has to have a high power

MAIN TYPES OF CONFLICTS

- ✓ Three main types of conflicts in software requirements
  - Specific characteristics of real-world objects
  - The logical or temporal conflict between two actions
  - Different terms for describing the same real-world object

- Consistency is a challenge since we, at least in the general case, need a complete overview of all requirements that are related to the same event, function, or parameter.

CORRECT

- ✓ Possible incorrect requirements
  - Forward referencing
  - Opacity
  - Noise
  - Etc.

REQUIREMENTS IMPRECISION

- Problems arise when requirements are not precisely stated.
  - Ambiguous requirements may be interpreted in different ways by developers and users.

- For the term ‘search’ in requirement 1
  - User intention – search for a patient name across all appointments in all clinics;
  - Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

FORWARD REFERENCING

- ✓ Requirement items that make use of problem world domain features that are not yet defined.

E, C, and D need to be mapped to a requirement item

FORWARD REFERENCING (CONT')

✓ A requirement of autonomous cruise control (ACC) system is:

ACC system shall maintain the preset speed of an ego-vehicle if there is no forward vehicle.

- Missing inference
  - Who sets the preset speed?
  - What is the value of the preset speed?

OPACITY

- ✓ Requirement items for which rational or dependencies are invisible.

Multiple unrelated concept mapping. A is not related to B

OPACITY (CONT')

- ✓ Requirement:

  Each time the freight train (Freight trains are trains used for carrying goods) doors are closed, the passengers must all seated.

- Opacity
  - There is no visible relationship between the freight trains and passengers.

NOISE

- Requirement items that yield no information on problem world features.

X refers to a concept undefined in the domain

NOISE (CONT')

✓ Requirement: The train system shall guarantee safe transportation of all passengers on their residence.

- Noise
  - A residence is an unknown concept within the train domain
  - The train can only transport passengers to the train station and not to their residence

CHARACTERISTICS OF GOOD SOFTWARE REQUIREMENTS SPECIFICATION*

- ✓ Complete
- ✓ Unambiguous
- ✓ Consistent
- ✓ Correct
- ➢ Verifiable
- ✓ Traceable
- ✓ Ranked for importance and/or stability
- ✓ Modifiable

* IEEE Std 830-1998

TESTABLE REQUIREMENT

- ✓ If a method cannot be devised to determine whether the software meets a particular requirement, then that requirement is not testable.

TESTABILITY CONCERNS

- Testability touches upon two areas of concern
  - How test-friendly is the requirement?
  - How easy is it to test the implementation?

- These two concerns are not independent and need to be considered together

THREE WAYS OF CHECKING GOAL ACHIEVEMENT

- ✓ Executing a test
  - Give input
  - Observe and check the output
  - Involves the computer system and peripherals
- ✓ Run experiments
  - Input / output but also involves the users
- ✓ Inspect the code and other artifacts
  - Evaluation based on documents

WHEN TO USE WHAT

T: Run test on code
E: Experiment
I: Inspect

NTNU

TO MAKE REQUIREMENTS TESTABLE

- ✓ A requirement needs to be stated in a precise way
- ✓ For some requirements, this is in place right from the start

When the ACC system is turned on, the "Active" light on the dashboard shall be turned on.

- ✓ For some other requirements, we need to change it to get a testable version

The system shall be easy to use.

REQUIREMENT TESTABILITY CHECKLIST

- ✓ Modifying phrases
- ✓ Vague words
- ✓ Pronouns with no reference
- ✓ Passive voice
- ✓ Negative requirements
- ✓ Assumptions and comparisons
- ✓ Indefinite pronouns
- ✓ Boundary values
- ✓ …

MODIFYING PHRASES

- The meaning of the following words and phrases are subject to interpretation and make requirement optional:
  - as appropriate
  - if practical
  - as required
  - to the extent necessary / practical

- “Shall be considered" only requires the contractor to think about it
- “And/or” mandatory/optional?
- “Shall/should/may”, should not be used for priority

VAGUE WORDS

- Vague words inject confusion, e.g.,
  - Manage, handle, process, necessary, appropriate
  - Usually, approximately, systematically, quickly
- Use words that express what the system must do
- ***Requirement***: The system shall process ABC data to the extent necessary to store it in an appropriate form for future access.
- ***Correction***: The system shall edit ABC data.

PRONOUNS WITH NO REFERENCE

- ***Requirement***: It shall be displayed.
- When this occurs, the writer is usually relying on a nearby requirement in the requirements document for the meaning of "it."
- As requirements are assigned for implementation, they are often reordered and regrouped, and the defining requirement is no longer nearby.

PASSIVE VOICE

- Requirements should be written in an active voice, which clearly shows X does or provides Y
- **Requirement with passive voice**: Z shall be calculated
- **Correction by using active voice**: The cruise control system shall calculate Z.

NEGATIVE REQUIREMENTS

- ✓ *Everything* outside the system is what the system does not do
- ✓ Testing would have to continue forever to prove that the system does not do something
- ✓ State what the system does

Negative requirement: Dangerous characters should not be used (blacklist).

Correction: Only characters A, B, C are allowed to be used (whitelist).

ASSUMPTIONS AND COMPARISONS

- ✓ **Requirement**: the system shall increase throughput by 15%.
- ✓ Sounds testable, but isn't
- ✓ The assumption is "over current system throughput"
- ✓ By comparing to another system, the meaning of the requirement changes when the other system changes

INDEFINITE PRONOUNS

- ✓ Indefinite pronouns are “stand-in” for unnamed people or things
- ✓ Their meaning is subject to interpretation

- All
- Another
- Any
- Anybody
- Anything
- Each
- Either
- Every
- Everybody
- Everyone
- Everything
- Few
- Many
- Most
- Much

BOUNDARY VALUES

- ✓ Vacation requests of up to 5 days do not require approval. Vacation requests of 5 to 10 days require supervisor approval. Vacation requests of 10 days or longer require management approval.

HOW EASY IS IT TO TEST THE IMPLEMENTATION?

- ✓ Some requirements are more difficult to test than others
- ✓ The volume of tests needed, e.g.,
  - Response time or storage capacity
- ✓ Type of event to be tested, e.g.,
  - Error handling or safety mechanisms
- ✓ The required state of the system before testing, e.g.,
  - A rare failure state or a specific transaction history

HOW EASY IS IT TO TEST THE IMPLEMENTATION? (CONT')

- ✓ Four concerns related to implementation and testability
  - Autonomy of the system under test
  - Observability of the testing progress
  - Re-test efficiency
  - Test restartability

AUTONOMY OF THE SYSTEM UNDER TEST

- How many other systems are needed to test this requirement?
- Requirement: “If the door is closed and a person is detected then send signal Open_Door”
- Can be tested using a simulated actuator
  - Simulate a “person detected” signal on the sensor and check if an Open_Door signal is sent to the actuator
- More difficult/costly to be tested if the actual sensors and actuators must be installed for running tests.

OBSERVABILITY

- ✓ How easy is it to observe the progress of the test execution?

  This is important for tests that do not produce output – e.g., the requirement is only concerned with an internal state change or update of a database.

- ✓ How easy to observe results of the test?

  Important for tests where the output is dependent on an internal state or database content.

RE-TEST EFFICIENCY

- How easy is it to perform the “test – check – change – re-test” cycle?

- This includes
  - Observe the test result and run only failed tests
  - Run only tests that are related to the change

TEST RESTARTABILITY

- How easy is it to
  - Stop the test temporarily
  - Study current state and output
  - Start the test again from the point where it was stopped

INVOLVE TESTERS IN REQUIREMENT PHASE

- ✓ That a requirement is testable does not necessarily mean that it is ***easy*** to test

- ✓ In order to have testable requirements it is important that

  The testers are involved right from the start of the project. It is difficult to add testability later.

- ✓ The tests are integrated parts of the requirement

NON-FUNCTIONAL REQUIREMENTS

"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."

WHAT ARE NON-FUNCTIONAL REQUIREMENTS (NFR)

- Many definitions can be found of NFR
- A NFR is basically an attribute of or a constraint on a system [1]
  - Attributes
    - **-ilities**: understandability, reliability, portability, flexibility, availability, maintainability, scalability, …
    - **-ities**: security, simplicity, clarity, ubiquity, integrity, safety, modularity, …
    - **-ness**: user-friendliness, robustness, timeliness, responsiveness, …
  - Constraint
    - **Physical**
    - **Environmental**
    - **Legal**
    - **Cultural**

NTNU

ISO/IEC 25010 PRODUCT QUALITY MODEL

The ISO model is a factor – criteria – metrics model. Only the first two levels are shown in the diagram.

MAKE RELIABILITY MEASURABLE

The quality concepts
(abstract notions of quality properties)

Measurable Quantities
(define some metrics)

Counts taken from design/execution
(realization of the metrics)

NTNU

RELIABILITY REQUIREMENTS SPECIFICATION

- **Maturity**:
  MTTF (Mean Time To Failure, the expected time to failure for a non-repairable system) > 500 hrs

- ✓ **Availability:**
  The system shall meet or exceed 99.99% uptime

- ✓ **Recoverability:**
  In case of an error, the time needed to get the system up and running again (MTTR) shall not exceed one hour

RELIABILITY TESTS

- ✓ MTTF calculation
  - Assume you tested 3 identical systems starting from time 0 until all of them failed
  - The first system failed at 10 hours, the second failed at 12 hours and the third failed at 13 hours
  - The MTTF is the average of the three failure times, which is (10+12+13)/3 = 11.6667 hours

- How to verify MTTF > 500 hrs?
  - Use 10 PCs. Test for two weeks (40 hr per week) => TTT (Total Test Time) = 800
  - Failed more than once? Satisfied/Not satisfied

TOTAL TEST TIME IN LAB VS. USAGE TIME

- We need to distinguish
  - Total test time (TTT) in the lab
  - Usage time (UT) of the system in real life

- The usage time is related to
  - How often is the system used at the users' site?
  - Number of users each time the system is used

TOTAL TEST TIME IN LAB VS. USAGE TIME (CONT')

A simple example:

- ✓ We have TTT = 800 hrs in the lab.
  - Use 10 PCs. Test for two weeks (40 hrs per week)
- ✓ The system will be used for only 20 hrs per week – e.g., for accounting purposes
- ✓ The 800hrs in the lab. TTT is equivalent to 1600 hrs of usage time
- ✓ Failed twice? Satisfy/Not satisfy MTTF > 500 hrs?

EXERCISE : AVAILABILITY REQUIREMENTS

- If the system shall meet or exceed the following availability, what is the maximum downtime per year to satisfy such availability requirements?

maximum downtime = (total time per year) x (1-availability)

Availability requirement	Maxium downtime per year?
99%	3.65 days/year
99.9%	8.76 hours/year
99.999%	5 minutes/year
99.9999%	31 seconds/ year

PERFORMANCE

Response speed and amount of resources used

- Software products
- Hardware facilities
- Materials (e.g., print paper, diskettes)

Criteria

- **Time behavior:** Response and processing times and throughput rates
- **Resource utilization:** Amount and types of resources used
- **Capability**: Minimum/maximum limits

PERFORMANCE REQUIREMENT EXAMPLES

- ✓ Authorization of an ATM withdrawal request shall take no more than 2.0 seconds.

Response time

- ✓ Webpages shall fully download in an average of 3 seconds or less over 30 megabits/second Internet connection.

Resource utilization

- ✓ The on-line banking systems should support minimum 10.000 credit card transactions proceed per second.

Capability

USABILITY

Help users achieve specified goals with

- Effectiveness
- Efficiency
- Satisfaction

Criteria

- **User interface aesthetics**
- **Easy to operate**
- **Easy to learn**

Usability criteria are subjective

Tests should have a strong component of subjectivism

SETTING REQUIREMENTS – THE MBO METHOD

- ✓ The method used in the example is based on T. Gilb's ideas of MbO – Management by Objectives [5]
- ✓ We start with the requirement – e.g., the system shall be easy to learn
- ✓ We then follow up with "what do you mean by…" until we reach something that is observable and thus testable

REQUIREMENTS – FIRST STEP

- The first step is to apply the MbO for each criterion

- Examples
  - Learn its application
    What do we mean by “learn application.”

  - Is the software product liked by the user
    What do you mean by “beautiful / pleasing / like”?

REQUIREMENTS – SECOND STEP

- ✓ Learn application: After a two weeks course and after two weeks use of the system, a normal user can solve a set of standardized problems

- ✓ Pleasing/like: Score high on a likeability scale – e.g., 90 % score 7 or higher on a scale from 1 to 10 – after a two weeks course and two weeks of real use

We need to develop a course, a set of standardized problems, and a likeability questionnaire.

USER INVOLVEMENT IN SETTING REQUIREMENTS

When we use MbO or other related techniques for setting requirements, we will in most cases have a situation where:

- ✓ The user will have to participate in the tests
- ✓ There will be a strong link between requirement and test
- ✓ In many cases, ***passing the test by the user will be the requirement***

NFR REQUIREMENTS – TO REMEMBER

- ✓ Some NFR requirements are conflict with each other
- ✓ Trade-off of NFR requirements will impact system design and implementation

CHARACTERISTICS OF GOOD SOFTWARE REQUIREMENTS SPECIFICATION*

- ✓ Complete
- ✓ Unambiguous
- ✓ Consistent
- ✓ Correct
- ✓ Verifiable
- ➢ Ranked for importance and/or stability
- ➢ Traceable
- ➢ Modifiable

* IEEE Std 830-1998

RANKED FOR IMPORTANCE AND/OR STABILITY

- ✓ If, and only if, each requirement in it has an identifier to indicate either the importance or stability of that particular requirement.

- ✓ Rank and distinguish classes of requirements
  - Essential
  - Conditional
  - Optional

REQUIREMENT TRACEBILITY

"The client kept changing the requirements on a daily basis, so we decided to freeze them until the next release."

REQUIREMENTS TRACEABILITY

"Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction, i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases."

Gotel and Finkelstein

BIDIRECTIONAL TRACEABILITY

PURPOSES OF REQUIREMENT TRACEABILITY

- ✓ Forward traceability
  - Ensure all requirements are implemented
  - Change impact analysis

- ✓ Backward traceability
  - Avoid “gold plating”
  - Change impact analysis
  - Defect impact analysis
  - Root cause analysis of defects

HOW TO IMPLEMENT TRACEABILITY?

- ✓ Traceability matrix

Project Name

Unique No.	Requirement	Source of Requirement	Software Reqs. Spec / Functional Req. Doc.	Design Spec.	Program Module	Test Case(s)	Successful Test Verification	Modification of Requirement	Remarks

EXAMPLE OF TRACEABILITY MATRIX

Requirement Source	Product Requirements	HLD Section #	LLD Section #	Code Unit	UTS Case #	STS Case #	User Manual
Business Rule #1	R00120 Credit Card Types	4.1 Parse Mag Strip	4.1.1 Read Card Type	Read_Card _Type.c Read_Card _Type.h	UT 4.1.032 UT 4.1.033 UT 4.1.038 UT 4.1.043	ST 120.020 ST 120.021 ST 120.022	Section 12
			4.1.2 Verify Card Type	Ver_Card _Type.c Ver_Card _Type.h Ver_Card _Types. dat	UT 4.2.012 UT 4.2.013 UT 4.2.016 UT 4.2.031 UT 4.2.045	ST 120.035 ST 120.036 ST 120.037 ST 120.037	Section 12
Use Case #132 step 6	R00230 Read Gas Flow	7.2.2 Gas Flow Meter Interface	7.2.2 Read Gas Flow Indicator	Read_Gas _Flow.c	UT 7.2.043 UT 7.2.044	ST 230.002 ST 230.003	Section 21.1.2
	R00231 Calculate Gas Price	7.3 Calculate Gas price	7.3 Calculate Gas price	Cal_Gas_ Price.c	UT 7.3.005 UT 7.3.006 UT 7.3.007	ST 231.001 ST 231.002 ST 231.003	Section 21.1.3

HOW TO IMPLEMENT TRACEABILITY? (CONT')

- ✓ Trace tagging

SRS

R00104 The system shall c
time prior to the actual dispe
requests cancellation.

SDS

SDS Identifier	SRS Tag	Component Name
7.01.032	R00104	Cancel_ Transaction

UTS

Test Case #	SDS Identifier	Test C
23476	7.01.032	Cancel_Befor
23477	7.01.032	Cancel_After_
23478	7.01.032	Cancel_After_

CHARACTERISTICS OF GOOD SOFTWARE REQUIREMENTS SPECIFICATION*

- ✓ Complete
- ✓ Unambiguous
- ✓ Consistent
- ✓ Correct
- ✓ Verifiable
- ✓ Ranked for importance and/or stability
- ✓ Traceable
- ➢ Modifiable

* IEEE Std 830-1998

MODIFIABLE

- If, and only if, a requirement structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style.
- Redundancy can easily lead to low modifiability
- Boilerplate can help improve modifiability
  - E.g., “will” -> “shall”

Course summary

Jingyue Li (Bill)

Exam

Vurderingsordning: Mappeevaluering

Termin	Statuskode	Vurdering	Vekting	Hjelpemidler	Dato	Tid	Eksamens-system
Vår	ORD	Arbeider	50/100				INSPERA
Vår	ORD	Hjemme-eksamen (1)	50/100		Utlevering 03.06.2022 09:00 Innlevering 03.06.2022 13:00		INSPERA
Sommer	UTS	Arbeider	50/100				INSPERA
Sommer	UTS	Hjemme-eksamen	50/100				INSPERA

Letter grade A-F

A: 89–100 points
B: 77–88 points
C: 65–76 points
D: 53–64 points
E: 41–52 points
F: 0–40 points

Evaluation and grading

- Exercises: 50%
- Exam: 50%
- You have to pass both!
  - You have to hand in **all** the exercises **and** get a passing grade to be allowed to take the exam
  - If you fail the exam, you will fail the course

Exercises	Exam	Final grade
Fail	Fail	Fail
Fail	Pass	Fail
Pass	Fail	Fail
Pass	Pass	Combine exercise and exam grades

Couse modules

- Requirement module
- Testing module
- Code review and refactoring
- Chosen topics
  - DevOps
  - Technical debt
  - Cost and effort estimation
  - Defect estimation
  - Software ecosystem
  - Software startup
- Exercises 1 - 3

Curriculum

- Journal and conference articles (uploaded to blackboard)
- Some chapters from books (accessible in NTNU library)

(1st/2nd edition)

Requirement module

Description	Students should be able to	To read
Requirement elicitation	• Apply goal oriented RE approach	• Ian Sommerville (2016), Software Engineering (10th ed.), ISBN 978- 0133943030, chapter 4 • Paper 1: Goal-oriented RE A guided tour
Requirement quality	• Identify quality issues of requirements and fix them • Define different function and non-functional requirements	• Paper 1: IEEE Recommended Practice for Software Requirements Specifications Std 830-1998 • Paper 2: Glinz, M. (2007). On Non-Functional Requirements. 15th IEEE International Requirements Engineering Conference (RE 2007), 21–26. • Paper 3: Ontology-Driven Guidance for Requirements Elicitation • Paper 4: Bidirectional requirement traceability

Testing module - 1

Description	Students should be able to	To read
Control flow and data flow testing	• Explain code, decision, path, and output coverages • Create test cases by using different data flow testing strategies • Explain how to use the test coverage information for different purposes	Software testing and quality assurance book, chapter 4 and 5
Domain and function testing	• Apply domain testing approach to generate test cases of single variable and multiple variables in combination • Explain risk-based testing	Software testing and quality assurance book, chapter 6 and 9

Testing module - 2

Description	Students should be able to	To read
Integration and system testing	• Explain different approaches for creating integration test cases and their pros and cons • Create different system test cases • Explain different categories of acceptance test cases • Explain different test prioritization approaches	Software testing and quality assurance book, chapter 7, 8, and 14
Regression testing	• Apply graph-walk safe regression test selection approach • Explain firewall regression test selection approach • Explain different regression test minimization and prioritization strategies	Paper 1: Regression testing minimization, selection, and prioritization: a survey

Testing module - 3

Description	Students should be able to	To read
Test in practice	• Explain concepts and strategies related to test planning and execution. • Create a test plan.	Software testing and quality assurance book, chapter 12 and 13

Code review and refactoring

Description	Students should be able to	To read
Code review and code refactoring	• Explain why code inspection and testing complement each other • Explain unsystematic vs. systematic reading techniques • Apply check-list based, defect-based, and perspective-based reading techniques • Explain the purpose and steps of code refactoring • Apply code refactoring methods to identify bad code smells in code (Python) and refactor them	Software reading techniques, chapter 3 and 4 Refactoring book: Improving the Design of Existing Code, chapter 3 (Bad smells in code) Paper 1: Modern code review a case study at google Paper 2: Expectations, Outcomes, and Challenges of Modern Code Review

Chosen topics - 1

Description	Students should be able to	To read
DevOps	• Explain the motivation for using DevOps • Explain the basic concepts of DevOps	• Slides only
Technical debt	• Explain the motivation and approaches to measure technical debt	• Slides only
Cost&Effort and defect estimation	• Explain importance and challenges in estimating cost and software defects • Describe different estimation techniques, i.e., expert judgement, estimation by analogy, Lines of code, and COCOMO • Describe machine learning techniques for software defect predictions	• Slides only

Chosen topics - 2

Description	Students should be able to	To read
Software ecosystem	• Explain definitions, key elements and examples of software ecosystems • Describe characteristics of software ecosystems: complexity, productivity, robustness, healthiness and coopetition	• Slides only
Software startup	• Explain definitions, key elements and examples of software startups • Describe concept of Technical Debt in Software startups • Explain logic of effectuation and causation • Describe concept and usages of Minimum Viable Product in Software startups	• Slides only

Structure of the exam

- Two case studies (10 points each, 20 points in total)
- Five open-ended questions (3 points each, 15 points in total)
- 15 close-ended (Multiple-choice) questions (1 point each, 15 points in total)

A case study example

- Company A is going to pay Company B for developing Autonomous Truck Platooning. Truck platooning (as shown in the above picture) is the linking of two or more trucks in convoy, using connectivity technology and automated driving support systems. These vehicles automatically maintain a set, close distance between each other when they are connected for certain parts of a journey, for instance, on motorways.

- The requirements Company B got from Company A are as follows.

We are going to develop Autonomous Truck Platooning. We want some numbers of trucks can drive autonomously, and other trucks are driven by humans. The autonomous trucks should drive at a default speed. We hope the truck platooning can drive along the route we set in GPS before the journey and follow the human drivers' command to change the route. We hope the autonomous trucks should have three chairs for the drivers. We want the truck to drive safely. So the trucks should have sensors to avoid collisions with obstacles because there are always other vehicles or obstacles in the motorway. When there are obstacles on the road, the trucks should stop. We also want the trucks to drive efficiently, which means that the trucks should minimize the distance between them and maximize the speed of the whole Platooning. The trucks should use sensors to detect the distances between trucks and the speed of the trucks.

A case study example (cont')

- Task: Identify quality issues in these requirements according to the requirements quality metrics: *ambiguity*, *inconsistency*, *forward referencing*, and *opacity*. If one requirement has several quality issues, list all of them. Then, try to fix the requirements quality issues of each requirement and write down the improved requirements.
- Example answers
- Ambiguity (1 point): So the trucks should have sensors to avoid collisions with obstacles because there are always other vehicles or obstacles in the motorway (Not explain obstacles).
  Your fix: Only need to make a short list of obstacles. It does not need to be a complete list.
- Inconsistency (1 point): E.g., We hope the truck platooning can drive along the route we set in GPS before the journey and follow the human drivers' command to change the route (Conflict on who can make the control decision).
  Your fix: Only need to eliminate the inconsistency.

An open-ended question example

Question: Explain what static backward slicing is and how to create backward slicing using data flow information.

Answers:

- The purpose of code slicing is to choose only a subset of code that is relevant to a particular variable at a certain point to minimize the code for more cost-efficient testing and debugging.

- Static backward slicing: A slice with respect to a variable *v* at a certain point *p* in the program is the set of statements that contributes to the value of the variable v at p.

- To establish static backward slicing, all Defs and All P-uses of a variable v before a certain point p are chosen.

Summary

- Use the slides as the table of content to read books and papers
- The content of the course, especially the chosen topics, varies each year. Not all questions from previous years are relevant to this year
- The case studies and open-ended questions in previous years' exams can be used for practice
- We will create a discussion channel in Blackboard for Q&A exam related questions

Where does code go when it is finished?

- A story about DevOps

About Me

- Platform Engineer at Vipps
  - Develop login-systems
- Very interested in software security
- **Twitter:** noratomas3
- **Linkedin:** Nora Tomas

NULL STRESS
VI TAR DET PÅ
vipps

About Me

- Studied Computer Science at Gløs
- Failed “a few” subjects in my time at NTNU
  - So don’t worry, you will be fine! :)

After this lecture I hope you will

- Know what a frontend and backend is
- Get a sense of what an API is
- Understand what deployment means and why it is important
- Know what DevOps means
- See (in the demo) what a DevOps engineer does
  - github.com/NoraTomas/DevOpsDemo

“Hello World”

BeginnersBook [~/PycharmProjects/BeginnersBook] - .../HelloWorld.py [Beg...

BeginnersBook 〉 HelloWorld.py 〉 HelloWorld

Pr...

HelloWorld.py

BeginnersBook

venv

HelloWorld.py

```
1  # This prints Hello World on the output screen
2  print('Hello World')
```

Run HelloWorld

```
/Users/chaitanyasingh/PycharmProjects/BeginnersBook/venv/bin/python /Us
Hello World

Process finished with exit code 0
```

1:47 n/a UTF-8

“Hello World”

- “Coding is not difficult” -> https://www.youtube.com/watch?v=hb7Q33ysCwI

“Hello World”

Time for a quick Kahoot!

Customer-Driven project - Localhost

Introducing Deployment

- “Software deployment is all of the activities that make a software system available for use”

Failed IT-projects - Forgotten Deployment

Forgetting deployment = forgetting to build the building

- Source: State of DevOps 2019 report
- Download here: https://cloud.google.com/devops/state-of-devops/

Agile Alignment Trap

- Companies get into the agile alignment trap

DevOps Engineer

- A lot of job postings for “DevOps Engineer”
- What does it really mean?

www.finn.no › Jobb › Ledige stillinger ▾ Translate this page

Ledige stillinger med søkeordet "devops" | FINN Jobb - FINN.no

Du finner 291 ledige stillinger med søkeordet "**devops**" på **FINN Jobb**. Søk blant alle typer **jobber** i hele **Norge**!

What is DevOps

To understand what DevOps engineer does...
Let us create and deploy an API!

Example: What is an API?

- API is a term used very loosely

Primary Secondary Success Danger Warning Info Light Dark Link

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>

<button type="button" class="btn btn-link">Link</button>
```

Example: What is an API?

- Most commonly, when people say API, they mean: HTTP endpoints (URLs) that you can use to retrieve information
- Example: https://openweathermap.org/current

API call

```
api.openweathermap.org/data/2.5/weather?q={city name}&appid=
{API key}
```

Let us make our own Weather API

- We use Java and Spring Boot
- What we will do
  - Show what a backend API looks like
  - Build a JAR file locally
  - Deploy the JAR file

Opening our API to the world

- Could do it from our own computer!
  - Need
    - Public IP
    - Open firewall in router
    - Make an “inbound connection” to port 8080
- Drawbacks from using our own computer
  - Your computer might become slow (with many connections)
  - A lot of other software taking space
  - Exposing your public IP to the world
  - You have to buy a new computer if you want more power
- A server is just a computer

SERVER :)

Let us deploy to EC2!

- EC2 is a virtual machine in AWS
- We can use that virtual machine - just like any other computer
- We just don’t own the hardware

We made a backend!

- Why is it a backend?
  - The Java-application is not supposed to run on users computers
  - End-users will not “download” this Jar-file and run it on their computers
  - We know what machine the Jar-file will run on, because we created it!
  - ***Front-end development:*** Making sure things work on different clients

Our Python application - was a frontend!

BeginnersBook [~/PycharmProjects/BeginnersBook] - .../HelloWorld.py [Beg...

BeginnersBook 〉 HelloWorld.py 〉 HelloWorld

Pr...

HelloWorld.py

BeginnersBook
venv
HelloWorld.py

```
1  # This prints Hello World on the output screen
2  print('Hello World')
```

Run HelloWorld

```
/Users/chaitanyasingh/PycharmProjects/BeginnersBook/venv/bin/python /Us
Hello World

Process finished with exit code 0
```

1:47 n/a UTF-8

Manual Deployment

- In the “old days” people did deployment like we do now
  - Developers would give operations a “manual” on what was required
- Many things can go wrong!
  - Operations installs the wrong version of Java on machines
  - Dependencies might not work well together
  - Operations can’t install what you need that easily (database software)
- Introducing Operations Engineer

Classic Failure - Dependencies

Before we build pipeline - quick recap!

- A frontend runs on the users computer
  - This is why frontend is more than just design, or HTML, or CSS
  - A console application (such as “Hello World” in Python) is also a type of frontend
- A backend runs on a computer that does not belong to the user. Usually it runs on a virtual machine, such as EC2 in AWS
- Backend and frontend communicate via the HTTP protocol
  - Also often called “An API”, but the term API is used very loosely and can mean almost anything
- A frontend and backend need to be deployed so that end users can interact with them
- The deployment process is very important and should not be left for last
- In our case, we deployed our backend by uploading the JAR-file to an EC2. We did this upload manually.
- Now, we want to automate the upload. This is where the pipeline comes in!

Pipelines - Automating Deployment

- A pipeline is a mechanism to move your “code” from a GitHub-repo into production
- Behind the scenes, a pipeline is just a computer that runs commands

Testing

- Testing is important because of automation
- Tests can be added to the pipeline
- Creating tests is like creating your first user

```
Run Test | Debug Test
class TestIncrementDecrement(object):
    Run Test | Debug Test
    def test_increment():
        assert increment(3) == 4

    Run Test | Debug Test
    def test_decrement():
        assert decrement(3) == 2
```

Thank you! :)

Control flow and data flow testing

Jingyue Li (Bill)

Outline

- ➢ Control flow testing and coverage
- Data flow testing and coverage
- Mutation testing and coverage

Industry standards define test needed

- IEEE 1012 – general software verification and validation
- IEC 61508 – general safety critical software
- ISO 26262 – automotive software
- Other domain specific standards, e.g.,
  - Healthcare
  - Oil & Gas
  - Railway

ISO 26262 – automotive software

The ASIL (Automotive System Integrity Level) – A, B, C, or D – is the outcome of the combination of three factors

- S – Severity. How dangerous is the event
- E – Probability. How likely is the event
- C – Controllability. How easy is it to control the event if it occurs

Finding the ASIL level

Severity	Probability	C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

ASIL chart*

ASIL chart

Functional category	Hazard	ASIL-A	ASIL-B	ASIL-C	ASIL-D
Driving	Sudden start				
	Abrupt acceleration				
	Loss of driving power				
Braking	Maximum 4 wheel braking				
	Loss of braking function				
Steering	Self steering				
	Steering lock				
	Loss of assistance				

* https://www.jnovel.co.jp/en/service/compiler/iso26262.html

Methods and measurements for software unit testing

Methods and measures	Sub-methods/Measures	ASIL			
		A	B	C	D
Functional tests	…	…			
Structural coverage	Statement coverage Decision coverage …	++ +	++ +	+ ++	+ ++
Resource usage measurement	…	+	+	+	++
Back-to-back test between simulation	…	+	+	+	++

Control flow coverage

- Statement coverage
- Decision coverage
- Path coverage

Statement coverage report

Coverage Report

http://www.cafeconlech

```
110 128            else if ( nav.isElement( first ) )
111                {
112 100                return nav.getElementQName( first );
113                }
114  28            else if ( nav.isAttribute( first ) )
115                {
116   0                    return nav.getAttributeQName( first );
117                }
118  28            else if ( nav.isProcessingInstruction( first ) )
119                {
120   0                    return nav.getProcessingInstructionTarget( first );
121                }
122  28            else if ( nav.isNamespace( first ) )
123                {
124   0                    return nav.getNamespacePrefix( first );
125                }
126  28            else if ( nav.isDocument( first ) )
127                {
128  28                return "";
129                }
130   0              else if ( nav.isComment( first ) )
131                {
132   0                    return "";
133                }
134   0              else if ( nav.isText( first ) )
135                {
136   0                    return "";
137                }
138                else {
139   0                    throw new FunctionCallException("The argument to the name
140                }
141            }
142
143   8        return "";
144
```

Done

Disabled

You can get test coverage report with [GitLab CI]

Coverage report

Coverage Report

http://www.cafeconlech

Packages

All
org.jaxen
org.jaxen.dom
org.jaxen.dom.html

All Packages

Classes

BaseXPath *(77%)*
Context *(93%)*
ContextSupport *(91*
DefaultNavigator *(3*
DocumentNavigator
DOMXPath *(100%)*
DocumentNavigator
HTMLXPath *(0%)*
NamespaceNode *(*
DocumentNavigator
Dom4jXPath *(100%*

Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	205	69%	80%	2.811
org.jaxen	24	77%	73%	1.38
org.jaxen.dom	3	55%	60%	1.907
org.jaxen.dom.html	2	0%	0%	1.364
org.jaxen.dom4j	2	78%	85%	2.395
org.jaxen.expr	73	73%	84%	1.566
org.jaxen.expr.iter	14	98%	100%	1.029
org.jaxen.function	27	64%	76%	5.373
org.jaxen.function.ext	6	63%	72%	4.235
org.jaxen.function.xslt	1	86%	100%	2.5
org.jaxen.javabean	4	44%	72%	1.87
org.jaxen.jdom	3	62%	63%	2.897
org.jaxen.pattern	13	49%	52%	2.135
org.jaxen.saxpath	8	51%	81%	1.887
org.jaxen.saxpath.base	6	95%	100%	10.723
org.jaxen.saxpath.helpers	2	28%	83%	1.34
org.jaxen.util	15	41%	50%	2.432
org.jaxen.xom	2	71%	66%	1.783

Reports generated by Cobertura.

Done

Disabled

Test coverage at Google

- Per-project statement coverage
- 1.5 years worth of data across hundreds of projects

Probably by Apr. 2015

The median is 78%, the 75th percentile is 85% and 90th is percentile 90%.

Decision coverage

- Each one of the possible branch (true/false or switch-case) from each **decision** point is executed at least once

```
*   Read P
    Read Q

    If P+Q > 100 THEN
        Print "Large"
    ENDIF

    IF P > 50
       Print    " P Large"
    ENDIF
```

Statement coverage:
A, C, D, E, G, H

Decision coverage:
A, C, D, E, G, H
+
A, B, E, F, H

Branch coverage

- Count branches covered
- Slightly different from decision coverage

If 3 out of 4 branches of a switch statement are executed

- Branch coverage 75%
- Decision coverage 0% (a decision is considered covered only if all its branches are covered)

Path coverage

- All possible **paths** of the software should be tested

Decision coverage:
A, C, D, E, G, H
+
A, B, E, F, H

Path coverage:
A, C, D, E, G, H
+
A, B, E, F, H
+
A, B, E, G, H
+
A, C, D, E, F, H

White-box testing methods

- Usually need to generate control/data flow graph first using some tools, e.g., Java parser (Front End)*
- Then, use different test inputs to achieve different coverages
- Path coverage is more difficult to achieve
  - How many test cases will be sufficient?
    - McCabe's cyclomatic complexity can help
  - How to systematically create test cases?
    - Decision table can help

* http://www.semanticdesigns.com/Products/FrontEnds/JavaFrontEnd.html

McCabe's cyclomatic complexity

- From control flow graph of the program
- $v(G) = E - N + 2P$
- $v(G)$ = cyclomatic complexity
- E = the number of edges of the graph
- N = the number of nodes of the graph
- P = the number of connected components

Graph example

- $v(G) = E - N + 2P$
- E: edge, N:node, P: connected component

- E = 9
- N = 8
- P = 1

- V(G) = 9 - 8 + 2 x 1 = 3

Using v(G) to count number of paths

- The minimum number of paths through the code is v(G)

- As long as the code graph is a DAG – Directed Acyclic Graph – the maximum number of paths is 2**|{predicates}| (when all predicates have only two possible values)

- Thus, we have that

  V(G) ≤ number of paths ≤ 2**|{predicates}|

Simple case - 1

S1;
IF P1 THEN S2 ELSE S3
S4;

E = 4, N = 4, P = 1, E-N+2P = 2

- V(G) = 2,
- One predicate (P1)
- Two test cases can cover all paths

Simple case – 2

S1;
IF P1 THEN X := a/c ELSE S3;
S4;

- V(G) = 2,
- One predicate (P1)
- Two test cases will cover all paths but not all cases. What about the case c = 0?

Nested decisions

```
S1;
IF P1 THEN S2 ELSE
    S3;
    IF P2 THEN S4 ELSE S5
FI
S6;
```

v(G) = 3, while Max = 4.
Three test case will cover all paths.

NTNU

Problem – the loop

```
S1;
DO
    IF P1 THEN S2 ELSE S3;
    S4
OD UNTIL P2
S5;
```

Not a Directed Acyclic Graph
v(G) = 3 and Max
is 4 but there is an “infinite”
number of paths

Problem – the loop (cont')

- Loops are the great problem in white box testing
- It is common practice to test the system going through each loop
  - 0 times – loop code never executed
  - 1 time – loop code executed once
  - 5 times – loop code executed several times
  - 20 times – loop code executed "many" times

Use decision table to generate test cases for path coverage

- Make a table of all predicates
- Insert all combinations of True / False/Switch – 1 / 0/many – for each predicate
- Construct a test for each combination

- A general technique used to achieve full path coverage
- However, in many cases, it may lead to over-testing

Using a decision table

P1	P2	P3	Test description or reference
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Decision table example – binary

P1	P2	Test description or reference
0	0	S1, S3, S5, S6
0	1	S1, S3, S4, S6
1	0	S1, S2, S6
1	1	S1, S2, S6

Go to **www.menti.com** and use the code **5729 3680**

Exercise

Outline

- Control flow testing and coverage
- ➢ Data flow testing and coverage
- Mutation testing and coverage

Data flow testing

- Data flow testing uses the control flow graph to explore the unreasonable things that can happen to data (data flow anomalies)

- Data flow anomalies are detected based on the associations between definition, initialization, and use of variables, e.g.,
  - Variables are used without being initialized

Variable lifecycle

- Variables have a life cycle
  - Defined and initialized
    - A value is bound to the variable
    - E.g., x = …
  - Used
    - The value of the variable is referred
    - Predicate use (p-use), E.g., if (x>10)
    - Computational use (c-use). E.g., y = x + 1;
  - Killed (destroyed)

Variable lifecycle (cont')

```
{                       // begin outer block
    int x = 1;          // x is defined as an integer within this outer block
        ...;            // x can be accessed here
    {                   // begin inner block
     int y = 2;         // y is defined within this inner block
        ...;            // both x and y can be accessed here
     }                  // y is automatically destroyed at the end of this block
    ...;                // x can still be accessed, but y is gone
}                       // x is automatically destroyed
```


NTNU

Data flow anomalies and normal use

- **dd: define and then define again – suspicious**
- **dk: define and then kill – potential bug**
- **ku: kill and then used – serious defect**
- **kk: kill and then kill again – potential bug**
- du: define and then use – OK
- kd: kill and then redefine – OK
- ud: use and then redefine – OK
- uk: use and then kill – OK
- uu: use and then use – OK
- …

Why data-flow testing?

- To uncover possible bugs in data usage during the execution of the code
- Test cases are created to trace every definition to each of its use, and every use is traced to each of its definition
- Various coverage strategies are employed for the creating of the test cases

Data-flow test strategies

- All definitions (AD): every definition of every variable should be covered by at least one use (c-use or p-use) of that variable
- All computational uses (ACU): for every variable, there is a path from every definition to every c-use of that definition
- All predicate-uses (APU): for every variable, there is a path from every definition to every p-use of that definition
- All c-uses/some p-uses (ACU+P): for every variable and every definition of that variable, at least one path from the definition to every c-use should be included. If there are definitions of the variable with no c-use following it, then add p-use test cases to cover every definition

Data-flow test strategies (cont')

- All p-use/some c-use (APU+C): for every variable and every definition of that variable, at least one path from the definition to every p-use should be included. If there are definitions of the variable with no p-use following it, then add c-use test cases to cover every definition
- All uses (AU): for every variable, there is a path from every definition to every use of that definition, i.e., APU + ACU
- All du paths (ADUP): test cases cover every simple sub-path from each variable definition to every p-use and c-use of that variable
- Note that the "kill" usage is not included in any of the test strategies

All definitions

- All definitions (AD): every definition of every variable should be covered by at least one use (c-use/p-use) of that variable.

*

AD

- (1, 2, x)
- (3, 8, a)
- (7, 8, a)
- (6, 5, x)

Test cases for AD coverage

x = 0, a = any (1, 2, 4, 5, 6, 5, 7, 8)

x = 1, a = any (1, 2, 3, 4, 8)

*https://www.cs.ccu.edu.tw/~naiwei/cs5812/st5.pdf

All computational uses

- All computational uses (ACU): for every variable, there is a path from every definition to every c-use of that definition

ACU

- (1, 3, x)
- (1, 6, x)
- (1, 7, x)
- (6, 6, x)
- (6, 7, x)
- (3, 8, a)
- (7, 8, a)

All predicate-uses

- All predicate-uses (APU): for every variable, there is a path from every definition to every p-use of that definition

APU

- (1, 2, x)
- (1, 4, x)
- (1, 5, x)
- (6, 5, x)

All c-uses/some p-uses

- All c-uses/some p-uses (ACU+P): for every variable and every definition of that variable, at least one path from the definition to every c-use should be included. If there are definitions of the variable with no c-use following it, then add p-use test cases to cover every definition

ACU

- (1, 3, x)
- (1, 6, x)
- (1, 7, x)
- (6, 6, x)
- (6, 7, x)
- (3, 8, a)
- (7, 8, a),

ACU + P

- (1, 3, x)
- (1, 6, x)
- (1, 7, x)
- (6, 6, x)
- (6, 7, x)
- (3, 8, a)
- (7, 8, a)

All p-use/some c-use

- All p-use/some c-use (APU+C): for every variable and every definition of that variable, at least one path from the definition to every p-use should be included. If there are definitions of the variable with no p-use following it, then add c-use test cases to cover every definition

APU
- (1, 2, x)
- (1, 4, x)
- (1, 5, x)
- (6, 5, x)

VS.

APU + C
- (1, 2, x)
- (1, 4, x)
- (1, 5, x)
- (6, 5, x)

- (3, 8, a)
- (7, 8, a)

All uses

- All uses (AU): for every variable, there is a path from every definition to every use of that definition, i.e., APU + ACU

AU

- (1, 2, x)
- (1, 4, x)
- (1, 5, x)
- (6, 5, x)

APU

- (1, 3, x)
- (1, 6, x)
- (1, 7, x)
- (6, 6, x)
- (6, 7, x)

- (3, 8, a)
- (7, 8, a)

ACU

All du paths

- All du paths (ADUP): More demanding than All uses (AU). If there are multiple paths between a given definition and a use, they must all be included
- However, ADUP includes loop just once

- In AU, we include path (3, 8, a), meaning either (3, 4, 5, 7, 8) OR (3, 4, 8) OR (3, 4, 5, 6, 5, 7, 8) is ok
- In ADUP, since there are multiple paths from 3 to 8, we need to include (3, 4, 5, 7, 8) AND (3, 4, 8) AND 3, 4, 5, 6, 5, 7, 8)

Go to **www.menti.com** and use the code **5729 3680**

Exercise

Relationship between strategies

Why do we measure test coverage?

- One way of using statement/branch/path/dataflow coverage is to use it as a test acceptance criteria
  - Run test cases
  - Have we reached, e.g., 85% statement coverage
    - Yes – stop testing
    - No – coverage measure will help us directly identify untested code and indirectly help develop/select new test cases

Use test coverage to estimate defect coverage

- Defect coverage: the fraction of actual defects initially present that would be detected by a given test set

Enumerable ***i***

- Blocks/statements
- Branches
- C-uses
- P-uses

Logarithmic-exponential relationship between test coverage and defect coverage

Use coverage for program slicing

- Static backward slicing
  - A slice with respect to a variable *v* at a certain point *p* in the program is the set of statements that **contributes** to the value of the variable v at p
  - *S(v, p)* denotes the set of nodes in the control flow graph that contributes to the value of the variable v at point p

Static backward slicing

S(x, 1) = {1}

S(x, 2) = {1}

S(x, 3) = {1, 2}

S(x, 4) = {1, 2}

S(x, 5) = {1, 2, 4}

S(x, 6) = {1, 2, 4, 5, 6}

S(x, 7) = {1, 2, 4, 5, 6}

All Defs and All P-uses of a variable v before a certain point p are used for slicing*

* http://www.cs.cmu.edu/afs/cs/academic/class/17654-f01/www/refs/Binkley.pdf

Program slicing can help many activities

- Facilitate debugging (Finding and localizing errors)
- Facilitate programming understanding

S(x, 6) = {1, 2, 4, 5, 6}

Outline

- Control flow testing and coverage
- Data flow testing and coverage
- ➢ Mutation testing and coverage

Mutation testing and coverage

- The tester can also measure coverage of seeded faults and use it as an indicator to measure whether the test set is adequate
  - Seed faults in the code
  - Have we found all the seeded faults
    - Yes – test set is adequate
    - No – One or more seeded faults are not found. This tells us which locations and types of defects have not been covered by test cases sufficiently. Then we need to develop/select more test cases.

Fault seeding and estimation

- We assume
  $N_0 / N = S_0 / S$
- Thus
  $N_0 = N * S_0 / S$,

- N_0: number of faults in the code
- N: number of faults found using a specified test set

- S_0: number of seeded faults
- S: number of seeded faults found using a specified test set

Typical mutants

- Where and how to seed the faults
  - Save and seed faults identified during earlier project activities
  - Draw faults to seed from an experience database containing typical faults
    - Arithmetic operator replacement
    - Logical connector replacement
    - Relational operator replacement
    - Absolute value insertion, etc.
- More info. can be found at
  - http://www.uio.no/studier/emner/matnat/ifi/INF4290/v10/undervisningsmateriale/INF4290-Mutest.pdf
  - https://www.softwaretestinghelp.com/what-is-mutation-testing/
  - https://pitest.org/java_mutation_testing_systems/

Code to be tested

- This is what the code might look like:

  1) *Read Age*
  2) *If age>14*
  3) *Doctor= General Physician()*
  4) *End if*

Seed faults (mutants)

- Mutation type 1: Relational operator replacement

  Mutant #1:

  1) Read Age
  2) If age<14 'Changing the > with <'
  3) Doctor= General Physician()
  4) End if

  Mutant #2:

  1) Read Age
  2) If age=14 'Changing the > with ='
  3) Doctor= General Physician()
  4) End if

Mutation coverage

- If our test inputs are 14 and 15. Which of these mutants will be killed?

Test inputs	Expected result	Mutant 1 Changing > with <	Mutant 2 Changing > with =
14	GP not assigned	Succeeds - GP not assigned	Fails - GP assigned
15	GP assigned	Fails - GP not assigned	Fails - GP not assigned

- 14 finds a failure when runs again mutant 2, not effective against mutant 1
- 15 finds failures when runs again mutants 1 and 2

Summary

- We studied
  - Control flow test and related coverage measurement
  - Data flow test and related coverage measurement
  - Mutation test and coverage measurement

Next lecture

- Black-box unit testing

Domain and function testing

Methods and measures for functional software integration testing

Methods and measures	Criteria	ASIL			
		A	B	C	D
Requirements based test	Test case have to be derived from the requirements. All SW component requirements are covered	++	++	++	++
External interface test	Test cases can be determined with the help of equivalence classes and input partitioning of the external interface. This can be completed by boundary value analysis. Boundary values for data types as well as plausible ranges of values for interface variables have to be considered.	+	++	++	++
Fault injection test	…	+	+	++	++
Error guessing test	…	+	+	++	++

Outline

- ➢ Single variable
- Multiple variables
- Error guessing/Risk based
- Output coverage and testing

The challenge

The total number of possible combinations of X and Y is $2^{32} \times 2^{32}$

Combinatorial explosion!

- We need to systematically reduce the set of all possible values to few manageable subsets

Partitioning of domain

- Identifying domain first
  - Linear (variables whose values can be mapped onto a number line, e.g., integer variables)
  - Non-linear (e.g., variables with enumeration values)

- Ideally, partitioning of input & output domain should result in non-overlapping or disjoint partitions

Black-box domain testing approaches

- Identify input and output domains

- Generate test case to cover single variables in the domains and combination of the variables
  - Equivalence classes, boundary values
  - Decision tables, pairwise testing
  - Error guessing/risk-based

Partitioning of domain of single variable

- Variable types
  - Linear domain variable (e.g., integer)
  - Linear domain variable with multiple ranges
  - String variable
  - Enumerated variable
  - Multidimensional variable

Partitioning domain of single variable

- Step 1: Identify the variable
- Step 2: Determine the domain
  - All possible values of the variable
- Step 3: Identify risks
- Step 4: Partition the domain into equivalence classes based on the identified risks

Linear domain variable

- Software application example

  BankA issues Visa credit cards with credit limits in the range of $4000 to $40000. A customer is not to be approved for credit limits outside this range.

  A customer can apply for the card using an online application form in which one of the fields requires that the customer type in his/her desired credit limit.

Linear domain variable (cont')

- Identify the variable: "credit-limit"
- Determine the input domain: $4000 =< credit-limit <= $40000
- Identify risks
  - Failure to process credit limit requests between $4000 and $40000 correctly
  - Failure to disapprove credit limit requests less than $4000
  - Failure to disapprove credit limit requests greater than $40000
  - Mishandling of negative credit limit requests
- Partition the input domain into equivalence classes based on risks
  - $4000, $40000
  - $3999
  - $4001
  - -$4000

Equivalence class testing

- Rationale: complete testing but no redundancy
- Cover each partition at least once
- E.g., to test whether software can identify the equilateral triangle
  - After using (5, 5, 5) as test input
  - We do not expect to learn much from using (6, 6, 6), (10, 10, 10) as test inputs

Linear domain variable with multiple ranges

If Taxable Income Is Between:	The Tax Due Is:
0 - $9,225	10% of taxable income
$9,226 - $37,450	$922.50 + 15% of the amount over $9,225
$37,451 - $90,750	$5,156.25 + 25% of the amount over $37,450
$90,751 - $189,300	$18,481.25 + 28% of the amount over $90,750
$189,301 - $411,500	$46,075.25 + 33% of the amount over $189,300
$411,501 - $413,200	$119,401.25 + 35% of the amount over $411,500
$413,201 +	$119,996.25 + 39.6% of the amount over $413,200

Linear domain variable with multiple ranges (cont')

- Identify risks
  - Failure to calculate the tax correctly for each of the income sub-ranges
  - Mishandling of low and high boundaries of each of the sub-ranges
  - Mishandling of values just beneath and beyond low and high boundaries respectively for each of the sub-ranges

Equivalence class testing vs. boundary value testing

Figure 6.1 Traditional equivalence class test cases.

*

* Software Testing, A craftsman's approach, 4th Edition

Boundary value testing

- Rationale: Errors tend to occur near the extreme values of an input variable, e.g.,
  - Loop condition, may test for < when they should test ≤
- The idea is to use input values

  Min, min+, nom, max-, max

- Two variables (hold one in nom and vary the other one)

  $<X_{1nom}, X_{2min}>$ $<X_{1nom}\ X_{2min+}>$ $<X_{1nom}, X_{2nom}>$ $<X_{1nom}, X_{2max-}>$ $<X_{1nom}, X_{2max}>$

  $<X_{1min}, X_{2nom}>$ $< X_{1min+}\ X_{2nom}>$ $< X_{1max-}, X_{2nom}>$ $< X_{1max}, X_{2nom}>$

- Several variables (hold one in nom and vary the others)

Robust boundary value testing

Figure 6.1 Traditional equivalence class test cases.

*

* Software Testing, A craftsman's approach, 4th Edition

Special value testing

- Special value testing of the tax calculation system
  - Mishandling of non-numbers
  - Mishandling of negative incomes
  - Mishandling of smallest and largest value at the system level, and beyond
  - …
- Another special value testing example
  - E.g., test how the system react if *29 Feb.* as the input of a date
- Most intuitive and least uniform (ad hoc)

String variables

- For example, a string variable ‘s’ has to have a letter for the first character; the rest of them can be any printable graphic ASCII characters.

- Domain
  - First character

    (A-Z) 65 <= ASCII (first character) <= 90

    (a-z) 97 <= ASCII (first character) <= 122
  - Remaining characters

    32 <= printable graphic ASCII characters <= 126

String variables (cont')

- Identify risks
  - Failure to process strings correctly that have letters as their first character (B3, Zb)
  - Failure to process boundary characters (A~) (the ASCII code of ~ is 126)
  - Mishandling of string values that have nonletters for the first character ([A)
  - Mishandling of characters that belong to the extended ASCII set (Aβ)

Enumerated variable

- A variable which takes only a set of values/set of options

- Risks
  - Failure to present font effect correctly that has a Superscript/Subscript/… option selected
  - Mishandling of no option selected
  - Mishandling of multiple options selected

Multidimensional variables

- Multidimensional variable

  There is more than one dimension of analyzing such a variable

- Software application example

  To log in a system, the user must input the correct username and password. The username can have five to fifteen characters. Also, only digits and lowercase characters are allowed for the username.

- Username variable has two dimensions

  Length and String

Menti Exercise

Go to **www.menti.com** and use the code **6438 5825**

Outline

- Single variable
- ➢ Multiple variables
- Error guessing/Risk based
- Output coverage and testing

Multivariable application example (1)

- Boundary value testing <1, 0, 1> <1, 0, 2><1, 0, 50> <1, 0, 99> <1, 0, 100>
  <1, 1, 1> <1, 1, 2><1, 1, 50> <1, 1, 99> <1, 1, 100> ...
- Equivalence class testing <1, 0, 1> <1, 1, 0> <101, 0, 1> <101, 1, 0> ...

- Also need to analyze complex logical relationships of the input variables

Decision table-based testing (cont')

<4, 1, 2> <3, 3, 3> <2, 2, 3>

Table 7.2 Decision Table for Triangle Problem

c1: a, b, c form a triangle?	F	T	T	T	T	T	T	T	T
c2: a = b?	—	T	T	T	T	F	F	F	F
c3: a = c?	—	T	T	F	F	T	T	F	F
c4: b = c?	—	T	F	T	F	T	F	T	F
a1: Not a triangle	X								
a2: Scalene									X
a3: Isosceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

*

* Software Testing, A craftsman's approach, 4th Edition

Why test variable combination?

- All variables will interact as they are part of one functional unit and they have to unite to achieve the duty that the functional unit is designated for
- Hence most of these variables influence each other in some, or the other way
- Testing variables in combination may reveal certain bugs that might not have been found by testing the variables in isolation

Multivariable application example (2)

- Online shopping system example
  - Parameters: Availability, Payment Method, Carrier, Delivery Schedule, Export Control

Availability	Payment	Carrier	Delivery Schedule	Export Control
Available	Credit	Mail	One Day	True
Not in Stock	Paypal	UPS	2-5 Working Days	False
Discontinued	Gift Voucher	Fedex	6-10 Working Days	
No Such Product			Over 10 Working Days	

4x3x3x4x2 = 288 combinations

The challenge of testing the online shopping system example

- Suppose there is a bug, and Credit does not work well with One Day delivery
  - Any combination of inputs that include Credit and a One Day delivery will expose that bug
  - Interaction between 2 variables
- Suppose Credit does not work well with a One Day delivery, but only with Fedex
  - Any combination of inputs that include Credit, a One Day delivery, and not with Fedex will expose that bug
  - Interaction between 3 variables
- Do we really need to test all 4x3x3x4x2 = 288 combinations?

Do we really need to test all combinations?

- The root cause analysis of many bugs shows they depend on the value of one variable (20%-68%)
- Most defects can be discovered in tests of the interactions between the values of two variables (65-97%)

Number of variables involved in triggering software faults*

Vars	Medical Devices	Browser	Server	NASA GSFC	Network Security
1	66	29	42	68	20
2	97	76	70	93	65
3	99	95	89	98	90
4	100	97	96	100	98
5		99	96		100
6		100	100		

*http://csrc.nist.gov/groups/SNS/acts/ftfi.htm

What is n-way test?

- Given any n variables (out of all the variables) of a system, every combination of values of these n variables is covered in at least one test

- For example, an application has 5 variables
  - 2-way test

    Every combination of values of any 2 variables (out of 5 variables) is covered in at least one test
  - 3-way test

    Every combination of values of any 3 variables (out of 5 variables) is covered in at least one test

What is n-way test (cont')?

- An application with 5 input variables A, B, C, D, E
  - A has values A1, A2
  - B has values B1, B2, B3,
  - C has values C1, C2
  - D has values D1, D2, D3, D4
  - E has values E1, E2
- Full combinations 2x3x2x4x2 = 96 test cases
- 2-way test (56 combinations)
  - Every combination of values of any 2 variables
  - AB, AC, AD, AE, BC, BD, BE, CD, CE, DE
  - AB (A1-B1, A1-B2, A1-B3, A2-B1, A2-B2, A2-B3)
  - AC (A1-C1, A1-C2, A2-C1, A2-C2)
  - …

All-Pairs combination testing

- 2-way testing
  - Every combination of values of 2 variables is covered in at least one test

- All-pair combination/pairwise testing is condensed 2-way testing
  - Many pairs can be tested at the same time in one combination

A1-B1, B1-C2, C2-D2 -> A1-B1-C2-D2

All-pair combination/pairwise testing approaches

- In Parameter Order
- Variation of In Parameter Order

In Parameter Order

- Initialization phase
- Horizontal growth phase
- Vertical growth phase

Example of In Parameter Order

- Variables
  - X (True, False)
  - Y (0, 5)
  - Z (P, Q, R)

In Parameter Order - Initialization

- X (True, False)
- Y (0, 5)

Find any two variables and generate test cases

- T= (True, 0)
  (True, 5)
  (False, 0)
  (False, 5)

In Parameter Order - Horizontal growth

- X (True, False)
- Y (0, 5)
- Z (P, Q, R)

- Add one more variable in each step to generate more combination
- Remove duplications

All 2-way test combinations

(True, 0)
(True, 5)
(False, 0)
(False, 5)

+

P
Q
R
P?Q?R?

vs.

~~(True, P)~~ ~~(True, Q)~~, (True, R)
~~(False, P)~~, (False, Q), ~~(False, R)~~
~~(0, P)~~, (0, Q), ~~(0, R)~~
~~(5, P)~~, ~~(5, Q)~~, (5, R)

In Parameter Order - Vertical growth

Merge short combinations to longer ones

A Variation of In Parameter Order

- Initialization phase
  - Order variables based on the number of their possible values
  - Generate test cases by full combinations of the first two variables
- Vertical growth phase
  - Add other variables one-by-one to generate a combination
  - Re-order certain variable values if needed
- Horizontal growth phase
  - If re-order cannot generate enough new combinations
  - Extend horizontally to add possibility of using re-ordering to generate new combinations

A Variation of In Parameter Order - Example

- A car ordering application *

- Variables
  - Order category (Buy, Sell)
  - Location (Oslo, Trondheim)
  - Car brand (BMW, Audi, Mercedes)
  - Registration numbers (Valid, Invalid)
  - Order type (E-Booking, In-store)
  - Order time (Working hours, Non-working hours)

*www.softwaretestinghelp.com/what-is-pairwise-testing/

Variation of In Parameter Order – Initialization (reorder variables)

- Order variables based on their number of values

- The results after ordering
  - **Car brand (BMW, Audi, Mercedes)**
  - Order category (Buy, Sell)
  - Location (Oslo, Trondheim)
  - Registration numbers (Valid, Invalid)
  - Order type (E-Booking, In-store)
  - Order time (Working hours, Non-working hours)

Vertical growth phase – iteration 1

Car Brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy				
BMW	Sell				
Audi	Buy				
Audi	Sell				
Mercedes	Buy				
Mercedes	Sell				

Vertical growth phase – iteration 2

- Add other variables one-by-one to generate a combination
- Re-order certain variable values if needed

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo			
BMW	Sell	Trondheim			
Audi	Buy	Oslo			
Audi	Sell	Trondheim			
Mercedes	Buy	Oslo			
Mercedes	Sell	Trondheim			

Vertical growth phase – iteration 2 (cont')

- After re-ordering

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo			
BMW	Sell	Trondheim			
Audi	Buy	Trondheim			
Audi	Sell	Oslo			
Mercedes	Buy	Oslo			
Mercedes	Sell	Trondheim			

Vertical growth phase – iteration 3

- Add other variables one-by-one to generate a combination
- Re-order certain variable values if needed

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid		
BMW	Sell	Trondheim	Invalid		
Audi	Buy	Trondheim	Valid		
Audi	Sell	Oslo	Invalid		
Mercedes	Buy	Oslo	Valid		
Mercedes	Sell	Trondheim	Invalid		

Vertical growth phase – iteration 3 (cont')

- After re-ordering

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid		
BMW	Sell	Trondheim	Invalid		
Audi	Buy	Trondheim	Valid		
Audi	Sell	Oslo	Invalid		
Mercedes	Buy	Oslo	Invalid		
Mercedes	Sell	Trondheim	Valid		

Vertical growth phase – iteration 4

- Add other variables one-by-one to generate a combination
- Re-order certain variable values if needed

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid	In Store	
BMW	Sell	Trondheim	Invalid	E-booking	
Audi	Buy	Trondheim	Valid	E-booking	
Audi	Sell	Oslo	Invalid	In Store	
Mercedes	Buy	Oslo	Invalid	E-booking	
Mercedes	Sell	Trondheim	Valid	In Store	

Vertical growth phase – iteration 5

If re-order cannot generate enough new combinations

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid	In Store	Working hours
BMW	Sell	Trondheim	Invalid	E-booking	Non-working hours
Audi	Buy	Trondheim	Valid	E-booking	Working hours
Audi	Sell	Oslo	Invalid	In Store	Non-working hours
Mercedes	Buy	Oslo	Invalid	E-booking	Working hours
Mercedes	Sell	Trondheim	Valid	In Store	Non-working hours

Horizontal growth

Extend horizontally to add possibility of using re-ordering to generate new combinations

Car brand	Order category	Location	Registration number	Order type	Order time
BMW	Buy	Oslo	Valid	In Store	Working hours
BMW	Sell	Trondheim	Invalid	E-booking	Non-working hours
	Buy				Non-working hours
Audi	Buy	Trondheim	Valid	E-booking	Working hours
Audi	Sell	Oslo	Invalid	In Store	Non-working hours
	Sell				Working hours
Mercedes	Buy	Oslo	Invalid	E-booking	Working hours
Mercedes	Sell	Trondheim	Valid	In Store	Non-working hours

All-pairs combination testing tools

- ACTS – ‘Advanced Combinatorial Testing System’, provided by NIST, an agency of the US Government

- VPTag free Pairwise Testing Tool

Outline

- Single variable
- Multiple variables
- ➢ Error guessing/Risk based
- Output coverage and testing

Error guessing / Risk-based

- Based on experience
  - Different types of faults
  - Probability of the faults
  - Consequence of the faults
  - …

Bach's risk-based testing*

- Bach's heuristics is based on his experience as a tester
- Two risk analysis approach
  - Inside-out: What can go wrong there?
  - Outside-in: What things are associate with this kind of risk?
- Based on this experience he has identified
  - A generic risk list – things that are important to test
  - A risk catalogue – things that often go wrong to a specific domain

*http://www.satisfice.com/articles/hrbt.pdf

Bach’s generic risk list

- Complex – large, intricate or convoluted
- New – no past history in this product
- Changed – anything that has been tampered with or “improved”
- Upstream dependency – a failure here will cascade through the system
- Downstream dependency – sensitive to failure in the rest of the system
- Critical – a failure here will cause serious damage

Bach's generic risk list (cont')

- Precise – must meet the requirements exactly
- Popular – will be used a lot
- Strategic – of special importance to the users or customers
- Third-party – developed outside the project
- Distributed – spread out over time or space but still required to work together
- Buggy – known to have a lot of problems
- Recent failure – has a recent history of failures

Bach’s risk catalogs example

- Wrong files installed
  - Temporary files not cleaned up
  - Old files not cleaned up after upgrade
  - Unneeded file is installed
  - Needed file not installed
  - Correct file installed in the wrong space
- Files clobbered
  - Old file replaces newer file
  - User data file clobbered during upgrade
- …

Outline

- Single variable
- Multiple variables
- Random testing
- Error guessing/Risk based
- ➢ Output coverage and testing

Output coverage

- All the coverage types that we have looked at so far have been related to input data
- It is also possible to define coverage based on output data
- Uncovered outputs help us define the new test cases

Output coverage (cont')

Output coverage (cont”)

- The main challenge with using output coverage measure is that output can be defined at several levels of details
  - An account summary
  - An account summary for a special type of customer
  - An account summary for a special event

Summary

- We studied
  - How to identify the domains of input & output variables
  - How to derive function test cases based on single and multiple domain variables
- Can be applied to
  - Unit test
  - Integration test
  - System test
  - Acceptance test

Integration, system, acceptance, and regression testing

V-model

Business requirements
Value realization test
Deployment test
Software requirements
Operation test
System test
We cover these tests
Regression test
Start - criteria
Workflow
Design
Integration test
External consistency
Unit design
Unit test
Internal consistency
Testing
Programmering

Integration testing

- Focus on testing interfaces between components
- Not as well understood as unit testing and system testing
- Usually poorly done

- Strategies
  - Decomposition-based
  - Call graph-based/interface matrix
  - Path-based

Decomposition-based integration

- Based on functional decomposition tree
- Need to know function dependencies between components
- Based on the order of integration testing, can be classified into four strategies
  - Top-down
  - Bottom-up
  - Sandwich
  - Big bang

Decomposition-based integration (cont')

- Top-down
  - Begin with main
  - Use "stubs" to simulate called functions
  - Replace stubs with real functions one by one

- Bottom-up
  - Begin with leaves
  - Use "drivers" to emulate functions call the leaves
  - Replace "drivers" with real function later

- Sandwich
- Big bang

Pros and cons of decomposition-based integration

- Pros
  - Incremental and intuitive
  - Easy fault isolation

- Cons
  - Need “stub” or “driver”

Call graph-based/interface matrix-based integration

- Based on the call graph/interface matrix of components
- Use actual components rather than “stubs” or “drivers”
- Two strategies
  - Pairwise integration
  - Neighborhoods integration

Pair-wise integration

*

Figure 13.9 Three pairs for pairwise integration.

* Software Testing, A craftsman's approach, 4th Edition

Interface matrix

	Sys 1	Sys 2	Sys 3	Sys 4
Sys 1	X	X		
Sys 2	X	X		
Sys 3			X	X
Sys 4			X	X

Neighborhoods integration

Figure 13.10 Three neighborhoods (of radius 1) for neighborhood integration.

Table 13.1 Neighborhoods of Radius 1 in Calendar Call Graph

Neighborhoods in Calendar Program Call Graph			
Node	*Unit Name*	*Predecessors*	*Successors*
1	Calendar (Main)	(None)	2, 3, 4, 5, 6, 7
2	getDate	1	8, 9
3	zodiac	1	9

* Software Testing, A craftsman's approach, 4th Edition

Pros and cons of call graph-based integration

- Pros
  - Do not need “stub” or “driver”
- Cons
  - Limited to local integration

Path-based integration

- Not focusing on testing interfaces among separately developed and tested unit
- Rather focus on interactions among these units

Path-based integration (cont’)

- Like enlarged unit testing
  - The node is not a statement
  - The node is a component/test unit
  - The edge is message transferred between components

Pros and cons of path-based integration

- Pros
  - Test more global and complex integrations
  - Closely coupled with actual system behavior

- Cons
  - Difficult fault isolation
  - Extra effort is need to identify message path

System tests categories

- Type of system tests
  - Functionality
  - Reliability
  - Usability
  - Performance
  - Robustness
  - Scalability
  - Stress
  - Load and stability
  - …

Robustness testing

- How sensitive is a system to erroneous inputs or changes in the operational environment?
  - Stupid/uncommon inputs (like testing special values in boundary testing)
  - Failures from other systems
  - Degraded node
  - Etc.

Scalability testing

- To identify how well the system can scale, i.e., the magnitude of demand that can be placed on the system while continuing to meet performance requirements

Stress testing

- It can ensure the system can perform acceptably under the worse-case condition

- The system is deliberately stressed by pushing it to and beyond its specific limits for a while

- For testing
  - Buffer allocation and memory carving
  - Network bandwidth

CULTURE

Olympics ticket system crashes

Domestic ticket sales in China are halted after 8 million hits and 200,000 orders per second overwhelm the booking system.

BY SUZANNE TINDAL | NOVEMBER 1, 2007 6:18 AM PDT

Load and stability testing

- Important for providing 24x7 services
- System and applications that run for months are likely to
  - Slow down
  - Encounter functionality problems
  - Silently failover
  - Crash altogether
- Ensure the system remains stable for a long period of time under full load

Scalability vs. Stress vs. Load and statbility testing

- Scalability

- Stress

- Load and stability

Using Apache JMeter to run load and performance testing

https://youtu.be/yNnbW2n9s8E

Acceptance testing

- ISTQB : **(user) acceptance testing**: Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.

Typical forms of acceptance testing

User acceptance testing

- Functions are correct?
- Test basis
  - User/business requirements
  - System requirements
  - Use cases
  - Business processes
  - Risk analysis reports

Operational acceptance test

- Read to operate?
  - Backup facilities
  - Procedures for disaster recovery
  - Training or manual for end-users
  - Maintenance procedures and manual
  - Security procedures

Contract and regulation acceptance testing

- Test against contract

- Test against regulations
  - Governmental regulation
  - Legal standards
  - Safety standards

Alpha and beta testing

- Alpha
  - At developers' sites
  - By internal staff
  - Before it is released to external customers

- Beta (field testing)
  - At customers' sites
  - Before the system is released to other customers

Acceptance testing in agile

- The terms "functional test", "acceptance test" and "customer test" are used more or less interchangeably.

- Referring to user stories

Outline

- Testing approaches
  - Integration tests
  - System tests
  - Acceptance tests
- ➢ Regression tests

Regression testing

- Basic regression testing concepts
- Regression test case selection
- Regression minimization
- Regression test case prioritization

What is regression testing?

- Constitute the vast majority of testing effort in many software development projects

Two types of regression testing

- Corrective regression testing
  - No requirements change
  - Modified code behaves correctly
  - Unchanged code continues to behave correctly

- Progressive regression testing
  - Requirements change
  - Newly added or modified code
  - Unchanged code continues to behave correctly

Overview of first & regression test cases*

Test code that is unchanged and not impacted by the changed code, unnecessary to run in regression testing, but can be run if you like

Test changed code and code impacted by the changed code, must run in regression testing

* https://www.uio.no/studier/emner/matnat/ifi/INF4290/v11/undervisningsmateriale/INF4290-RegTest.pdf

Regression testing processes

- Test revalidation
  - Are test cases obsolete?
- (Regression) test selection
  - Verify changed and impacted code (corrective)
  - Verify new structure and requirements (progressive)
- Test minimization
  - Remove redundant test cases
- Test prioritization
  - Rank test cases and run them according to available resources

Test revalidation

- How can a test case be obsolete?
  - Their input/output relation is no longer correct due to changes in specification
  - They no longer test what they were designed to test due to modifications to the program
  - They are “structural” test cases that no longer contribute to structural coverage of the program

(Regression) test selection

- We now consider only corrective regression testing
- Strategies of (regression) test selection
  - Retest all
    - Can be costly, unless test execution is highly automated
  - Random selection
    - Better than no test
    - Hard to ensure coverage of the changed code
  - Selecting modification traversing tests
    - “Safe” regression test selection technique is preferred

Automated regression testing example

https://youtu.be/2jAy9cdblaE

Safe regression test selection

A technique that does not discard any test that will traverse a modified or impacted statement is known as "safe" regression test selection technique

- Assume program P has been tested by test set T against specification S
- Assume P is modified to P' to fix some bugs (i.e., specification S is not changed)

Question:

- What information is needed in order to select a safe regression test subset from T to verify P'?

General (regression) test selection process

- Establish traces between P and T
  - Execute P with test cases
  - Record program entities executed when running tests
- Compare P with P' to find differences
  - Identify program entities changed
- Select test cases from T that traverse the changed program and impacted entities

General (regression) test selection process (cont')

- Based on program entities traced and compared, the test selection methods can be classified into
  - Dynamic slicing based approach (statements)
  - Graph-walk approach (nodes in control flow graph/program dependence graph)
  - Firewall approach (OO classes, COTS components, etc.)
  - …

Graph-walk approach example

Code P

```
int  M (int x, int y){
  int z;
  if (x<y)
     z = f1 (x, y);
  else
     z = f2(x, y);
  return z;
}
```

```
int f1( int a, int b){
  if ((a+1) == b)
     return a*a;
  else
     return b*b; }
```

```
int f2( int a, int b){
  if (a == (b+1))
     return b*b;
  else
     return a*a; }
```

Code P’

If((a-1) == b)

Test set T of P

t1: <x = 1, y = 2>

t2 : <x = 1, y = 3>

t3 : <x = 3, y = 1>

Which test cases to be included in the safe subset to test P’?

Graph-walk approach example – step 1

- Establish trace between test case and CFG (control flow graph) nodes

Test set T

t1: <x = 1, y = 2>

t2 : <x = 1, y = 3>

t3 : <x = 3, y = 1>

Nodes traversed by tests

Functions	Nodes			
	1	2	3	4
M	t1, t2, t3	t1, t2	t3	t1, t2, t3
f1	t1, t2	t1	t2	-
f2	t3	None	t3	-

Graph-walk approach example – step 2

- Compare P and P' to find differences

If((a-1) == b)

Node 1 in f1

Graph-walk approach example – step 3

- Select test cases from T that traverse the changed CFG nodes

Test set T

t1: <x = 1; y = 2>

t2 : <x = 1, y = 3>

t3 : <x = 3, y = 1>

Set of tests that traverse a certain node

Functions	Nodes			
	1	2	3	4
M	t1, t2, t3	t1, t2	t3	t1,t2, t3
f1	t1, t2	t1	t2	-
f2	t3	None	t3	-

A technique that does not discard any test that will traverse a modified or impacted statement is known as “safe” regression test selection technique.

- Here, we did not cover the impacted statement
- Test oracles of the tests may need to be changed

General (regression) test selection process (cont')

- Based on program entities traced and compared, the test selection methods can be classified into
  - Dynamic slicing based approach (statements)
  - Graph-walk approach (nodes in control flow graph/program dependence graph)
  - Firewall approach (OO classes, COTS components, etc.)
  - …

Firewall approach

What is a firewall?

- A ***firewall*** separates the classes that depend on the class that is changed from the rest of the classes

Firewall approach (cont')

Why use a firewall?

- The ***firewall*** approach is based on the first-level dependencies of modified components

- The most severe effects are in those components that send messages to the changed component or receive messages from the changed component

Process for determining “Firewall” in OO systems

1. Identify classes changed

2. Inheritance? Then descendants of the changed classes

3. Classes send messages to or receive messages from the changed class

One empirical study of using firewall approach [1]

- System studied
  - Distributed component-based J2EE sys. owned by Swedbank
  - 1.2M line of code from 27 000 classes
  - 2 new releases of the system

- Steps 1: Coverage analysis and establishing trace
  - Instrumentation of Java byte code and run-time collection of coverage data during original test
  - Used a tool called Emma [2]

One empirical study of using firewall approach [1] (Cont')

Step 2: Find classes in firewall

- Identify changes – produced MD5 signatures for each compiled class file. If code changes, class signature will be different.
- Own tool

Step 3: Extracting dependency between classes

- Dependency finder [3]

Step 3: Select test cases that traverse classes in firewall

Another empirical study of using firewall approach [4]

- Studied an OO telecommunication software system at ABB

- Compared two firewall approaches
  - One level from changed component
  - More than one level from the changed component, depending on their logical dependencies

Another empirical study of using firewall approach [4] (Cont')

- Observation
  - Extended firewall approach finds more faults than classical firewall approach, but more costly to run

- Recommendations
  - Routine incremental changes (one level approach)
  - Major release (> one level approach)

Test minimization

- To reduce redundancies in the safe subset
- If every entity covered by t2 is a subset of entities covered by another test case t1, then we remove t2
- The entities could be
  - Statement, CFG nodes, functions, etc.

Test minimization example

- t1 covers the following statement
- t2 also covers the same statement
- Then, we can remove t2

(a<b)||(a<c)

Risk of test minimization

- Test minimization is **risky** and is not necessarily safe
  - It depends on the modifications (from P to P'), the faults, the entities used

(a<b)||(a<c)

t2 may test the (a<c) part and t1 may test the (a<b) part

If the t2 is removed because this statement is covered by t1, then, we will miss the opportunity to test (a<c)

Test prioritization

- Different from test minimization

- Test prioritization
  - Is not going to remove any test cases from the safe subset
  - Is to rank regression tests based on some criteria

- The goal is to reveal faults early

Test prioritization approaches

- Unlike test prioritization of the first test
  - **More information available**
  - Fault finding effectiveness
  - Test coverage
  - Cost of running the test

- Prioritization strategies [5]
  - Coverage based (high coverage)
  - Cost-aware based (low cost to run, high fault-finding probability)
  - …

Summary

- We studied the following testing approaches
  - Integration
  - System
  - Acceptance

- We studied regression test
  - Selection
  - Minimization
  - Prioritization

References

- [1] White L., Abdullah K. A firewall approach for the regression testing of object-oriented software. Proc. of the Software Quality Week, 1997.
- [2] http://emma.sourceforge.net/
- [3] Skoglund M. and Runeson P. A Case study of The Class Firewall Regression Test Selection Technique on Large Scale Distributed Software System. Proc. of empirical software engineering conf. 2015.
- [4] White L. Jaber K. Robinson B. and Rajich V. Extended firewall for regression testing: an experience report. Journal of software maintenance and evolution. Vol. 20. 2008
- [5] Yoo S., Harman M. Regression testing minimization, selection and prioritization: a survey, Software testing, verification, and reliability, 2007.

NTNU

System Testing Planning

Theory and Practice

Chapter 12

System Test Planning and Automation

AGENDA

1

Testing Planning

2

Testing Execution

Why do we have to plan tests?

- The purpose is to get ready and organized for test execution
- Without a plan it is highly unlikely that
  - the desired level of system testing is performed within the stipulated time and without overusing resources
  - Expected Costs and deadlines will be reached
- Planning for system testing IS part of the overall planning for a software project!
- It provides the framework, scope, resources, schedule, and budget for the system testing part of the project.

A System Test Plan

- Provides guidance for the executive management to support the test project
- Establishes the foundation of the system testing part of the overall software project
- Provides assurance of test coverage by creating a requirement traceability matrix
- Outlines an orderly schedule of events and test milestones that are tracked
- Specifies the personnel, financial, equipment, and facility resources required to support the system testing part of a software project

Planning the Test (Outline)

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Introduction

The *introduction* section of the system test plan includes:

- Test project name
- Revision history
- Terminology and definitions
- Name of the approvers and the date of approval
- References
- Summary of the rest of the test plan

The *feature description* section summarizes the system features that will be tested during the execution of this plan.

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Assumptions

- The *assumptions* section describes the areas for which test cases will not be designed
- Examples:
  - The necessary equipments to carry out scalability testing may not be available
  - It may not be possible to procure third-party equipments in time to conduct interoperability testing
  - It may not be possible to conduct compliance test for regulatory bodies and environment tests in the laboratory

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Test Approach

- Issues discovered by customers that were not caught during system testing in the past project
- Outstanding issues that need to be tested differently need to be discussed here
- Discuss test automation strategy
- Test cases from the database that can be re-used in this test plan

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Test Suite Structure

- Detail *test groups* and *subgroups* based on the test categories identified in the *test approach* section
- Test objectives are created for each test group and subgroups based on the system requirements and functional specification documents
- Identification of test objectives provides a clue to the total number of test cases needs to be developed
- A traceability matrix is generated to make an association between requirements and test objectives to provide the test coverage

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Test Environment

Multiple test environments are constructed in practice

Test activities can not interfere with development activities

To run security tests you need an environment that can be put down.

To run scalability tests one may need more resources than to run functionality tests

Test Environment

Test Bed Preparations

Get Information about the customer deployment architecture including hardware, software and their manufacturers

List of third-party products to be integrated with the SUT.

Third party test tools to be used to monitor, simulate, and /or generate real traffic.

Hardware equipment necessary to support special features specified in the requirements, for example backup, some network switches etc.

Analysis of the non-functional requirements for checking what is necessary to perform these tests

List of small but necessary networking, cables hubs, needed for the test.

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Test Execution Strategy

- It’s a game plan that addresses the following concerns:
  - How many times are the tests cases executed and when?
  - What does one do with the failed test cases?
  - Which test cases groups are blockers?
  - What happens when too many test cases fail?
  - In which order are the test cases executed?

Test Execution Strategy

- The processes of system test execution, defect detection, and fixing defects are intricately intertwined
  - Some test cases cannot be executed unless certain defects are detected and fixed
  - A programmer may introduce new defects while fixing one defect, which may not be successful
  - The development team releases a new build for system testing by working on a subset of the reported defects, rather than all the defects
  - It is a waste of resources to run the entire test set T on a build if too many test cases fail
  - When the system is more stable it may be wise to start prioritizing the tests for regression tests

An effective and efficient execution strategy must take into account the characteristics of the test execution, defect detection and defect removal!

A Multi-Cycle System Test Strategy

(a) Progress of system testing in terms of test cycles

(b) Desired increase in quality level

Characterization of Test Cycles

- Goals and Assumptions
- Test execution
- Revert and Extension criteria
- Actions
- Exit criteria

Characterization of Test Cycles

Goals and Assumptions

- **System test team sets its own goals to be achieved in each test cycle**
- **Goals are specified in terms of the number of test cases to pass in a cycle;**
- **Assumptions specify some "tolerance level for the test"**

Characterization of Test Cycles

Test Execution

- **Test Cases are executed in multiple test environments as specified in the test plan.**
- Prioritization of the test execution changes between test cycles.
- **Some good practices for prioritization of test cases**
  - **Test cases that exercise basic functionalities have higher priority than the rest in the first test cycle**
  - **Test cases that have failed in one test cycle have a higher priority in the following test cycle**
  - **Test cases in certain groups have higher priority than others**

Characterization of Test Cycles

Revert and Extension criteria

- **The conditions for prematurely terminating a test cycle and for extending a test cycle must be precisely stated**
- **It may not be useful to continue a test cycle if it is found that a software is of poor quality**
- **Often a test cycle is extended due to various reasons**
  - A need to re-execute all the test cases in a particular test group because a large fraction of the test cases within the group failed
  - A significantly large number of new test cases were added while test execution was in progress

Characterization of Test Cycles

Actions (Examples)

- Too many test cases may fail during a test cycle
  - **The development team needs to be alerted**
  - **The developers should take action in the form of root cause analysis (RCA)**
  - **Corrective actions are taken by updating the design specification, reviewing the code, and adding new test cases to the unit and integration test plan**
- The system test team has to design a large number of new test cases during a test cycle
  - **Tester have then to see why this happened.**
    - Mistmatches between what the tester expected and what was delivered?
    - Poor plannning of the tests ?
    - Different expectations of the tests objectives for that cycle?

Characterization of Test Cycles

Exit Criteria

- **An exit criterion specifies the termination of a test cycle**
  - Mere execution of all the test cases in a test suite does not mean that a cycle has completed maybe there is a need to add more “quality” criteria:
    - a percentage of passed/not passed
    - Number of environment that is acceptable
- **Example of exit criteria**
  - **95% of test cases passed and all the known defects are in CLOSED state**

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Test Effort Estimation

Cost of the test and the time to complete the test

- The number of test cases to be designed for the software project based on:
  - Test Group Category
  - Function Points
- Effort required to create a detailed test case
- Effort required to execute a test case and analyze the results of execution
- Effort needed to create the test environments
- Effort needed to report the defects, reproduce the failures and follow up with development team

Test Effort Estimation

TABLE 12.6 Test Effort Estimation for FR–ATM PVC Service Interworking

Test (Sub)Groups	Estimated Number of Test Cases	Person-Day to Create	Person-Day to Execute
Configuration	40	4	6
Monitoring	30	4	4
Traffic Management	3	2	3
Congestion	4	2	4
SIWF Translation Mode Mapping	12	4	4
Alarm	4	1	1
Interface	9	5	3
Robustness	44(4 + 40)	4	6
Performance	18	6	10
Stress	2	1.5	2
Load and stability	2	1.5	2
Regression	150(60 + 90)	0	15
Total	**318**	**35**	**60**

Scheduling and Test Milestones

ID	Task Name	Duration	Start	Finish
1	Development of the test plan	5 days	Jan 3	Jan 7
2	Review the test plan	0 days	Jan 7	Jan 7
3	Update the test plan	1 day	Jan 10	Jan 10
4	Test plan approved	0 days	Jan 10	Jan 10
5	Procurement of equipment	7 days	Jan 11	Jan 19
6	Set up the test environment	10 days	Jan 20	Feb 2
7	Creation of test cases in test factory	18 days	Jan 11	Feb 3
8	Review the test cases	0 days	Feb 3	Feb 3
9	Update the test cases in test factory	4 days	Feb 4	Feb 9
10	Test cases are in released state	0 days	Feb 9	Feb 9
11	Creation of test suite in test factory	1 day	Feb 10	Feb 10
12	Entrance criteria readiness meeting	1 day	Feb 11	Feb 11
13	Official release of S/W to system test	0 days	Feb 14	Feb 14
14	Start of first test cycle	0 days	Feb 14	Feb 14
15	First test cycle	15 days	Feb 14	Mar 4
16	End of first test cycle	0 days	Mar 4	Mar 4
17	Start of second test cycle	0 days	Mar 4	Mar 4
18	Second test cycle	15 days	Mar 7	Mar 25
19	End of second test cycle	0 days	Mar 25	Mar 25
20	Beta release criteria review meeting	0 days	Mar 25	Mar 25
21	Release the software to beta customer	1 days	Mar 28	Mar 28
22	Start of third test cycle	0 days	Mar 28	Mar 28
23	Third test cycle	5 days	Mar 29	Apr 4
24	End of third test cycle	0 days	Apr 4	Apr 4
25	Test report preparation	4 days	Apr 5	Apr 8
26	Release criteria review meeting	0 days	Apr 8	Apr 8
27	Release the software to the customer	1 day	Apr 13	Apr 13

Planning the Test

- Introduction and Feature Description
- Assumptions
- Test Approach
- Test Suite Structure
- Test Environment
- Test Execution Strategy
- Test Effort Estimation
- Scheduling and Milestones

Menti?

AGENDA

1

Testing Planning

2

Testing Execution

NTNU

System Test Execution

Theory and Practice

Chapter 13

System Test Execution

Outline

- Lifecycle of a Defect
- Metrics for Monitoring Test execution
- Metrics for Monitoring Defect Reports
- Defect Analysis Techniques:
- Measuring Test Effectiveness
- System Test Report

Test Execution

- It is important to monitor the processes of test execution, tracking defects and measure test effectiveness.
  - There is a pressure to meet a tight schedule close to the delivery date;
  - There is a need to discover most of the defects before delivering the product;
  - It is essential to verify that defect fixes are working and have not resulted in new defects

Lifecycle of a Defect

- The life-cycle of a defect is usually simplistically represented by a state-transition diagram with five states:
  - NEW
    - A problem report with a severity level and priority is filled
  - ASSIGNED
    - The problem is assigned to an appropriate person
  - OPEN
    - The assigned person is actively working on the problem to resolve it
  - RESOLVED
    - The assigned person has resolved the problem and is waiting for the submitter to verify and close it.
  - CLOSED
    - The submitter has verified the resolution of the problem and accepted it.

Lifecycle of a Defect Jira

https://www.educba.com/jira-bug-life-cycle/

Defects Characteristics

Priority - A Measure of how soon the defect needs to be fixed (urgency)

Critical – must be resolved as soon as possible

High – must be resolved in a high priority

Medium – must be resolved but there is no pressure of time

Low – should be resolved when possible.

Severity - A measure of the extent of the detrimental effect of the defect on the operation of the product.

Critical : when one or more critical system functionalities are impaired and there is no workaround;

High: some functionalities of the system are impaired but there are workarounds;

Medium : no critical functionalities of the system is impaired.

Low: the problem is mostly cosmetic.

Defects Characteristics Jira

View Priorities

The table below shows the priorities used in this version of JIRA, in order from highest to lowest.

- Translate priorities

Name	Description	Icon	Color	Order	Operations
Highest	This problem will block progress.				Edit · Delete · Default
High	Serious problem that could block progress.				Edit · Delete · Default
Medium	Has the potential to affect progress.				Edit · Delete · Default
Low	Minor problem or easily worked around.				Edit · Delete · Default
Lowest	Trivial problem with little or no impact on progress.				Edit · Delete · Default

Add New Priority

Name *

Description

Icon URL * [select image]

(relative to the JIRA web application e.g /images/icons OR starting with http://)

Priority Color *

Add

https://www.cwiki.us/display/JIRA064/Defining+Priority+Field+Values

Defects Characteristics Jira

https://www.softwaretestinghelp.com/jira-bug-tracking/

Metrics for Tracking the Test Execution

- Test execution brings forth 3 different facets of software development:
  - Developers would like to know the degree to which the system meets the explicit as well as implicit requirements.
  - The delivery date cannot be precisely predicted due to the uncertainty in fixing problems.
  - The customer is excited to take the delivery of the product.

Metrics for Tracking the Test Execution

- It is very important to monitor certain metrics which truly represent the progress of system testing and reveal the quality level of the system!
  - Monitoring the test execution
  - Monitoring the defects
- This is very important for timely decisions about the test!

Metrics for Monitoring Test Execution

Test case Escapes (TCE)

- A significant increase in the number of NEW test cases during execution implies that deficiencies in the test design

Planned versus Actual Execution (PAE) Rate

- Compare the actual number of of test cases executed every week with the planned number of test cases

Execution Status of Test (EST) Cases

- Periodically monitor the number of test cases lying in different states
  - *Failed, Passed, Blocked, Invalid* and *Untested*

Monitoring Test Execution Jira

project IN (BM, AB) AND issuetype = Bug and assignee=currentuser() and priority in (blocker, critical) and status in ("To Do", open) Basic

1–9 of 9

T	Key	Summary	Assignee	Reporter	P	Status	Resolution	Created	Updated
	BM-132	Test	Swati Seela [Administrator]	Swati Seela [Administrator]		OPEN	Unresolved	17/Feb/16	03/Mar/16
	AB-38	Rest_validatAccount_creditcard_ ; credit card validation gives incorrect error messgae when Expiry date is prior than the present date.	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-37	Update functionality for SSN,accets input'0'with success,but does not update in DB	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-35	Timeout Error for server socket	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-29	Buy credit card accepts negative amount and it's adding in available limit in database	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-27	In CC card restful service "Add account" Credit limit ,available limit and cash limit accepts 0 and show "msg": "Scuccessfully saved credit card information".	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-26	In CC restful service "add account" CC expiry date should be greater than the current date but acccepts the date prior to the current date.	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15
	AB-19	If we give non matching name and cedit card number i.e name of one person and registered credit card of other person the amount in the request get deposited in the account of name provided.	Swati Seela [Administrator]	Test		TO DO	Unresolved	13/Aug/15	19/Aug/15

https://www.softwaretestinghelp.com/jira-bug-tracking/

Metrics for Monitoring Defect Reports

- Function as Designed (FAD) Count
  - False positives
  - If the number of false positives are high, maybe the testers have inadequate understanding of the system.
- Irreproducible Defects (IRD) Count
  - If a defect cannot be reproduced, then the developers may not be able to gain useful insight into the cause of the failure.
  - If many defects are irreproducible, may mean that the system is unreliable for some reason.
- Defects Arrival Rate (DAR) Count
  - Measures who is finding the defects. For example, if the marketing and users are the ones that are mostly reporting bugs, it may mean that the test is ineffective.

Metrics for Monitoring Defect Reports

- Outstanding Defects (OD) Count
  - Defects that are always coming back. Or never resolved.
  - If defects are not resolved and only increasing, it may mean that the system is not increasing the quality level.
- Crash Defects (CD) Count
  - Defects causing a system to crash must be recognized as an important category of defects because a system crash is a serious event leading to complete unavailability of the system and possibly loss of data.
- Lead time to Resolution of Defects Count
  - If the defects that appear in the system are usually easy to fix and fast to fix it means that the system has better quality.

Orthogonal Defect Classification (ODC)

- A methodology for rapid capturing of the semantics of each software defect.
- In the NEW state, the submitter needs to fill out the following ODC attributes or fields:
  - *Activity:* This is the activity that was being performed at the time the defect was discovered
  - *Trigger:* The environment or condition that had to exist for the defect to surface
  - *Impact:* This refers to the effect, the defect would have on the customer, if the defect had escaped to the field.
- The owner needs to fill out the other ODC attributes or fields when the defect is moved to RESOLVED state.
- The ODC analysis can be combined with Pareto analysis to focus on error-prone parts of the software
  - 80% of the problems can be fixed with 20% of the effort

Orthogonal Defect Classification

- Target: The target represents the high-level identity, such as design, code, or documentation, of the entity that was fixed
- Defect type: The defect type represents the actual correction that was made
- Qualifier: The qualifier specifies whether the fix was made due to missing, incorrect, or extraneous code
- Source: The source indicates whether the defect was found in code developed in house, reused from a library, ported from one platform to another, or provided by a vendor
- Age: The history of the design or code that had the problem. The age specifies whether the defect was found in new, old (base), rewritten, or refixed code.
    - New: The defect is in a new function which was created by and for the current project.
    - Base: The defect is in a part of the product which has not been modified by the current project. The defect was not injected by the current project
    - Rewritten: The defect was introduced as a direct result of redesigning and/or rewriting of old function in an attempt to improve its design or quality.
    - Refixed: The defect was introduced by the solution provided to fix a previous defect.

Monitoring Test Execution Jira

https://www.softwaretestinghelp.com/jira-bug-tracking/

Defect Causal Analysis (DCA)

Used to raise the quality level of the products at a lower cost.

Focuses on understanding of cause-effect relationship of an error to

- Take actions to prevent similar errors from occurring in the future;
- Reduce the number of defects;
- Focus on systematic errors;

Defect Causal Analysis (DCA) Five Steps

Cause-effect diagram for defect causal analysis (DCA

Defect Causal Analysis (DCA)

- Examples of Recommended Actions can come from the DCA:
  - Training
    - People improving their knowledge about the product
  - Improvement in Communication
    - Communication between testers and developers can be of a big help in cases of too many false positives or duplicated defects reporting for example.
  - Using Tools
    - Recommend the use of tools to monitor memory for example
  - Process improvements
    - Actions to improve existing processes or defining new processes.

https://www.knowledgehut.com/blog/agile/root-cause-analysis-agile-teams

Defect Causal Analysis (DCA) in Agile

- Best Place for this are on retrospectives!

https://www.benlinders.com/2013/getting-to-the-root-causes-of-problems-in-a-retrospective/

System Test Report

A final summary report is created after the completion of all the tests

System Test Report

1. Introduction to the Test Project
2. Summary of Test Results
3. Performance Characteristics
4. Scaling Limitations
5. Stability Observations
6. Interoperability of the System
7. Hardware/Software Compatible Matrix
8. Compliance Requirement Status

Table 13.14: Structure of the final system test report.

System Test Report

Testflo example

https://deviniti.com/support/addon/server/testflo-83/latest/test-execution-report/#&gid=null&pid=2

Readiness Criteria Example

- All the test cases from the test suite should have been executed
- Test case results are updated with Passed, Failed, Blocked, or Invalid status
- The requirements are updated by moving each requirement from the Verification state to either the Closed or the Decline state, as discussed in Chapter 11
- The pass rate of test cases is very high, say, 98%
- No crash in the past two weeks of testing has been observed
- No known defect with critical or high severity exists in the product
- Not more than a certain number of known defects with medium and low levels of severity exist in the product
- All the resolved defects must be in the CLOSED state
- The user guides are in place
- Trouble shooting guide is available
- The test report is completed and approved

Measuring Test Effectiveness

- After a product is deployed measure the number of defects found by the customers that were not found by the testers and development team.
  - ESCAPED DEFECTS

Defect Removal Efficiency (DRE) metric defined as follows:

$$\mathrm{DRE} = \frac{\text{Number of Defects Found in Testing}}{\text{Number of Defects Found in Testing} + \text{Number of Detects Not Found}}$$

Measuring Test Effectiveness with Spoilage Metric

A new metric called spoilage is defined as

$$\text{Spoliage} = \frac{\sum(\text{Number of Defects} \times \text{Discovered Phage})}{\text{Total Number of Defects}}$$

Measuring Test Effectiveness with Spoilage Metric

Phase Injected	Phase Discovered								Weight	Total Defects	Spoilage = Weight/Total Defects
	Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing	Acceptance Testing			
Requirements	0	7	6	3	0	0	12	28	56	17	3.294117647
High-Level		0	8	8	3	8	30	6	63	22	2.863636364
Detailed Design			0	13	6	12	20	0	51	25	2.04
Coding				0	63	48	111	48	270	136	1.985294118
Summary	0	7	14	24	72	68	173	82	440	200	2.2

A spoilage value close to 1 is an indication of a more effective defect discovery process

How is this performed in Agile?

Discuss in groups how to do a test planning in testing management tool you know.

You can also use the example of TestFlo

https://deviniti.com/support/addon/server/testflo-84/latest/test-planning/

Further Material

How is this performed in Agile?

Certified Tester

Foundation Level Extension Syllabus
Agile Tester

Version 2014

International Software Testing Qualifications Board

ISTQB

Copyright Notice
This document may be copied in its entirety, or extracts made, if the source is acknowledged.

https://www.istqb.org/downloads/send/5-foundation-level-agile-tester/41-agile-tester-extension-syllabus.html

Further work

A complete quiz about Agile Testing from ISTB with 40 Questions

https://softwaretester.net/istqb_sample_exam_agile_tester_fl/

Code refactoring and review

Outline

- Code refactoring principles
  - Why refactoring
  - Bad smells in code
  - General refactoring principles and techniques
- Refactoring of Python code
- Code inspection and review
  - Testing vs. inspection
  - Reading techniques

Code refactoring principles

Readable and maintainable code is desired by industry

For software development (S), general programming knowledge (GP) and maintenance (MT) are the top two items.

GP: Clean and readable code with good documentation

MT: experience, especially with maintaining code

What competence do software companies want

Tor Stålhane, IDI, NTNU

Guttorm Sindre, IDI, NTNU

Extended version of conference paper "Hva vil programvareindustrien ha?", MNT-konferansen 2019

Stålhane, Deraas, Sindre and Abrahamsson:

Bad smells in code (code smells)

- Code smells are any violation of fundamental design principles that decrease the overall quality of the code.
- Not bugs or errors
- Can certainly add to the chance of bugs and failures down the line.

Code smells categories

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List

- Divergent Change
- Shotgun Surgery

- Feature Envy
- Data Clumps
- Primitive Obsession

- Switch Statements
- Lazy Element
- Speculative Generality
- Temporary Field

- Message Chains
- Middle Man

- Data Class
- Refused Bequest

Cod refactoring goals and properties

- Change the internal structure without changing external behavior

- Eliminate code smells for
  - Readability
  - Consistency
  - Maintainability

- Properties
  - Preserve correctness
  - One step at a time
  - Frequent testing

Code refactoring steps

- Designing solid tests for the section to be refactored
- Reviewing the code to identify bad smells of code
- Introducing refactoring and running tests (One step at a time)

Refactoring risks and countermeasures

- Refactoring is regarded as an overhead activity
- Introducing failures with refactoring
- Outdated comments and documents

- Balance cost and benefits
- Better to start from day one! Not at the end of the project/delivery
- Should have sufficient and efficient regression tests

Relevant books

Old edition, available online at NTNU lib.

New edition

Remove Duplicated Code – Pull up method (Don’t repeat yourself (DRY))

- You have methods with identical results on subclasses.

Remove Duplicated code - Substitute Algorithm

- You want to replace an algorithm with one that is clearer.

```
String foundPerson(String[] people){

  for (int i = 0; i < people.length; i++) {
      if (people[i].equals ("Don"))
      { return "Don"; }

       if (people[i].equals ("John"))
       { return "John"; }

       if (people[i].equals ("Kent"))
       { return "Kent";  }
    }
    return "";
}
```

```
String foundPerson(String[] people){
     List candidates = Arrays.asList(new
String[] {"Don", "John", "Kent"});

  for (int i=0; i<people.length; i++)
       if (candidates.contains(people[i]))
              return people[i];

    return "";

}
```

Reduce size – shorten method/class

- **Long Method**
  - E.g., Extract method
- **Large Class**
  - E.g., Extract class, subclass, interface

Extract method example

If you have to spend effort looking at a fragment of code and figure out what it is doing, then you should extract it into a function/method and name it after "*what*."

```
void printOwing()
{
  printBanner(); //print details
  System.out.println ("name:
            " + _name);
  System.out.println ("amount
            " + getOutstanding());
}
```

```
void printOwing() {
        printBanner();
        printDetails(getOutstanding());
}

void printDetails (double outstanding) {
  System.out.println ("name:" + _name);
  System.out.println ("amount:" +
outstanding);
}
```

Reduce size – shorten parameter list

- **Long Parameter List**
  - E.g., Introduce Parameter Object

Divergent Change

- **Code smell**
  - One module is often changed in different ways for different reasons.
  - Classes have more than distinct responsibilities that it has more than one reason to change
  - Violation of single responsibility design principle

- **Refactoring**
  - You identify everything that changes for a particular cause and put them all together

```java
public class Account {
    private int accountNumber;
    private double balance= 0;

    public Account(int accountNumber){
        this.accountNumber = accountNumber;
    }

    public int getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

    public void credit(double amount) {
        balance += amount;
    }

    public void debit(double amount) {
        balance -= amount;
    }

    public String toXml() {
        return "<account><id>" + Integer.toString(getAccountNumber()) + "</id>" +
                "<balance>" + Double.toString(getBalance()) + "</balance></account>";
    }
}
```

More than one responsibility, operation of a bank account + serializing the account

https://www.slideshare.net/nadeembtech/code-craftsmanship

```java
public class Account {
    private int accountNumber;
    private double balance= 0;

    public Account(int accountNumber){
        this.accountNumber = accountNumber;
    }

    public int getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

    public void credit(double amount) {
        balance += amount;
    }

    public void debit(double amount) {
        balance -= amount;
    }
}
```

```java
public class AccountXMLSerializer {

    public String toXml(Account account) {
        return "<account>" +
                "<id>" + Integer.toString(account.getAccountNumber()) + "</id>" +
                "<balance>" + Double.toString(account.getBalance()) + "</balance>" +
                "</account>";
    }
}
```

https://www.slideshare.net/nadeembtech/code-craftsmanship

Shotgun Surgery

- **Code smell**
  - A single change is made to multiple classes simultaneously
  - When changes are all over the place, they are hard to find, and it’s easy to miss important changes.

Graph from refactoring.guru

- **Refactoring**
  - You put all the changes into a single class
  - Once and only once (OAOO)

```java
package com.example.codesmell;

public class Account {

	private String type;
	private String accountNumber;
	private int amount;

	public Account(String type,String accountNumber,int amount)
	{
		this.amount=amount;
		this.type=type;
		this.accountNumber=accountNumber;
	}


	public void debit(int debit) throws Exception
	{
		if(amount <= 500)
		{
			throw new Exception("Mininum balance shuold be over 500");
		}

		amount = amount-debit;
		System.out.println("Now amount is" + amount);

	}

	public void transfer(Account from,Account to,int cerditAmount) throws Exception
	{
		if(from.amount <= 500)
		{
			throw new Exception("Mininum balance shuold be over 500");
		}

		to.amount = amount+cerditAmount;

	}

	public void sendWarningMessage()
	{
		if(amount <= 500)
		{
			System.out.println("amount should be over 500");
		}
	}

```

When we add another criterion in the validation logic: if the account type is personal and the balance is over 500

We have to make changes to all methods

https://dzone.com/articles/code-smell-shot-surgery

```java
package com.example.codesmell;

public class AcountRefactored {

        private String type;
        private String accountNumber;
        private int amount;



        public AcountRefactored(String type,String accountNumber,int amount)
        {
                this.amount=amount;
                this.type=type;
                this.accountNumber=accountNumber;
        }

        private boolean isAccountUnderflow()
        {
              return amount<=500;

        }


        public void debit(int debit) throws Exception
        {
                if(isAccountUnderflow())
                {
                        throw new Exception("Mininum balance shuold be over 500");
                }

                amount = amount-debit;
                System.out.println("Now amount is" + amount);

        }

        public void transfer(AcountRefactored from,AcountRefactored to,int cerditAmount) throws Exception
        {
                if(isAccountUnderflow())
                {
                        throw new Exception("Mininum balance shuold be over 500");
                }

                to.amount = amount+cerditAmount;

        }
```

Menti: code refactoring performed?

Go to **menti.com**, code **6344 0068**

```java
public class Account {
    double principal,rate;   int daysActive,accountType;

    public static final int STANDARD=0, BUDGET=1,
            PREMIUM=2, PREMIUM_PLUS=3;
}

...

public static double calculateFee(Account[] accounts)
{
    double totalFee= 0.0;
    Account account;
    for (int i=0;i<accounts.length;i++) {
        account=accounts[i];
        if (account.accountType== Account.PREMIUM||
                account.accountType== Account.PREMIUM_PLUS)
            totalFee+= .0125 * (       // 1.25% broker's fee
                    account.principal* Math.pow(account.rate,
                            (account.daysActive/365.25))
                            -account.principal);   // interest-principal
    }

    return totalFee;

}

```

```java
/** An individual account.  Also see CorporateAccount. */

public class Account {

    private double principal;
    /** The yearly, compounded rate (at 365.25 days per year). */

    private double rate; /** Days since last interest pay out */
    private int daysActive;
    private Type type;
}

/** The varieties of account our bank offers. */
public enum Type {STANDARD, BUDGET, PREMIUM, PREMIUM_PLUS}


/** Compute interest. **/
public double interest() {
    double years = daysActive / 365.25;
    double compoundInterest = principal * Math.pow(rate, years);
    return compoundInterest - principal;
}

/** Return true if this is a premium account. **/
public boolean isPremium() {
    return accountType == Type.PREMIUM ||
            accountType == Type.PREMIUM_PLUS;
}

/** The portion of the interest that goes to the broker. **/
public static final double BROKER_FEE_PERCENT = 0.0125;

/** Return the sum of the broker fees for all the given accounts. **/
public static double calculateFee(Account accounts[]) {
    double totalFee = 0.0;
    for (Account account : accounts) {
        if (account.isPremium()) {
            totalFee += BROKER_FEE_PERCENT * account.interest();
            }
        }
    return totalFee;
    }
}
```

http://www.ifi.uzh.ch/seal/teaching/courses/softwareWartung-FS16/CodeReviewExercise.pdf*

Increase cohesion

- **Feature envy**
  - A function in one module spends more time communicating with functions or data inside another module than it does within its own module.
  - Move function to give it a dream home

https://waog.wordpress.com/2014/08/25/code-smell-feature-envy/

Increase cohesion (Cont')

- **Data clumps**
  - Bunches of data often hang around together
  - Consolidate the data together, e.g., Introduce Parameter Object or Preserve Whole Object

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan =
plan.withinRange(daysTempRange());
```

Primitive Obsession

- Primitive fields are basic built-in building blocks of a language, e.g., int, string, or constants

- Primitive Obsession is when the code relies too much on primitives and when uses primitive types to represent an object in a domain

```php
class contactUs
{
    public function addressUsa()
    {
        $address = new Array();
        $address['streetNo'] = 2074;
        $address['streetName'] = 'JFK street';
        $address['zipCode'] = '507874';

        return $address['streetName']. ' '. $address['streetNo']. ', '. $address['zipCode'];
    }

    public function addressGermany()
    {
        $address = new Array();
        $address['streetNo'] = '25';
        $address['streetName'] = 'Frankfurter str.';
        $address['zipCode'] = '80256';

        return $address['streetName']. ' '. $address['streetNo']. ', '. $address['zipCode'];
    }

    public function hotLine(){
        return '+49 01687 000 000';
    }
}
```

The address is defined as an array. Every time we need the address we will have to hard code it

https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318

```php
class address{

    private $streetNo;
    private $streetName;
    private $zipCode;

    public function addressUsa()
    {
        $this->streetNo = 2074;
        $this->streetName = 'JFK street';
        $this->zipCode = '507874';

        return $this->streetName. ' '. $this->streetNo. ', '. $this->zipCode;
    }
    public function addressGermany()
    {
        $this->streetNo = 25;
        $this->streetName = 'Frankfurter str.';
        $this->zipCode = '80256';

        return $this->streetName. ' '. $this->streetNo. ', '. $this->zipCode;
    }
}
```

We create a new class called Address
Every time we need to add/edit an address we hit the Address class

https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318

Code smells categories

- **Duplicated Code**
- **Long Method**
- **Large Class**
- **Long Parameter List**

Decrease size

- **Divergent Change**
- **Shotgun Surgery**

Localize change

- **Feature Envy**
- **Data Clumps**
- **Primitive Obsession**

Increase cohesion

- **Switch Statements**
- **Lazy Element**
- **Speculative Generality**
- **Temporary Field**
- **Message Chains**
- **Middle Man**

Simplify logic within and between objects

- **Data Class**
- **Refused Bequest**

Proper OO practice

Simplify logic within object – Switch statement

- Replace Conditional with Polymorphism

```
double getSpeed() {
  switch (_type) {
   case EUROPEAN:
     return getBaseSpeed();
   case AFRICAN:
     return getBaseSpeed() - getLoadFactor() ;
    case NORWEGIAN_BLUE:
      return  getBaseSpeed(_voltage);
    }
   throw new RuntimeException (“error");
}
```

Simplify logic within object – Switch statement (cont')

- Replace Parameter with Explicit Methods

```
void setValue (String name, int value) {

  if (name.equals("height")) {
      _height = value;
      return;
  }

  if (name.equals("width")) {
      _width = value;
     return;
  }

   Assert.shouldNeverReachHere();
}
```

```
void setHeight(int arg) {

  _height = arg;

 }

void setWidth (int arg) {

   _width = arg;

}
```

Simplify logic within object – Lazy Element

- E.g., Lazy class: A class that isn't doing enough to pay for itself should be eliminated

```
class Person {
    get officeAreaCode ()
     { return this._telephoneNumber.areaCode;}
    get officeNumber ()
     {return this._telephoneNumber.number;}
}

Class TelephoneNumber {
    get areaCode () { return this._areaCode;}
    get number () { return this._number;}
}
```

```
Class Person {
    get officeAreaCode ()
{return this._officeAreaCode;}

    get officeNumber () {return
this._officeNumber;}

}
```


NTNU

Simplify logic within object - Speculative Generality

- When you have code that isn't actually needed today.
- Such code often exist to support future behavior, which may or may not be necessary in the future
- Remove unnecessary delegation, unused parameters, dead code

```
Public class Customer {
    private String name;
    private String address;

    Public Customer (String name, String add)
    { this.name = name;
      this.address = add;
     }

      String GetName () {
            return name;
        }
  }
```

Simplify logic within object – Temporary Field

- Class has a variable which is only used in some situation
- Trying to understand why a variable is there when it doesn’t seem to be used

```php
class User
{
    private $id;
    private $name;
    private $priviledges;
    private $content;
    private $contactDetails;

    public function __construct()
    {
        $this->contactDetails = new UserContactDetails();
    }
    function notify($message)
    {
        $messageBody = 'Dear ' . $this->name;
        $messageBody .= '( ' . $this->contactDetails->getStreetNumber() . ' ' .
            $this->contactDetails->getStreetName() . ').';
        $messageBody .= $message;
        $notificationService->sendSMS($this->contactDetails->getPhoneNubmer(), $messageBody);
    }
    function delete()
    {
        systemDelete($this->id);
    }

    function update() {
        systemUpdate($this->id, $this->priviledges, $this->content);
    }
```

$name and $contactDetails are only used in the notify() method.

https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318

Why not pass them as a method parameters

```
function notify($userName, $contactDetails, $message) {

    $messageBody = 'Dear ' . $userName;
    $messageBody .= '( ' . $contactDetails->getStreetNumber() . ' ' .
        $contactDetails->getStreetName() . ').';
    $messageBody .= $message;

    $notificationService->sendSMS($contactDetails->getPhoneNubmer(), $messageBody);
}
```

https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318

Simplify logic between objects

- **Message Chains**
- **Middle Man**

Employee->EmployeeConfig->Config

```php
Class Employee{
    public function getConfiguration(){
        $this->employeeConfig->getConfiguration();
    }
}

Class EmployeeConfig{
    public function getConfiguration(){
        $this->config->getConfiguration();
    }
}

Class Config{
    public function getConfiguration(){
        $this->loadConfiguration();
    }
}
```

Employee->Config

```php
Class Employee{
    public function getConfiguration(){
        $this->config->getConfiguration();
    }
}
```

Proper OO practice

- **Data Class**
  - Make a public field private and provide accessors

```
public String _name
```

```
private String _name;
public String getName() {return _name;}
public void setName(String arg) {_name = arg;}
```

Proper OO practice (Cont')

- **Refused Bequest**
  - Only some of the inheritances are need, i.e., hierarchy is wrong
  - Inheritance makes no sense and the subclass really does have nothing in common with the superclass

https://refactoring.guru/smells/refused-bequest

Comments in code

- Good to have, can be simplified
- **Best if code is self-explanatory**
  - If you need a comment to explain what a block of code does, simply the code
  - If the method is already extracted but you still need a comment to explain what it does, rename the method
  - If you need to state some rules about the required state of the system, use Assertion

Python code refactoring

Python code smell* – code conventions

- Some are relevant to python code conventions** , e.g.,
  - Function names should comply with a naming convention
  - Class names should comply with a naming convention

- Tool support
  - Pylint (Checking structure of the code is compliant with PEP-8)

Pylint
Star your Python code!

* https://rules.sonarsource.com/python/type/Code%20Smell/RSPEC-1720

** https://www.python.org/dev/peps/pep-0008/

Pylint demo

https://www.youtube.com/watch?v=fFY5103p5-c

Python code smell*

- Remove duplications
  - Redundant pairs of parentheses should be removed
  - Using features of Python, e.g., Decorator**
- Reduce size
  - Functions, methods and lambdas should not have too many parameters
  - Lines should not be too long
- Speculative Generality, e.g.,
  - Nested blocks of code should not be left empty
  - Unused local variables should be removed

* https://rules.sonarsource.com/python/type/Code%20Smell/RSPEC-1720

**https://www.youtube.com/watch?v=n_Y-_7R2KsY

Python code smell* (cont’)

- Reduce complexity and simplify logics
  - Cognitive Complexity of functions should not be too high
  - Functions should not be too complex
  - Control flow statements "if", "for", "while", "try" and "with" should not be nested too deeply
  - Functions should not contain too many return statements

* https://rules.sonarsource.com/python/type/Code%20Smell/RSPEC-1720

Python code smell – tool support

E.g., PyCharm

Inspection and code review

What is inspection?

- Visual examination of software product

- Identify software anomalies
  - Errors
  - Code smells
  - Deviations from specifications
  - Deviations from standards
    - E.g., Java code conventions

    www.oracle.com/technetwork/java/codeconventions-150003.pdf

Why do we need inspection?

- Many software artifacts cannot be verified by running tests, e.g.,
  - Requirement specification
  - Design
  - Pseudocode
  - User manual

- Inspection reduces defect rates
- Inspection complements testing
- Inspection identifies code smells
- Inspection provides additional benefits

Inspection finds types of defects different from testing

Number of different types of defects detected by testing vs. inspection [1]

Some defect types	Testing	Inspection
Uninitialized variables	1	4
Illegal behavior, e.g., division by zero	49	20
Incorrectly formulated branch conditions	2	13
Missing branches, including both conditionals and their embedded statements	4	10

Inspection reduces defect rates

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

Source: Adapted from *Programming Productivity* (Jones 1986a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).

Testing vs. Inspection

	Testing	Inspection
Pros	• Can, at least partly, be automated • Can consistently repeat several actions • Fast and high volume	• Can be used on all types of documents, not just code • Can see the complete picture, not only a spot check • Can uses all information in the team • Can be innovative and inductive
Cons	• Is only a spot check • Can only be used for code • May need several stubs and drivers	• Is difficult to use on complex, dynamic situations • Unreliable (people can get tired) • Slow and low volume

Inspection provides additional benefits [2]

- Code and change understanding
- Knowledge transfer
- Increased team awareness
- Creation of alternative solutions to problems

Code review at Google [3]

- "All code that gets submitted needs to be reviewed by at least one other person, and either the code writer or the reviewer needs to have readability in that language. Most people use Mondrian to do code reviews, and obviously, we spend a good chunk of our time reviewing code."

  --Amanda Camp, Software Engineer, Google

Code reviews in OSS projects [4]

- Asynchronous and distributed review
  - Lightweight with tool support
  - Not limited to the existence of defects
  - Encourage discussing the best solution
- Frequent review
  - Most peer review happens within hours of completing a change
- Incremental review
  - Reviews should be of changes that are small, independent, and complete

The potential system benefit of accepting code that hasn't been discussed by a group of experts doesn't outweigh the risks. (Linus Torvalds)

Tools to support inspection

- Google’s Mondrian
- Facebook’s Phabricator
- GitHub
- ...

GitHub review tool

Software reading techniques

- Unstructured: Ad hoc reading
- Semi-structured: checklist-based reading
- Structured or systematic reading
  - Defect-based reading
  - Perspective-based reading

A sample (partial) checklist for requirement review [4]

1. Requirements specifications shall be testable.
2. Requirements specifications shall not conflict with other requirements specifications.
3. Conditional requirements specifications shall cover all cases.
4. Numerical values in requirements specifications shall include physical units if applicable.

A sample (partial) checklist for code review [4]

1. Have resources (e.g., memory, file descriptor, database connection) properly been freed?
2. Are shared variables protected/thread-safe?
3. Is logging implemented?
4. Are comments updated and consistent with the code?
5. Is data unnecessarily copied, saved, or reloaded?
6. Is the number of cores checked before spawning threads?

Checklist-based reading with active guidance

- Use a tailored checklist

- Guide reader how to use the checklist
  - Where to find?
  - To find what?
  - How to detect?

An example of a checklist with active guidance [4]

Where What How

Section	Feature	Checklist item
For each class	Inheritance	Q1. Is all inheritance required by the design implemented in the class?
		Q2. Is the inheritance appropriate?
	Constructor	Q3. Are all instance variables initialized with meaningful values?
		Q4. If a call to base class is required in the constructor, is it present?
For each method	Data referencing	Q5. Are all parameters used within a method?
		Q6. Are the correct class constants used?
		Q7. Are indices of data structures operating within the correct boundaries?

Best practice of checklists

- Should be periodically revised
- Should be concise and fit on one page
- Should not be too general
- Should not be used for conventions which are better checked or enforced with software tools

Unsystematic vs. Systematic reading

Defect-based reading

- Each reader is given specific responsibility to discover a particular class of defects

- Specific responsibility from defect taxonomy
  - Wrong or missing assignment
  - Wrong or missing data validation
  - Wrong timing or sequencing
  - Interface problems
  - Etc.

Perspective-based reading

- Each reader reviews software artifact from a particular perspective
- Then combine results from different perspectives
- Perspective examples
  - Novice user and expert user (GUI)
  - User, designer, tester (Requirements)

Template for perspective-based reading

- Introduction
  - Explain stakeholder’s interests
- Instruction
  - How to read and extract information
- Questions
  - For reviewers to answer by following the instruction

Novice use usability reading guideline [5]

- Introduction
  - You review the usability of the software to make PPT from the perspective of users who are new to use software to make slides.
- Instruction 1
  - Scan through menu, sub-menu, help info. in GUI
- Questions
  - Are there instructions or online help?
  - Are items in menu, sub-menu unambiguous in meaning?

Expert use usability reading guideline

- Introduction
  - You review the usability of the software to make PPT from the perspective of an expert who wants to use the software for making slides quickly.
- Instruction
  - Execute actions through finding and using shortcuts.
- Questions related to the instruction
  - Is it easy to find the frequently used shortcuts?
  - Is there any online help to guide the usage of shortcuts?

Benefits of perspective-based reading

- Focused
  - Particular coverage of part of the document
- Detailed
  - Instruction and questions
- Adaptable
  - To particular software artifacts
- Tailorable
  - To organizational and project setting

Effective factors affecting inspection [6]

- Individual expertise and training
  - Most important factor
- Amount of materials
  - Not to attempt to inspect too many materials in one cycle
  - E.g., less than 125 lines of code per hour
- Team size
  - 4 inspectors may not find significantly more defects than 2 inspectors
  - 4 or 2 usually find more defects than 1
- Process and tool support
  - With good tool support, inspection meeting may not be necessary

Summary

- We have studied
  - Why and how to do code refactoring
  - Why and how to perform code review

References

- [1] S.S. So et al. An empirical Evaluation of Six Methods to Detect Faults in Software, Software Testing, Verification, and Reliability, 12, 3, 2002, pp. 155-171.
- [2] A. Bacchelli and C. Bird: Expectations, outcomes, and challenges of modern code review. Proc. ICSE 2013.
- [3] CSE 403 course at University of Washington, available at https://courses.cs.washington.edu/courses/cse403/11sp/lectures/lecture15-reviews.pdf
- [4] Y. M. Zhu: Software reading techniques. Apress, 2016.
- [5] Z. Zhang, V. Basili, and B. Shneiderman: Perspective-based usability inspection: An empirical validation of efficacy. Journal of empirical software engineering, 4, 1999, pp. 43-69.
- [6] S. Kollanus and J. Koskinen: Survey of software inspection research, The Open Software Engineering Journal, 3, 2009, pp. 15-34.
- [7] D. Rombach et al. Impact of research on practice in the field of inspections, reviews, and walkthroughs, ACM SIGSOFT SE notes, 33, 6, 2008, pp. 26-35.

Software estimation – cost and quality

Mar 2022

Anh Nguyen Duc

Agenda

- The role of project estimation
- Traditional approaches for effort estimation
- Machine Learning in effort prediction
- Defect prediction models
- Within vs. Between project prediction
- Just-in-time defect prediction

What is a Project Estimate?

A declaration about needed

- cost and
- time
- quality
- human skill
- for delivering the project scope

How productive is this team/ developer in this project?

- What else?

Importance of project estimation

- ✓ Ressource allocation decisions
- ✓ Basis for the decision to start (or not to start)a project
- ✓ Foundation for project planning and set-up (business case)
- ✓ Foundation for project controlling
- ✓ If project time is a given,number of ressources can be determined
- ✓ Owner of an estimate is an indication about who is taking the project risk
- ✓ Decision and ressource allocation implications => Estimates are often part of political games
- ✓ Estimating is a core task of project management

Challenges for estimation

- ✓ Incomplete knowledge about:
  - ❖ *Project scope and changes*
  - ❖ *Prospective resources and staffing*
  - ❖ *Technical and organizational environment*
  - ❖ *Infrastructure*
  - ❖ *Feasibility of functional requirements*
- ✓ *Comparability of projects in case of new or changing technologies, staff, methodologies*
- ✓ *Learning curve problem*
- ✓ *Different expectations towards project manager*

Challenges for estimation

- ✓ Estimation is too low
  - ✓ *Scope and tasks(WBS) incomplete/unknown*
- ✓ Estimation is too high
  - ✓ *Political / human reasons*
  - ✓ *Learning curve*
- ✓ New technologies can make new parameters necessary

Challenges for estimation

Estimation uncertainty

Estimating Effort

Cost Estimation

- ✓ Effort costs (the dominant factor in most projects)
  - ✓ – salaries of engineers involved in the project
  - ✓ – costs of building, heating, lighting
  - ✓ – costs of networking and communications
  - ✓ – costs of shared facilities (e.g library, staff restaurant, etc.)
  - ✓ – costs of pensions, health insurance, etc.
- ✓ Other costs
  - ✓ – *Hardware and software costs*
  - ✓ – *Travel and training costs*

Cost Estimation

- ✓ Staff categories (based on experience, qualification and skills),for example:
  - ✓ teamlead, junior business analyst, senior business analyst, junior programmer, senior programmer, subject matter expert
- ✓ Cost rate: Cost per person per day
  - ✓ 2 alternatives for cost rate:
    - ✓ *Single cost ratef or all types(no differentiation necessary)*
    - ✓ *Assign different cost rates to different categories*
- ✓ Personnel cost = person days x cost rate

Estimating effort- Basic principles

Select an estimation model (or build one first)

Evaluate known information: project scope,resources, software process (for example documentation requirements),system components

Feed this information as parametric input data into the model
Model converts the input into an estimate about the effort

Basic of an Estimation model

Examples:

Data Input	Estimate
Size & Project Data	Effort & Schedule
System Model	Performance
Software Process	Cycle Time.

How to build an estimation model?

Calibrating the model

Top-Down and Bottom-Up Estimation

Two common approaches for estimations

Top-Down Approach

- Estimate effort for the whole project
- Breakdown to different project phases and work products

Bottom-Up Approach

- Start with effort estimates for tasks on the lowest possible level
- Aggregate the estimates until top activities are reached

Top-Down and Bottom-Up Estimation

Top-Down Approach

- Normally used in the planning phase when little information is available how to solve the problem
- Based on experiences from similar projects
- Not appropriate for project controlling (too high-level)
- Risk add-ons usual as result tends to be too low

Bottom-Up Approach

- Normally used after activities are broken down to task level and estimates for the tasks are available
- Result can be used for project controlling (detailed level)
- Smaller risk add-ons (tends to be too high)
- Often a mixed approach with recurring estimation cycles is used.

Estimation Techniques

- ✓ Expert judgement
- ✓ Estimation by analogy
- ✓ Parkinson's Law
- ✓ Pricing to win
- ✓ Lines of code
- ✓ Function point analysis
- ✓ Algorithmic cost modelling
- ✓ COCOMO

Expert judgement

= Guess from experienced people

- Mostly used top-down for the whole project, but also for some parts of a bottom-up approach
- Relatively cheap estimation method. Can be accurate if experts have direct experience of similar systems
- No better than the participants
- Result justification difficult
- Very inaccurate if there are no experts!

Estimation by analogy

(do you work on the same project before?)

- The cost of a project is computed by comparing the project to a similar project in the same application domain
- Accurate if project data available
- Impossible if no comparable project has been tackled. Needs systematically maintained cost database

Parkinson's Law

"work expands to fill the time allotted for its completion"

- The project costs whatever resources (people, money, time) are available
- No overspend
- System is usually unfinished

Pricing to win

The project costs whatever the customer has to spend on it

- You get the contract
- The probability that the customer gets the system he or she wants is small. Costs do not accurately reflect the work required

Line of Code (LOC)

Traditional way for estimating application size (FORTRAN and assembler -> line-oriented languages)

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

Line of Code (LOC)

Traditional way for estimating application size (FORTRAN and assembler -> line-oriented languages)

Advantage: Easy to do

Disadvantages:

- No standard definition for Line of Code (logical versus physical)
- Of no help given a written project scope or functional design
- You get what you measure: If the number of lines of code is the primary measure of productivity, programmers ignore opportunities of reuse
- Multi-language environments: Hard to compare mixed language projects with single language projects

The use of lines of code metrics for productivity should be regarded as professional malpractice (Caspers Jones).

Function Point Estimation

Based on FP metric for the size of a product

Based on the number of inputs (Inp), outputs (Out), inquiries (Inq), master files (Maf), interfaces (Inf)

Step 1: Classify each component of the product (Inp, Out, Inq, Maf, Inf) as simple, average average, or complex

- Assign the appropriate number of function points
- The sum of function pointers for each component gives UFP (unadjusted function points)

Function Point Estimation

- Step 2: Compute the technical complexity factor (TCF)
  - Assign a value from 0 (“not present”) to 5 (“strong influence throughout”) to each of 14 factors such as transaction rates, portability
  - Add the 14 numbers: This gives the total degree of influence (DI)
    - *$TCF = 0.65 + 0.01 \times DI$*
    - *The technical complexity factor (TCF) lies between 0.65 and 1.35*
- Step 3 The number of function function points (FP) is:
    - *$FP = UFP \times TCF$*

Function Point Estimation (example)

Function Point Estimation (example)

Mapping Functions to Transaction Types

Add Customer Change Customer Delete Customer Receive payment Deposit Item Retrieve Item Add Place Change Place Data Delete Place	External Inputs
Print Customer item list Print Bill Print Item List	External Outputs
Query Customer Query Customer's items Query Places Query Stored Items	External Inquiries

Function Point Estimation (example)

Calculate the Unadjusted Function Points

Function Type	Number		Weight Factors simple	average	complex		
External Input (EI)		x	3	4	6	=	
External Output (EO)		x	4	5	7	=	
External Queries (EQ)		x	3	4	6	=	
Internal Datasets (ILF)		x	7	10	15	=	
Interfaces (EIF)		x	5	7	10	=	
					Unadjusted Function Points (UFP)	=	

The unadjusted function points are adjusted with general system complexity (GSC) factors

Function Point Estimation (example)

After the GSC factors are determined, compute the

Value Added Factor (VAF): $\mathbf{VAF} = 0.65 + 0.01 * \sum_{i=1}^{14} GSC_i$ $\quad GSC_i = 0,1,\ldots,5$

Function Points (FP) = Unadjusted Function Points (UFP)* VAF

PF (Performance factor) = Number of function points that can be completed per day

Effort = FP / PF

The unadjusted function points are adjusted with general system complexity (GSC) factors

Function Point Estimation (challenges)

- Complete description of functions necessary
  - Often not the case in early project stages -> especially in iterative software processes
- Internal functions (algorithms) rather underestimated, as model is based on user oriented requirements and functions
- Only complexity of specification is estimated
  - Implementation is often more relevant for estimation
- High uncertainty in calculating function points:
  - Weight factors are usually deducted from past experiences (environment, used technology and tools may be out-of-date in the current project)
- Not suitable for project controlling.

Algorithmic cost modelling

- Most models before 2005 are basing on some form of parametric formulas
- After 2005, more advanced prediction and classification approaches

- 65-85: the search for right parametric forms
- 85-95: advances in size and complexity metrics
- 95-05: proliferation of software development styles
- 05-15: advances in prediction and classification models

Belady and Lehman (1972)

$$\text{Effort} = p + K^{c-d}$$

- *Effort*: total maintenance effort
- *p*: productive effort, including analysis, design, code testing
- *d*: degree of maintenance team familiarity with the software
- *c*: complexity caused by lack of structured design and document
- *K*: empirical constant, depends on the environment

The COCOMO model

- Effort (person month) is a function of size (LOC)
- Exists in three stages
  - Basic - Gives a 'ball-park' estimate based on product attributes Intermediate
  - Modifies basic estimate using project and process attributes
  - Advanced - Estimates project phases and parts separately

Effort = b *x* Size ^{c}

- Organic: small teams develops software in known environment (*b*=2.4, *c*=1.05)
- Embedded: inflexible and constrained environment (*b*=3.6, *c*=1.20)
- Semidetached: varying levels of team experience working on larger projects (*b*=3.0, *c*=1.12)

Effort = b x Size ^{c} x EAF

- Intermediate model
- b, c: calibrated factors
- EAF: effort adjustment factor

COCOMO suite of models (1981-2007)

Advantages

- Repeatable estimations
- Easy to modify input
- Easy to customize and refine formula

Disadvantages

- Subjective inputs
- Unable to deal with exceptional conditions
- Mainly designed for waterfall
- Needs historical data for calibration

Figure from Boehm et al. 2008

Machine Learning-based Estimating Effort

Types of ML techniques

Case-Based Reasoning (CBR)

Decision Trees (DT)

Bayesian Networks (BN)

Support Vector Regression (SVR)

Genetic Algorithms (GA)

Genetic Programming (GP)

Association Rules (AR)

Artificial Neural Networks (ANN)

J. Wen, S. Li, Z. Lin, Y. Hu, and C. Huang, “Systematic literature review of machine learning based software development effort estimation models,” *Information and Software Technology*, vol. 54, no. 1, pp. 41–59, Jan. 2012

Case-Based Reasoning (CBR)

- Finding a similar past case, and reusing it in the new problem situation
- CBR is the most widely used method in the software estimation practice

Similarity measure:

- Manhattan distance
- Euclidean distance
- Minkowski distance
- grey relational coefficient
- Gaussian distance
- Mahalanobis distance

ID	Feature	Full name or explanation
1	TeamExp	Team experience in years
2	ManagerExp	Project manager's experience in years
3	YearEnd	Year of completion
4	Length	The length of project
5	Transactions	Number of transaction processed
6	Entities	Number of entities
7	PointsAdjust	Adjusted function points
8	PointsNonAjust	Unadjusted function points
9	Envergure	Complex measure derived from other factors
10	Effort	Measured in person-hours

Jorgensen M, Shepperd M (2007) A systematic review of software development cost estimation studies. IEEE Trans Softw Eng 33:33–53
E. Rashid, S. Patnaik, and V. Bhattacherjee, "Software Quality Estimation using Machine Learning: Case-based Reasoning Technique," *International Journal of Computer Applications*, vol. 58, no. 14, pp. 43–48, Nov. 2012.

Decision Trees (DT) / Classification and regression trees (CART)

- relatively easy to understand and are also very effective
- DT splits the original dataset to sub datasets which are more homogeneous.
- Recursive Binary Splitting: *all the features are considered and different split points are tried*

When to stop splitting?

Jorgensen M, Shepperd M (2007) A systematic review of software development cost estimation studies. IEEE Trans Softw Eng 33:33–53
E. Rashid, S. Patnaik, and V. Bhattacherjee, "Software Quality Estimation using Machine Learning: Case-based Reasoning Technique," *International Journal of Computer Applications*, vol. 58, no. 14, pp. 43–48, Nov. 2012.

Support Vector Machine (SVM)

- Builds a model that assigns new case to one category or the other (binary classifier)
- Map training cases to points in space to maximise the width of the gap between the two categories.
- A new case is then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

H_1 does not separate the classes.
H_2 does, but only with a small margin.
H_3 separates them with the maximal margin.

Artificial Neural Network (ANN)

- Input layers: size of the software in function points or Lines Of Code (LOC), and other effort drivers such as complexity of the software, database size, experience etc.
- Each node has its own linear regression model, compose of input and output variables

K. K. Gebretsadik and W. T. Sewunetie, “Designing Machine Learning Method for Software Project Effort Prediction,” *Computer Science and Engineering*, vol. 9, no. 1, pp. 6–11, 2019.

Deep Learning

- Model for Agile projects, predict efforts for the next Iteration
- Adopt Long Short Term Memory / Recurrent Neural Network
- Input: SP: story points, TD length: the number of words in the title and description of an issue, LOC: line of code

M. Choetkiertikul, H. K. Dam, T. Tran, T. Pham, A. Ghose, and T. Menzies, "A Deep Learning Model for Estimating Story Points," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 637–656, Jul. 2019

Defect Prediction

Importance of defect prediction

- Software defect: flaw or imperfection in software work or software process
- A defect is also referred as a fault or a bug
- Every software project has bugs, which we hope to reduce.
- Defect prediction is essential in the field of software quality and software reliability
- Focus on predicting those defects that affect project and product performance

Urgent Severe	Urgent Not Severe
Not Urgent Severe	Not Urgent Not Severe

Pareto rule (20-80 rule)

Most defects (80%) exist in few modules (20%)

Defect data is usually imbalanced

Early days prediction models - COQUALMO

- predicting number of residual defects/KSLOC (Thousands of Source Lines of Code) or defects/FP (Function Point) in a software product
- Base on COCOMO II model

Algorithmic approaches - Basic principles

Software defect prediction – mainly base on historical data

When to use: (1) use for planning (2) in-process monitoring of defect discovery numbers

What to predict?

- Number of defect
- Defect density
- Defect proneness

Software entity	Software attribute	Software metrics	References	# of Refs
Product	Size	LOC or number of use cases	S2, S37, S15, S21, S49, S32, S47, S35, S27, S13, S42, S34, S10, S19, S41, S16, S36,	17
		Size of artifact	S1, S13	2
		Code metrics: size metrics from NASA projects (Halstead size metrics)	S50, S33, S21, S28, S11, S12, S3, S9, S4, S25	10
		Requirement metrics: action, conditional, continuance, imperative, incomplete, option, risk level, source, weak phrase	S33, S28, S11, S12, S3, S9, S4, S20, S25	9
	Structure	McCabe Metrics (Complexity etc.)	S52, S21, S17, S47, S42, S44, S30, S39, S22,	12
		Halstead Metrics (total number of operators, operands etc.)	S7, S48, S19	
		Object-oriented Metrics (Complexity, Length, Coupling, Cohesion, Modularity, Reuse)	S51, S50, S29, S18, S14, S6, S16, S36, S19	9
		Design metrics from UML (e.g. Chidamber and Kemerer (1994))		
		Data flow complexity (DC), cyclomatic complexity (CC)	S8, S24, S43	3
		Requirements complexity (RC), Complexity of new functionality	S2, S37, S32, S35, S34, S48	6
		Program dependencies	S38	1
		Design metrics: edge count, node count, branch count, decision count, multiple condition count and condition count, densities, complexities	S20	1
		Architectural design metrics to quantify SDL (Specification and Description Language) blocks	S45	1
		Design, review or development effort measured in person hour	S1, S5, S23, S37, S31, S40, S35, S13, S43	9
		Creation effort, review effort	S26, S46	2
		Design review effectiveness (DRE)	S30, S7	2
		Review, inspection and walkthrough (RIW)	S2, S8, S32, S24, S34, S30, S7, S10, S48	9
		Total months of the project duration	S15, S31, S41	3
		Requirements stability (RS), requirement change request (RCR)	S2, S37, S17, S8, S32, S35, S27, S24, S34, S43, S30, S7, S10, S48	14
		Capability Maturity Model Integration (CMMI) Level	S2, S37, S8, S32, S40, S35, S24, S34, S30, S7,	10
		Number of detected defects from review	S1, S37, S15, S35, S13	5
		Requirement fault density (RFD), design defect density (DDD), fault days number (FDN), code defect density (CDD),	S15, S17, S8, S27, S24, S10, S48,	7
		Analysis, design, review quality	S37, S40, S35, S43	4
		Quality of documented test cases (QDT)	S35, S7, S41	3
		Defined process followed (DPF)	S2, S32, S34, S35, S37	5
		Number of stakeholders/members	S15, S49, S41	3
		Development language	S37, S15	2
		Configuration management	S37, S35, S41	3
		Project planning	S37, S35	2
		Scale of distributed communication	S37, S35	2
		Vendor management	S37, S35	2
		DBMS type, development solution, industry area	S15, S41	2
		Techno complexity	S26, S49, S46	3
		Urgency	S46	1
		Novelty to developer	S49	1
	Human characteristics	Staff experience	S2, S37, S32, S40, S35, S27, S34, S43, S7, S10	10
		Staff motivation	S37, S35	2
		Programmer capability	S37, S35, S30, S7	4
		Staff training quality	S37, S35	2
		Internal communication/interaction	S37, S35	2
		Productivity	S15	1
		Practitioners level	S26, S46	2
		Stakeholder involvement	S2, S32, S34	3
		People dependence	S41	1

R. Özakıncı and A. Tarhan, “Early software defect prediction: A systematic map and review,” *Journal of Systems and Software*, vol. 144, pp. 216–239, Oct. 2018, doi: 10.1016/j.jss.2018.06.025.

With-in vs. Cross-project prediction

due to the diversity in development processes, a defect prediction model is often not transferable and requires to be rebuilt when the target project changes

Critiques on defect prediction

- the unknown relationship between defects and failures
- problems with the “multivariate” statistical approach
- problems of using size and complexity metrics as sole “predictors” of defects
- problems in statistical methodology and data quality
- false claims about software decomposition and the “Goldilock’s Conjecture”

A typical ML model for defect prediction

History of ML/AI approaches in defect prediction

S. Omri and C. Sinz, “Deep Learning for Software Defect Prediction: A Survey,” in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, New York, NY, USA: Association for Computing Machinery, 2020, pp. 209–214. Accessed: Mar. 19, 2022. [Online]

Just-in-time defect prediction

Fixing Commit Message

Revert "Make VisibleRefFilter.Filter reuse the refs passed from JGit."

This reverts commit b032a529f83892dfbdfb375c47a90d89756dd8ab. This commit introduced an issue where tags were not replicated under certain circumstances.

Bug: Issue 2500
Bug: Issue 1748
Change-Id: I9c902b99c7f656c7002cf3eab9e525f22a22fb85

Defective Commit: b032a529f83892dfbdfb375c47a90d89756dd8ab

2 gerrit-server/src/main/java/com/google/gerrit/server/git/VisibleRefFilter.java

```
103 103   if (!deferredTags.isEmpty() && (!result.isEmpty() || filterTagsSeperately)) {
104 104     TagMatcher tags = tagCache.get(projectName).matcher(
105 105         tagCache,
106 106         db,
107     -       filterTagsSeperately ? filter(db.getAllRefs()).values() : result.values());
    107 +       filterTagsSeperately ? filter(refs).values() : result.values());
108 108     for (Ref tag : deferredTags) {
109 109       if (tags.isReachable(tag)) {
110 110         result.put(tag.getName(), tag);
111 111       }
112 112     }
113 113   }
```

Fixing Commit: 6db280663f836096c30a9626e7170f4a36d8cc1f

2 gerrit-server/src/main/java/com/google/gerrit/server/git/VisibleRefFilter.java

```
112 112   if (!deferredTags.isEmpty() && (!result.isEmpty() || filterTagsSeperately)) {
113 113     TagMatcher tags = tagCache.get(projectName).matcher(
114 114         tagCache,
115 115         db,
116     -       filterTagsSeperately ? filter(refs).values() : result.values());
    116 +       filterTagsSeperately ? filter(db.getAllRefs()).values() : result.values());
117 117     for (Ref tag : deferredTags) {
118 118       if (tags.isReachable(tag)) {
119 119         result.put(tag.getName(), tag);
120 120       }
121 121     }
122 122   }
```

Technical Debt Management in Visma

Mili Orucevic
Chief Software Quality Engineer

VISMA

Agenda

- Technical Debt in General
- How are we working with Technical Debt in Visma?
- Dive into some examples
  - Assessment
  - Dashboards
  - Indexes
  - Example of large Technical Debt
- Research: Why Technical Debt Matter?

VISMA

Mili Orucevic

Chief Software Quality Engineer

https://www.linkedin.com/in/milio

VISMA

Lehman’s Law of Software Evolution

Law of Continuing Change

“A system must be continually adapted or it becomes progressively less satisfactory”

Law of Increasing Complexity

“As a system evolves, its complexity increases unless work is done to maintain or reduce it.”

VISMA

Technical Debt definition

"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The **danger occurs when the debt is not repaid.** Every minute spent on **not-quite-right code** counts as **interest on that debt**. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object- oriented or otherwise"

1992 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)

Ward Cunningham

"In software-intensive systems, technical debt consists of design or implementation constructs that are **expedient in the short term** but that set up a technical context that **can make a future change more costly or impossible**. Technical debt is a contingent liability whose impact is limited to internal system qualities—primarily, but not only, maintainability and evolvability."

Kruchten, Nord, Ozkaya (p.5)

Managing Technical Debt

Software systems are prone to the **build up of cruft** - deficiencies in internal quality that **make it harder than it would ideally be to modify and extend** the system further. Technical Debt is a metaphor, coined by Ward Cunningham, that frames how to think about dealing with this cruft, thinking of it like a financial debt. The extra effort that it takes to add new features is the interest paid on the debt.

Blogpost, 21.05.2019:
https://martinfowler.com/bliki/TechnicalDebt.html

Martin Fowler

VISMA

Technical Debt Landscape

←← Visible →→

←← Mostly Invisible →→

←← Visible →→

Business

New Features

Additional Functionality

Architecture
Architecture smells
Pattern Violations
Structural Complexity

Code
Code Complexity
Code Smells
Coding Style Violations
Low Internal Quality

Production Infrastructure
Build, Test and Deploy Issues

Defects

Low External Quality

End Users

Evolvability

Maintainability

Product Roadmap

Internal Team Work

End Product

VISMA

Cause, effect and management

VISMA

Visualizing Technical Debt

Possible symptoms of Technical Debt

- Decline in number of releases
- Increase in delivery lead time
- Less and less time for developing new features

- Might treat symptoms instead of real issues

VISMA

Measuring Technical Debt

Static Code Analysis

- 18626 days or 51 years of technical debt, for 1106 projects
  - ~16 days effort per project
- First 10 days of March, 51 days of effort of technical debt was introduced
  - 7 min per project per day

Other examples outside Visma

- Project: 15 year old code based
- Technical debt accumulated: 4000 years

Should not be the only way to measure technical debt

VISMA

How are we creating awareness and work with Technical Debt in Visma?

VISMA

Visma - a SaaS company

14 000
Engaged employees

With more than **6 500** developers

Since 2014 more than **140 companies** have joined Visma

1 135 000
Customers

We are where you are
Strong local presence with more than **150** locations

Running through Visma's systems in September '21:

10,3 million payslips
23,2 million invoices

What did we want to solve?

Raise the **awareness** of technical debt in teams

Create **visibility** of technical debt

Common **process** for dealing with technical debt

Keep a technical debt **register** for teams

Ensure that technical debt is **planned** and **repaid** where reasonable

VISMA

Technical Debt Maturity Assessment

Technical Debt Self-assessmen

TDM activity / TDM levels	TD Repayment	TD Communication	TD Prioritization
Organized (level 3)	☑ Continuous refactorin with monthly assigne of the development t	Continuous communication about TD issues in meeting with all stakeholders.	☑ Prioritization is done based on specific calculated numbers. Recommended to use Severity as the calculated number
Semi organized (level 2)	☑ Refactoring during normal development previously identified issues.	☑ Product/Service Owner communicate development issues to business stakeholders. Communication must describe customer value.	☑ Prioritization is done with basic low/medium/high scaling that is based on hunches and rough estimations.
Unorganized (level 1)	Refactoring only when the problem is impossible to avoid or no refactoring at	TD is not part of development discussions and often done only in coffee table discussions.	Prioritization is not conducted and decisions are done based on a hunch.
Responsibility for activity	Development team, softw architect(s)	Development team, software architect(s), team manager, product/service owner	Software architect(s), team manager, product/service owner

Technical Debt Maturity Model

Prevention

Primary responsible roles: Architects, developers, and anyone ese writing code

Unorganized (level 1)	The team has not agreed on coding standards and does not perform code reviews
Semi organized (level 2)	☐ Following the team coding standards and performing code reviews is optional or ad hoc
Organized (level 3)	☐ The team follows coding standards and performs code review for all non-trivial changes

Identification

Primary responsible roles: Architects, developers and anyone else that witnesses technical debt

Unorganized (level 1)	The team does not identify any technical debt during the development process, except when it happens accidentally
Semi organized (level 2)	☐ The team identifies technical debt during implementation and code reviews
Organized (level 3)	☐ The team identifies technical debt during implementation and code reviews ☐ The team identifies technical debt using static code analysis tools (e.g. ReSharper, SonarQube, Coverty) ☐ The team identifies technical debt as part of operational processes like incident reviews and problem management

Documentation

Primary responsible roles: Architects and developers

Unorganized (level 1)	The team does not document any technical debt, or documents it in a separate backlog
Organized (level 3)	☐ Technical debt is documented in the same backlog as everything else

Analysis

Primary responsible roles: Architects and developers

Unorganized (level 1)	Technical debt is not analyzed
Semi organized (level 2)	☐ Technical debt is analyzed and summarized using the Priority field in JIRA
Organized (level 3)	☐ Technical debt is analyzed and a severity score is calculated and documented in JIRA

Monitoring

Primary responsible roles: Product/Service Owner and Architects

Unorganized (level 1)	Technical debt is not monitored
Semi organized (level 2)	☐ Technical debt is measured over time in JIRA by looking at the number of technical debt issues
Organized (level 3)	☐ Different types of technical debt metrics is measured over time by looking at the sum of all severity scores and metrics in SonarQube

Communication

Primary responsible roles: Product/Service Owner, architects and developers

Unorganized (level 1)	The team rarely talks about technical debt internally (in the team) or externally (with stakeholders)
Semi organized (level 2)	☐ The team frequently talks about technical debt internally (in the team)
Organized (level 3)	☐ The team continuously communicates technical debt risks to stakeholders, focusing on the business and/or customer value of paying down the technical debt with the hig score. (Using the Architect Maturity index and a list of technical debt issues in JIRA sorted by the severity score are two ways of doing this.)

Planning

Primary responsible roles: Product/Service Owner and architects

Unorganized (level 1)	The team does not have a well-defined process for deciding what technical debt should be paid down next
Semi organized (level 2)	☐ The team decides on what technical debt should be paid down next based on the value of the Priority field in JIRA
Organized (level 3)	☐ The team decides on what technical debt should be paid down next based on the severity scores

Repayment

Primary responsible roles: Product/Service Owner

Unorganized (level 1)	The team only pays down technical debt when it becomes a blocking issue
Semi organized (level 2)	☐ The team performs sufficient refactoring as part of the normal backlog prioritization and development process
Organized (level 3)	☐ Some of the team's capacity is always reserved for paying down technical debt (20% or more is the industry standard and highly recommended)
Superorganized (level 4)	☐ The team has instituted a maximum cap on technical debt, and will dedicate 100% of their capacity to paying down technical debt whenever the agreed cap is exceeded

Inspired by research done at the Lappeenranta University of Technology by Jesse Yli-Huumo.

VISMA

Technical Debt Maturity Assessment

Latest version

Category	Requirements
Prevention	☐ The team follows coding standards and performs code review for all non-trivial changes ☐ The intention of implementing any workaround or corner-cutting is discussed within the team before any code is written; team decides together if the workaround will be implemented or more time will be allocated to avoid it › More Info
Identification	☐ The team identifies technical debt during implementation and code reviews ☐ The team identifies technical debt using static/dynamic code analysis tools (e.g. ReSharper, SonarQube, Coverity, Snyk) and adds it in the backlog. › More info ☐ The team identifies technical debt as part of operational processes like incident reviews and problem management ☐ A process to identify outdated dependencies is implemented (e.g Dependabot, dotnet-outdated)
Documentation	☐ Technical debt is documented in the same backlog as everything else and the NFR label is used ☐ The team documents tech debt about to be introduced (intentional) technical debt › More Info ☐ Our top 3 TD issues are logged and documented in a way that are understandable for stakeholders and decision makers as well. Business impact, if relevant, is part of the documentation. › More Info
Analysis	☐ Technical debt is analyzed and a severity score is calculated and documented in JIRA › Why Severity Scores? ☐ The most important technical debt for our product is reflected in the top 3 TD issues logged in Jira
Monitoring	☐ All types of technical debt is visible in Jira by looking at the sum of severity scores. Here are suggestions for TD JIRA Dashboard / TD Dashboard › Click here to expand...
Communication	☐ The team continuously communicates technical debt risks to stakeholders and decision makers, focusing on the business and/or customer value of paying down the technical debt with the highest severity score. (Using the ArchTech Maturity Index and a list of technical debt issues in JIRA sorted by the severity score are two ways of doing this.) › Why is it important to communicate and be transparent about he technical debt?
Planning	☐ The team includes technical debt decisions and planning into their periodic processes. › Intention and More Info ☐ The team decides on what technical debt should be paid down next based on the severity scores
Repayment	☐ Some of the team's capacity is always reserved for paying down technical debt (20% or more is the industry standard and highly recommended)

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Technical Debt Maturity Assessment

Category	Description
Prevention	How is introduction of unintentional technical debt prevented?
Identification	How, and where is technical debt identified?
Documentation	How is technical debt documented and labeled?
Analysis	How is technical debt analyzed with emphasis on risk likelihood, impact and severity?
Monitoring	How is technical debt monitored over time, f.ex. trends, top TD issues
Communication	How is technical debt communicated outside the team, f.ex. towards stakeholders?
Planning	How is technical debt included and prioritized in the improvement plans?
Repayment	How is technical debt repaid?

VISMA

Example assessment

Link to assessment

Dashboard - team A

Filter Results: TDMA Example

T			P	Risk Impact	Risk Likelihood	Risk Severity ↓
			=	3	3	9
			=	3	1	3
			=	1	1	1
			=	1	1	1
			=	1	1	1
			=	1	1	1
			≪	1	1	1
			=	1	1	1
			=	1	1	1
			=	1	1	1
			=	1	1	1
			=	1	1	1

1–12 of 12

VISMA

Dashboard - team B

Filter Results: TDMA Example 2

T	Key	Summary	P	Risk Impact	Risk Likelihood	Risk Severity ↓
			=			
			=			
			^			
			=			
			=			
			=	4	4	16
			=	4	4	16
			=	3	5	15
			=	4	3	12
			^	3	4	12
			=	3	3	9
			=	3	3	9
			=	3	3	9
			^	3	3	9
			=	3	3	9
			=	3	3	9
			=	4	2	8
			=	4	2	8
			=	2	3	6
			=	3	2	6
			=	2	3	6
			=	2	3	6
			=	2	3	6
			=	2	3	6
			=	2	3	6
			=	2	2	4
			=	2	2	4
			=	2	2	4
			=	2	2	4
			=	2	2	4
			=	2	2	4
			=	2	2	4
			=	2	2	4
			=	2	2	4
			v	3	1	3
			v	3	1	3
			v	2	1	2

Examples from SonarQube

QUALITY GATE STATUS

Passed

All conditions passed

MEASURES

New Code
Since November 10, 2020

Overall Code

QUALITY GATE STATUS

Failed

1 conditions fai

On New Code

1 N

MEASURES

New Code

Overall Code

QUALITY GATE STATUS

Failed

1 conditions failed

On New Code

1 New Critical Issues is greater than 0

MEASURES

New Code
Since 1.0.217-SNAPSHOT
Started 6 hours ago

Overall Code

11 Bugs

4 Vulnerabilities

0 Security Hotspots

13d Debt

340 Code Smells

Maintainability A

77.7% Coverage on 17k Lines to cover

994 Unit Tests

2.0% Duplications on 41k Lines

53 Duplicated Blocks

Conditions

Conditions on New Code

Conditions on New Code apply to all branches and to Pull Requests.

Metric	Operator	Value
Coverage	is less than	60.0%
Duplicated Lines (%)	is greater than	3.0%

Conditions on Overall Code

Conditions on Overall Code apply to branches only.

Metric	Operator	Value
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	2.0%
Maintainability Rating	is worse than	A
Reliability Rating	is worse than	A
Security Rating	is worse than	A
Unit Test Errors	is greater than	0
Unit Test Failures	is greater than	0

VISMA

Examples from SonarQube

'System.Exception' should not be thrown by user code.
Code Smell

'System.Exception' should not be thrown by user code.
Code Smell

Refactor this code to not nest more than 3 control flow statements.
Code Smell

'System.Exception' should not be thrown by user code.
Code Smell

Remove the unused local variable 'request'.
Code Smell

Fix this implementation of 'IDisposable' to conform to the dispose pattern.
Code Smell +2

```
67
68          }
69
70          private async Task<A[redacted]Name)
71          {
72              [redacted]
73              var request = new GetSecretValueRequest {SecretId = secretName};
```

Remove the unused local variable 'request'. Why is this an issue? last year L73
Code Smell Major Open [redacted] 5min effort Comment unused

Unused local variables should be removed

Unused local variables should be removed

Code Smell Minor unused Available Since Aug 14, 2015 SonarQube (C#) Constant/issue: 5min

If a local variable is declared but not used, it is dead code and should be removed. Doing so will improve maintainability because developers will not wonder what the variable is used for.

Noncompliant Code Example

```
public int NumberOfMinutes(int hours)
{
  int seconds = 0;   // seconds is never used
  return hours * 60;
}
```

General exceptions should never be thrown | Utility classes should not

VISMA

“Not our fault”

Example of a bigger technical debt

- Unintentional
- External factors
- Time

“Move from AngularJS to new version of Angular”

~ 1583 hours = 66 days

VISMA

Automatically track deviations

Visma Index

VISMA

Visma Index

VISMA

Visma Index

A&T - Technical Debt Maturity Assessment (TDMA)

1 . Technical Debt Maturity Assessment not done or report older than 12 months	Confluence Expiration date 3 months ago	1	1	1		
2 . Technical Debt Maturity Assessment not done or report older than 18 months	Confluence Expiration date in 3 months	0	450	0		
3 . Unresolved critical issues from the Tech Debt Assessment older than 30 days	Jira	0	200	0		
4 . Unresolved high issues from the Tech Debt Assessment older than 90 days	Jira	1	100	100		
5 . Unresolved medium issues from the Tech Debt Assessment older than 90 days	Jira	0	20	0		
6 . Unresolved low issues from the Tech Debt Assessment older than 90 days	Jira	9	1	9		
				110		

VISMA

Why Technical Debt Matter?

Lessons learned from own research

VISMA

Technical debt matters!

Products with **more technical debt** are developed by teams with **lower customer-centricity and innovation** scores.

Presumably, it's difficult to have time and energy for being innovative and customer-centric when you have work with and around old technology and practices.

VISMA

Technical debt matters!

The reduced innovation and customer-centricity in products with high technical debt drives down customer satisfaction and revenue growth.

1. Product NPS by technical debt (quartiles)

2. CBV Growth Rate (%) by technical debt (quartiles)

1. In general, products with less technical debt are more successful

- Higher product NPS
- Higher product growth rate

2. Product lines in the bottom 25% of technical debt have higher growth rate than expected. These product lines tend to have lower CBV, and could be smaller products that have accumulated technical debt through rapid growth.

VISMA

Entrepreneurial

Responsible

Dedicated

Inclusive

Make progress happen

VISMA

TDT 4242 Advanced Software Engineering

- Spring 2022

About Jingyue Li (Bill)

- Jingyue Li (Bill)
  - Master (Computer science) in China
  - Architect: IBM China Ltd.
    - Banking solutions
  - PhD and Post-Doc (Software engineering) at IDI, NTNU
  - Principal researcher: DNV Research & Innovation
    - Software engineering
    - Dependability of critical software systems
  - Teaching experience:
    - Software architecture, IT projects, software security and data privacy, advanced software engineering, empirical software engineering
    - Started teaching this course from 2017
  - Jingyue.li@ntnu.no

About Daniela

- Co-Responsible teacher: Daniela Cruzes
- Master (Computer science) in Brazil
- Dr. Ing (Software engineering) in Brazil
- Post-Doc in USA and at IDI
- Tester: CPqD.
- Billing solutions for Telecom.
- Software Security Officer: FARA.
- Transport Intelligent Systems
- Senior Research Scientist: SINTEF
- Professor of Software Engineering:
- NTNU
- Lead Security Researcher : VISMA

- Email: dcruzes@ntnu.no

About Anh Nguyen-Duc

- Occupation & Experience
  - Assoc. Prof. at NTNU, Assoc. Prof. At USN
  - Entrepreneurs (Muml, VVN, etc), Startup mentors (VVNOR, Klikkit, Mitanbud, Comartek AS, etc)
- Research interest:
  - Software startups & software ecosystem
  - Secure software development, Static analysis
  - AI for SE, SE for AI, AI ethics
- Education
  - Master (Software Engineering) in Sweden & Germany
  - PhD and Post-Doc (Software engineering) at IDI, NTNU
- Teaching experience:
  - Customer driven project (TDT4290), Software engineering (TDT4140), IT project (IT2901), Web development & HCI (WEB1000), Software process (SYS1000), Project management (PRO1000, MIS405), Green IT (MSM4103), Product Process and People (MS4102)
- Contact: anhn@ntnu.no or angu@usn.no

Outline

- ➢ Knowledge to learn to be a good software engineer
- What advanced SE knowledge to be covered in this course?
- How is this course organized?

Goals of the software methodology – Ensure value creation

- Ensure economic gain
- Ensure personal gain
- Ensure organizational (business) gain
- Ensure societal gain

- Phases of value achievement
  - Plan for value achievement. Business case and value realization plan
  - Perform change
  - Implement change and realize the value

Categories of software based on functions

- Application software (web, mobile, IoT, control SW)
- System software (OS, DB, …)
- Computer programming tools (compiler, MATLAB, etc.)
- …

Project types

- In-house development:
  - The same company makes requirements, develops, and uses the system
- Custom development:
  - One company is customer, and
    - Another company analyzes requirements and develops system
    - Another company analyzes requirements, yet another develops
  - May include outsourcing to subcontractors
- Mass-market software
  - Project to buy or rent package (SaaS)
  - Project to develop an adaptation of package (e.g., ERP-system)
- Reuse of resources in digital or software ecosystems

Typical software development lifecycle models*

Waterfall model

V-model

* https://melsatar.blog/2012/03/15/software-development-life-cycle-models-and-methodologies/

Typical software development lifecycle models (cont')*

Evolutionary Prototyping Model

Spiral Model

* https://melsatar.blog/2012/03/15/software-development-life-cycle-models-and-methodologies/

Typical software development lifecycle models (cont”)*

Agile Model

DevOps

* https://melsatar.blog/2012/03/15/software-development-life-cycle-models-and-methodologies/

Tasks within development and maintenance of software systems

Software Engineering Body of Knowledge (SWEBOK)* covered

- Software requirements
- Software design
- Software construction
- Software testing
- Software maintenance
- Software configuration management
- Software engineering management
- Software engineering process

*https://www.computer.org/web/swebok

Software Engineering Body of Knowledge (SWEBOK) covered (Cont')

- Software engineering models and methods
- Software quality
- Software engineering professional practice
- Software engineering economics
- Computing foundations
- Mathematical foundations
- Engineering foundations

Related courses

- TDT4140 Software Engineering
- TDT4250 Advanced Software Design
- TDT4290 Customer-driven project

Outline

- Knowledge to learn to be a good software engineer
- ➢ What advanced SE knowledge to be covered in this course?
- ➢ How is this course organized?

High-level course modules

- High-quality requirements
- High-quality and effective testing
- High-quality and maintainable code
- Advanced topics
  - DevOps
  - Cost and defect estimation
  - Software startup
  - Software eco-systems

Tentative lecture plan

High-quality requirements

Week	Date	Lecture content	Speaker
3	17.01.22	Requirement engineering 1	Anh Nguyen Duc
4	24.01.22	Requirement engineering 2	Anh Nguyen Duc

Challenges in Requirements Engineering

- Importance of getting requirements right:
  1/3 budget to correct errors originates from requirements

Phase in which fixed	Relative Cost
Requirements	1
Design	3 – 6
Coding	10
Development Testing	15 – 40
Acceptance Testing	30 – 70
Operations	40 – 1000

Challenges in Requirements Engineering (Cont')

- Why projects are canceled:

Factors	Percentage of Responses
Incomplete Requirements	13.1%
Lack of User Involvement	12.4%
Lack of Resources	10.6%
Unrealistic Expectations	9.9%
Lack of Executive Support	9.3%
Changing Requirements	8.7%
Lack of Planning	8.1%
Didn't need it any longer	7.5%
Lack of IT Management	4.3%
Technology Illiteracy	9.9%

Requirements Engineering process

- Use cases
- User stories
- Goal-oriented RE approaches

Characteristics of good software requirements specification*

- Complete
- Unambiguous
- Consistent
- Correct
- Verifiable
- Traceable
- Ranked for importance and/or stability
- Modifiable

* IEEE Std 830-1998

Requirements matched by testing – the V-model

IEC 61508 – Test requirements

	Technique/Measure *	Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Probabilistic testing	C.5.1	---	R	R	**R**
2	Dynamic analysis and testing	B.6.5 Table B.2	R	HR	HR	**HR**
3	Data recording and analysis	C.5.2	HR	HR	HR	**HR**
4	Functional and black box testing	B.5.1 B.5.2 Table B.3	HR	HR	HR	**HR**
5	Performance testing	Table B.6	R	R	HR	**HR**
6	Model based testing	C.5.27	R	R	HR	**HR**
7	Interface testing	C.5.3	R	R	HR	**HR**
8	Test management and automation tools	C.4.7	R	HR	HR	**HR**
9	Forward traceability between the software design specification and the module and integration test specifications	C.2.11	R	R	HR	**HR**
10	**Formal verification**	**C.5.12**	**---**	**---**	**R**	**R**

Tentative lecture plan

High-quality and effective testing

Week	Date	Lecture content	Speaker
6	07.02.22	Control flow and data flow testing	Jingyue Li
7	14.02.22	Functional testing with Black Box	Jingyue Li
8	21.02.22	System and acceptance testing and Regression testing	Jingyue Li
9	28.02.22	Testing in Practice	Daniela Cruzes

Tentative lecture plan

High-quality and maintainable code

Week	Date	Lecture content	Speaker
10	07.03.22	Code review and code refactoring	Jingyue Li
11	14.03.22	Technical Debt process in Industry	Guest lecture from Visma

Code review, analysis, and refactoring

Refactoring Code

- Making code better, may not be faster
- Better structured, better built
- More readable, more understandable
- Easy to work with
- Easy to add new features
- Easy to spot and fix bugs
- Keeping code in control
- Improving existing code

Tentative lecture plan

Advanced topics

Week	Date	Lecture content	Speaker
5	31.01.22	DevOps and Git Lab	Nora Thomas from Vipps.no
12	21.03.22	Cost and defect estimation	Anh Nguyen Duc
13	28.03.22	Software startup Software eco-systems	Anh Nguyen Duc

DevOps

- DevOps is the practice of operations and development engineers participating together in the entire service lifecycle, from design through the development process to production support.
- Integrated development and operations
- Supporting quick releases

Cost and defect estimation - Cost

- Why?
  - 55% total number of project runs out of budgets
  - Almost every software project experiences some sort of project delays
  - Estimation is needed for better planning and preventive actions
- Purpose of software estimation
- How
  - Expert judgement: Delphi method, focus group, planning poker, etc
  - Regression-based method: COCOMO models, Putnam model, and function points
  - Simulation-based
  - Machine learning based approach: Bayesian Network, Deep Learning, etc
- Where to apply:
  - Waterfall vs. Agile
  - Within-company vs. cross-company
  - Private vs open source dataset

Cost and defect estimation - Defect

- Defects vs. fault, error, breach, failure
- Software defect can be very costly!
- Defect prediction is a cost-effective to deal with defect
  - Informed static analysis
  - Bug triaging
  - Faulty vs. non-faulty classification
- Approaches
- Metrics:
  - Project-level metrics: project type, budget, duration
  - Team level metrics: engineer's competence
  - Code level metrics: LOC, CK metric suite, code smell, number of functions, number of interfaces, number of class, code complexity, etc
  - Product level metrics: number of component, cohesion, etc
  - Collaboration level metrics: people-to-people interaction, people-to-file interaction
- Advanced defect prediction methods
  - Regression-based approaches
  - Support Vector Machine
  - Machine Learning (supervised/ unsupervised)
  - Decision Trees
  - Naïve Bayes
  - Deep Learning

Software startup

- Airbnb, Uber, Spotify, Skype, Kahoot, AppSumo, Vipps, Mitandbud, etc are used to be startups
- Startup companies are unique:
  - Little or no operating history
  - Limited resources
  - Multiple influences
  - Dynamic technologies and markets
- A temporary organizational state with special way of working, engineering practices and activities can be done different than in established contexts

Software startup

- Understanding failure in startups
- Reasoning about software startup engineering
  - Problem-solution fit and Product-market fit
  - Startup way of working
  - Technical debt
  - Influences of multiple stakeholders
  - Competence and financial situations
- Components of software startups
  - Organization
  - Competence
  - Product
  - Way of working
  - Startup ecosystem

Software startup

- Processes, framework, practices for software startup engineering
  - Lean approach to developing product and businesses
  - Minimum Viable Product
  - Requirement Engineering
  - Minimum Viable User Experience
  - Design thinking
  - Customer journey
  - Growth hacking
- Continuous experimentation in startups
  - Goal-driven hypotheses
  - Build-measure-learn loops
  - Hypothesis driven engineering

Software ecosystem

- "*a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them*" Messerschmitt and Szyperski (2003)
- Key elements:
  - A central hub: an owner of the platform
  - A platform: Usually technological platform
  - Niche players: who generate value from the platform
- Example: app stores, open source projects, etc

Software ecosystem

- Scope of ecosystem

A. A hub between suppliers and customers

B. The hierachy of hubs

C. Dynamic relationships among hubs, suppliers and customers

Software ecosystem

- Scope of ecosystem
- Healthiness of ecosystem
- Collaboration vs. Competition
- Sustainability

Curriculum

- Journal and conference articles (will be uploaded to blackboard)
- Some chapters from books (accessible in NTNU library)

(1st/2nd edition)

Teaching goals

- Apply goal-oriented requirement engineering method and requirement boilerplate to identify and model requirements;
- Identify and correct typical requirements quality issues;
- Apply different testing and code review, analysis, and refactoring strategies;
- Explain the industrial state of the practice methods of verifying and validating high-assurance software-intensive system;

Teaching goals (Cont')

- Understand the state of the practice agile methods, such as DevOps, and apply corresponding tools;
- Explain the issues and solutions of distributed and collaborative software development and maintenance;
- Explain key issues and solutions to managing and operating large and complex software-intensive systems.

Evaluation and grading

- Exercises: 50%
  - Exercise 1: 10%
  - Exercise 2: 25%
  - Exercise 3: 15%

A: 89–100 points
B: 77–88 points
C: 65–76 points
D: 53–64 points
E: 41–52 points
F: 0–40 points

- Exam (3rd June): 50%
- You have to pass both!
  - You have to hand in **all** the exercises **and** get a passing grade to be allowed to take the exam
  - If you fail the exam, you will fail the course

Exercises	Exam	Final grade
Fail	Fail	Fail
Fail	Pass	Fail
Pass	Fail	Fail
Pass	Pass	Combine exercise and exam grades

Exercise 1

- The purpose is to let students practice
  - Software requirement elicitation methods
  - Evaluating software requirements quality

- You need to
  - Based on an existing medium-sized web application, collect and understand new requirements you get from a peer group
  - Document the new requirements in high quality

Exercise 2

- The purpose is to let students practice
  - DevOps tools
  - Extending existing code
  - Software testing methods
- We will ask you extending the existing system and test the existing code
- You need to
  - Apply DevOps tools
  - Develop new code
  - Run tests manually and write test scripts

Exercise 3

- The purpose is to let students practice
  - Code review
  - Code analysis
  - Code refactoring

- You need to
  - Do manual code review of the existing code
  - Refactor the code based on your review and analysis results

Exercise tasks and schedule

The exercise introduction lectures are digital via Zoom on Tuesday 08:15 AM - 10:00 AM.
The exercise introduction lecture schedule is in the following Table.

ID	Weeks	Exercises schedule		
		Introduction lecture	Start	Deliverable
1	3-7	18 Jan. 08:15 - 10:00	18 Jan.	• Final report of exercise 1. 14 February at 23:59 (Monday), Week 7
2	7-12	15 Feb. 08:15 - 10:00	15 Feb.	• Final report of exercise 2. 21 March at 23:59 (Monday), Week 12
3	12-14	22 March 08:15 - 10:00	22 March	• Final report of exercises 3. 4 April at 23:59 (Monday), Week 14

In other weeks without exercise introduction lectures, we will have TAs online via the same Zoom link to answer questions.

Exercise groups

- 1-2 students in each group
- To form a group
  - Send an e-mail with a list of the names and e-mails of the group members to tdt4242@idi.ntnu.no
- If you don't have a group
  - Send an e-mail to tdt4242@idi.ntnu.no
  - You will be assigned to a group
- Deadline: 1 Feb.

Deadline for exercises

All exercises will have a deadline for delivery.

This deadline may only be exceeded after agreement with the

- course responsible
- teaching assistant (email: tdt4242@idi.ntnu.no)

If no such agreement exists, we will deduct 20% points on the grade for any obligatory exercise for each week it is delayed.

Reference group

- A group of 2-3 students that have a special duty to provide feedback about the course during the semester
- We'll have 2-3 meetings where we can discuss content and form of lectures and assignments
- The group should be formed during the first 3 weeks, so please nominate yourself (or others)
- More information (In Norwegian) → http://www.ntnu.no/utdanningskvalitet/roller.html#Studentene

About you

I need to know a little more about you to adapt my teaching focuses and exercises!

Empirical research on Software Ecosystem (SeCo) and Software Startup

Anh Nguyen Duc

Universitetet i Sørøst-Norge

NTNU
Norwegian University of Science and Technology

Agenda

- Software Ecosystem (SECO)
- Software Startups

Software Ecosystem

Introduction

- Software ecosystems is an emerging trend within the software industry
- The shift from closed organizations and processes towards open structures
- Ecosystem metaphor is used to understand relationships among stakeholders

Biological ecosystem

Definitions of Software Ecosystem (SeCo)

- SeCo as an informal network of independent units that have a positive influence on the economic success of a software product and benefit from it (Kittlaus and Clough)
- SeCo as consisting of the set of software solutions that enable, support, and automate the activities and transactions by the actors in the associated social or business ecosystems and the organizations that provide these solutions (Bosch)
- a set of actors functioning as a unit and interacting with a shared market for software and services, together with the relationships among them (Jansen)
- Three shared concepts stand out in these definitions:
  - (1) actors, organizations and businesses,
  - (2) networks and social or business ecosystems, and
  - (3) software.

Menti.com

Examples of Software Ecosystems you know?

Foundation – two-sided market theory

Parker, Geoffrey; Van Alstyne, Marshall (2005). "Two-Sided Network Effects: A Theory of Information Product Design". *Management Science*. **51** (10): 1494–1504

Software Ecosystem (SECO)

- App stores: Apple, Google, Microsoft, etc

Software Ecosystem (SECO)

- Service providers on top of Shopify ecosystem

Other examples of Software Ecosystems

- **Standards** - XML, BPM, OSGi, J2EE, Corba, SEPA, etc.
- **Products** - OpenOffice, Microsoft Word, SAP BusinessOne, Grand Theft Auto, etc.
- **Hardware** - Playstation 3, HTC Diamond, PDAs, BMW 5 series, etc.
- **Platforms** - .Net, Facebook, Android, OS X, etc.

Types of SeCo

Example: The Apple iOS software ecosystem is based on a software platform and coordinated by a privately owned entity with a commercial extension market to which participants can submit extensions after making a payment .

- Core technology
  - software platform,
  - software service platform,
  - software standard
- Coordinated by
  - Community
  - Private organization
- Extension market
  - no extension market,
  - a list of extensions,
  - an extension market,
  - a commercial extension market,
  - multiple extension markets
- Accessibility
  - for free,
  - after a screening,
  - after making a payment

Characteristics of SeCos

- **Complexity -** In a platform ecosystem with numerous actors, complexity must be controlled somehow to reduce risk of gridlocks, unpredictable ripple effects and co-innovation problems
- **Productivity** - A network's ability to consistently transform technology and other raw materials of innovation into lower costs and new products. Simple to measure: return on invested capital.
- **Robustness** - Should be capable of surviving disruptions such as unforeseen technological change.
- **Niche creation** - the ecosystem's capacity to increase meaningful diversity through the creation of valuable new functions or niches.

Strategy as Ecology, Marco Iansiti and Roy Levien. Harvard Business Review, March 2004.

A healthy Software Ecosystem

- A stable and sustainable platform and mechanism to govern the platform
- Developers, software firms making benefits from collective innovation
- Consumers receive high quality services/ products, opportunities for value co-creation

Manikas, K., & Hansen, K. M. (2013). Software ecosystems – A systematic literature review. Journal of Systems and Software, 86(5), 1294–1306. https://doi.org/10.1016/j.jss.2012.12.026

A healthy Software Ecosystem

- Sustainability is the key!

- How to design and maintain a sustainable software ecosystem ?

A healthy Software Ecosystem

- Sustainability is the key!

- A viable business model
- A well-organized inter-organizational interaction model
- A paradigm supporting Collaboration
- A mechanism to allow Coopetition

Raza-Ullah, T., Bengtsson, M., & Kock, S. (2014). The coopetition paradox and tension in coopetition at multiple levels. Industrial Marketing Management, 43(2), 189–198. https://doi.org/10.1016/j.indmarman.2013.11.001

The 3C in SeCo

Coopetition in a nutshell

Netflix uses AWS cloud services while compte with Amazon Prime

https://fourweekmba.com/coopetition/

A model of coopetition

Eriksson, P. E. (2008). Procurement Effects on Coopetition in Client-Contractor Relationships. Journal of Construction Engineering and Management, 134(2), 103–111. https://doi.org/10.1061/(ASCE)0733-9364(2008)134:2(103)

Open coopetition

Coopetition +
Open Source

https://en.wikipedia.org/wiki/Open_coopetition

How do commercial companies maintain both collaboration and competition in an open source software ecosystem?

Our cases

Anh Nguyen Duc; Daniela Cruzes S.; Geir Hanssen K.; Pekka Abrahamsson; Terje Snarby (2017). "Coopetition of software firms in open source software ecosystems". ICSOB 2017

Wireshark ecosystem - social analysis

Wireshark ecosystem - conceptual layers

How do commercial firms manage coopetition with other firms in such context?

How do commercial firms manage coopetition with other firms in such context?

- Open-core sourcing policy
- Upstream first - get features integrated into open source projects before integrating them into firm’s product

A model of open coopetition in SECO

Anh Nguyen-Duc, Daniela S Cruzes, Snarby Terje, Pekka Abrahamsson (2019) Do software firms collaborate or compete? A model of coopetition in community-initiated OSS projects. E-informatyka Software Engineering

Software Startups

Software startups

Startup companies are unique:

- Little or no operating history
- Limited resources
- Multiple influences
- Dynamic technologies and markets
- Scalable business model

And more …

Software startups

Startup companies are unique:

- Software in value proposition
- Innovation focus
- Lack of resources
- High uncertainty
- Time-pressure
- Highly reactive
- Rapidly evolving
- Temporary organizational state
- Little/ no history of stable operations
- Product development and business development in parallel
- Seek for product-market fit
- Strong presence of entrepreneurial personalities

- **60%** of startups do not survive in the first five years
- **75%** of venture capital funded startups fail
- Little rigor and relevance exist in the studies about software startups
- Do not focus on investigating issues and challenges

Menti.com

Name a software startup that you know in Norway

Software startups

1. J. Melegati, A. Goldman, F. Kon, and X. Wang, “A model of requirements engineering in software startups,” *Information and Software Technology*, vol. 109, pp. 92–107, May 2019
2. V Berg, J Birkeland, A Nguyen-Duc, IO Pappas, L Jaccheri, Software startup engineering: A systematic mapping study, Journal of Systems and Software 144, 255-274, 2018

Software startup research

- Empirical
- Reveal ar
  reflect on
  phenomer

- **Technical debt:**
  - when development teams take actions to expedite the delivery of a piece of functionality or a project which later needs to be refactored
  - Intentional vs. unintentional
  - prioritizing speedy delivery over perfect code
- Technical debt as integral parts of software startups: the need to shorten time-to-market, by speeding up the development through low-precision engineering activities, is counterbalanced by the need to restructure the product before targeting further growth

1. C. Giardino, N. Paternoster, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, “Software Development in Startup Companies: The Greenfield Startup Model,” IEEE Transactions on Software Engineering, vol. 42, no. 6, pp. 585–604, Jun. 2016

- Technical debt:

1. C. Giardino, N. Paternoster, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, “Software Development in Startup Companies: The Greenfield Startup Model,” IEEE Transactions on Software Engineering, vol. 42, no. 6, pp. 585–604, Jun. 2016

Technical debt – four facet model in software startups

1. Cico et al. Technical Debt Approach and Consciousness in Startups Transitioning to Growth Phase - A Four Facet Model. Submitted to IEEE Access, 2022

Technical debt – fo

1. Cico et al. Technical Debt Approach and Consciousness in Sta
Access, 2022

- Domain knowledge, technical expertise, and teamwork are the key components in the early stages of a start-up. Shortages of any of these components can be compensated with specific practices
- Key difference and difficulty to practice software engineering in

Start-up life-cycle phases

INCEPTION Build and launch the first version of the product	STABILIZATION Develop the product further and prepare for scaling	GROWTH Aim to achieve desired market share and growth rate	MATURITY Transition into an established organisation

Engineering process areas

I TEAM

- G1 P1 Establish a team — 12 cases
- C1 Manage a large team — 14 cases
- C2 Lack of team expertise, engagement, coordination, and leadership — 14 cases
- C3 P2 Need for specialist skills — 33 cases
- P3 Outsource engineering work — 6 cases

MENTS
RING

- G2 P4 Validate own, invented ideas — 35 cases
- C4 P5 Establish a feedback loop Customer input drives product work — 32 cases
- Balance customer value with time to market — 6 cases
- Support business needs — 19 cases

1. J.
Sta
2. E.
En
3. C.
Gr

- **Hypothesis engineering** in software startups
  - There are always assumptions
  - Decisions are often driven by the instinct of startup founders
  - Lean startup – is this really used?

1. J. Melegati, H. Edison, and X. Wang, “XPro: a Model to Explain the Limited Adoption and Implementation of Experimentation in Software Startups,” IEEE Transactions on Software Engineering, pp. 1–1, 2020
2. E. Ries, *The Lean Startup: How Today’s Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, 1 edition. New York: Currency, 2011.

- Cognitive map to visualize the hypotheses and their relationships

1. J. Melegati, H. Edison, and X. Wang, “XPro: a Model to Startups,” IEEE Transactions on Software Engineering, , ,
2. E. Ries, *The Lean Startup: How Today’s Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, 1 edition. New York: Currency, 2011.

Nguyen Duc, A., Shah S. and Abrahamsson, P. (2016). Towards an early stage software startups evolution model. Euromicro SEAA

- People:
  - Startup Founders
  - Stakeholders in Startup Ecosystems
- Cynefine-based categories of startups' decisions:
- Simple decisions:
  - Setting up social media accounts for the startup
  - Focusing on acquiring funding
  - Creating a common team calendar

K.-K. Kemell, E. Ventilä, P. Kettunen, and T. Mikkonen, "Amidst Uncertainty -- or Not? Decision-Making in Early-Stage Software Startups," *arXiv:2102.06501 [cs]*, vol. 370, pp. 369–377, 2019, doi: .

Nguyen Duc, A., Shah S. and Abrahamsson, P. (2016). Towards an early stage software startups evolution model. Euromicro SEAA

- Cynefine-based categories of startups' decisions:
- Complicated decisions:
  - Practicing public speaking
  - Deciding on different pricing models
- Complex decisions:
  - Validating the most important features
  - Gathering a team

K.-K. Kemell, E. Ventilä, P. Kettunen, and T. Mikkonen, "Amidst Uncertainty -- or Not? Decision-Making in Early-Stage Software Startups," *arXiv:2102.06501 [cs]*, vol. 370, pp. 369–377, 2019, doi: .

- Two logics of making decisions: Effectuation and Causations
- Effectuation:

Sarasvathy, S.D.: Causation and eectuation: Toward a theoretical shift from economic inevitability to entrepreneurial contingency. Academy of management Review 26(2), 243-263 (2001

- Two logics of making decisions in startup projects:
  - Will they apply to decisions at technical and product-level as well?

A. Nguyen-Duc, K.-K. Kemell, and P. Abrahamsson, “The entrepreneurial logic of startup software development: A study of 40 software startups,” *Empir Software Eng*, vol. 26, no. 5, p. 91, Jul. 2021

- **What** - Minimum Viable Product
  - … as a prototype
  - … as a communication tool
  - …. as a learning mechanism
  - …. as documentation approach

V. Berg, J. Birkeland, A. Nguyen-Duc, I. O. Pappas, and L. Jaccheri, “Achieving agility and quality in product development – an empirical study of hardware startups,” *Journal of Systems and Software*, vol. 167, p. 110599, Sep. 2020

Nguyen-Duc, A., Wang, X., & Abrahamsson, P. (2017). What Influences the Speed of Prototyping? An Empirical Investigation of Twenty Software Startups. *XP Conference*

- **How -**

  Produ

  - …
  - …

Nguyen-Duc, A., V
Startups. *XP Confe*

Thank you! Questions?

Anh Nguyen-Duc

angu@usn.no

Goal-Oriented Requirements Engineering: A Guided Tour

Axel van Lamsweerde

Département d'Ingénierie Informatique
Université catholique de Louvain
B-1348 Louvain-la-Neuve (Belgium)
avl@info.ucl.ac.be

Abstract

Goals capture, at different levels of abstraction, the various objectives the system under consideration should achieve. Goal-oriented requirements engineering is concerned with the use of goals for eliciting, elaborating, structuring, specifying, analyzing, negotiating, documenting, and modifying requirements. This area has received increasing attention over the past few years.

The paper reviews various research efforts undertaken along this line of research. The arguments in favor of goal orientation are first briefly discussed. The paper then compares the main approaches to goal modeling, goal specification and goal-based reasoning in the many activities of the requirements engineering process. To make the discussion more concrete, a real case study is used to suggest what a goal-oriented requirements engineering method may look like. Experience with such approaches and tool support are briefly discussed as well.

1. Introduction

Goals have long been recognized to be essential components involved in the requirements engineering (RE) process. As Ross and Schoman stated in their seminal paper, *"requirements definition must say why a system is needed, based on current or foreseen conditions, which may be internal operations or an external market. It must say what system features will serve and satisfy this context. And it must say how the system is to be constructed"* [Ros77]. Many informal system development methodologies from the good old times included some form of goal-based analysis, called context analysiis [Ros77], definition study [Hic74], participative analysis [Mun81], and so forth. Typically, the current system under consideration is analyzed in its organizational, operational and technical setting; problems are pointed out and opportunities are identified; high-level goals are then identified and refined to address such problems and meet the opportunities; requirements are then elaborated to meet those goals. Such natural practice has led requirements documentation standards to require a specific document section devoted to the objectives the system should meet (see, e.g., the IEEE-Std-830/1993 standards).

Surprisingly enough, goals have been largely ignored both from the literature on software modeling and specification and from the literature on object-oriented analysis (one notable exception is [Rub92]). UML advocates sometimes confess the need for higher-level abstractions: *"In my work, I focus on user goals first, and then I come up with use cases to satisfy them; by the end of the elaboration period, I expect to have at least one set of system interaction use cases for each user goal I have identified"* [Fow97, p.45]). The prominent tendency in software modeling research has been to abstract programming constructs up to requirements level rather than propagate requirements abstractions down to programming level [Myl99].

Requirements engineering research has increasingly recognized the leading role played by goals in the RE process [Yue87, Rob89, Ber91, Dar91, Myl92, Jar93, Zav97b]. Such recognition has led to a whole stream of research on goal modeling, goal specification, and goal-based reasoning for multiple purposes, such as requirements elaboration, verification or conflict management, and under multiple forms, from informal to qualitative to formal.

The objective of this paper is to provide a brief but hopefully comprehensive review of the major efforts undertaken along this line of research. Section 2 first provides some background material on what goals are, what they are useful for, where they are coming from, and when they should be made explicit in the RE process. Section 3 discusses the major efforts in modeling goals in terms of features and links to other artefacts found in requirements models. Section 4 reviews the major techniques used for specifying goals. Section 5 on goal-based reasoning reviews how goals are used in basic activities of the RE process such as requirements elicitation, elaboration, verification, validation, explanation, and negotiation, and in particular for difficult aspects of that process such as conflict management, requirements deidealization, and alternative selection. Section 6 then suggests what a goal-oriented RE method may look like by enacting it on a real case study of a safety-critical train control system. This naturally leads to a brief review, in Section 7, of industrial projects in which the use of such methods was felt conclusive; the supporting tools used in those projects are also briefly discussed there. Section 8 just opens some fairly recent pieces of goal-based work beyong requirements engineering.

2. The background picture

Reviewing the current state of the art in goal-oriented RE would not make much sense without first addressing the what, why, where and when questions about this area of research.

Invited mini-tutorial paper, appeared in
Proceedings RE'01, 5th IEEE International Symposium on Requirements Engineering, Toronto, August 2001, 249-263.

What are goals?

A *goal* is an objective the system under consideration should achieve. Goal formulations thus refer to intended properties to be ensured; they are optative statements as opposed to indicative ones, and bounded by the subject matter [Jac95, Zav97a].

Goals may be formulated at different levels of abstraction, ranging from high-level, strategic concerns (such as *"serve more passengers"* for a train transportation system or *"provide ubiquitous cash service"* for an ATM network system) to low-level, technical concerns (such as *"acceleration command delivered on time"* for a train transportation system or *"card kept after 3 wrong password entries"* for an ATM system).

Goals also cover different types of concerns: functional concerns associated with the services to be provided, and non-functional concerns associated with quality of service --such as safety, security, accuracy, performance, and so forth.

The *system* which a goal refers to may be the current one or the system-to-be; both of them are involved in the RE process. High-level goals often refer to both systems. The system-to-be is in essence composite; it comprises both the software and its environment, and is. made of active components such as humans, devices and software. As opposed to passive ones, active components have choice of behavior [Fea87, Yue87, Fic92]; henceforth we will call them *agents*. Unlike requirements, a goal may in general require the cooperation of a hybrid combination of multiple agents to achieve it [Dar93]. In a train transportation system, for example, the high-level goal of safe transportation will typically require the cooperation of on board train controllers, the train tracking system, station computers, the communication infrastructure, passengers, and so forth. In an ATM system, the goal of providing cash to eligible users will require the cooperation of the ATM software, sensors/actuators, the customer, etc. One of the important outcomes of the RE process is the decision on what parts of the system will be automated and what parts will not. A goal under responsibility of a single agent in the software-to-be becomes a *requirement* whereas a goal under responsibility of a single agent in the environment of the software-to-be becomes an *assumption* [Lam98b, Lam98c]. Unlike requirements, assumptions cannot be enforced by the software-to-be; they will hopefully be satisfied thanks to organizational norms and regulations, physical laws, etc.

Why are goals needed?

There are many reasons why goals are so important in the RE process.

- Achieving requirements completeness is a major RE concern.. Goals provide a precise criterion for *sufficient completeness* of a requirements specification; the specification is complete with respect to a set of goals if all the goals can be proved to be achieved from the specification and the properties known about the domain considered [Yue87].
- Avoiding irrelevant requirements is another major RE concern. Goals provide a precise criterion for requirements *pertinence*; a requirement is pertinent with respect to a set of goals in the domain considered if its specification is used in the proof of one goal at least [Yue87].
- Explaining requirements to stakeholders is another important issue. Goals provide the rationale for requirements, in a way similar to design goals in design processes [Mos85, Lee91]. A requirement appears because of some underlying goal which provides a base for it [Ros77, Dar91, Som97]. More explicitly, a goal refinement tree provides traceability links from high-level strategic objectives to low-level technical requirements. In particular, for business application systems, goals may be used to relate the software-to-be to organizational and business contexts [Yu93].
- Goal refinement provides a natural mechanism for structuring complex requirements documents for increased readability. (This at least has been our experience in all industrial prjects we have been involved in, see Section 7.)
- Requirements engineers are faced with many alternatives to be considered during the requirements elaboration process. Our extensive experience revealed that alternative goal refinements provide the right level of abstraction at which decision makers can be involved for validating choices being made or suggesting other alternatives overlooked so far. Alternative goal refinements allow alternative system proposals to be explored [Lam00c].
- Managing conflicts among multiple viewpoints is another major RE concern [Nus94]. Goals have been recognized to provide the roots for detecting conflicts among requirements and for resolving them eventually [Rob89, Lam98b].
- Separating stable from more volatile information is another important concern for managing requirements evolution. A requirement represents one particular way of achieving some specific goal; the requirement is therefore more likely to evolve, towards another way of achieving that same goal, than the goal itself. The higher level a goal is, the more stable it will be. Others have made that same observation [Ant94]. It turns out that different system versions often share a common set of high-level goals; the current system and the system-to-be correspond to alternative refinements of common goals in the goal refinement graph, and can therefore be integrated into one single goal model (see Section 3).
- Last but not least, goals drive the identification of requirements to support them; they have been shown to be among the basic driving forces, together with scenarios, for a systematic requirements elaboration process [Dar91, Rub92, Dar93, Ant98, Dub98, Kai00, Lam00c]. We will come back to this in Sections 5 and 6.

Where are goals coming from?

Goal identification is not necessarily an easy task [Lam95, Ant98, Hau98, Rol98]. Sometimes they are explicitly stated by stakehokders or in preliminary material available to requirements engineers. Most often they are implicit so that goal elicitation has to be undertaken.

The preliminary analysis of the current system is an important source for goal identification. Such analysis usually results in a list of problems and deficiencies that can be for-

mulated precisely. Negating those formulations yields a first list of goals to be achieved by the system-to-be.

In our experience, goals can also be identified systematically by searching for intentional keywords in the preliminary documents provided, interview transcripts, etc. [Lam00c].

Once a preliminary set of goals and requirements is obtained and validated with stakeholders, many other goals can be identified by *refinement* and by *abstraction*, just by asking HOW and WHY questions about the goals/requirements already available, respectively [Lam95, Lam00c].

More sophisticated techniques for goal refinement and abstraction (notably, from scenarios) will be reviewed in Section 5. Other goals are identified by resolving conflicts among goals or obstacles to goal achievement, see Section 5 too.

A common misunderstanding about goal-oriented approaches is that they are inherently top-down; this is by no means the case as it should hopefully be clear now from the discussion above.

When should goals be made explicit?

It is generally argued that goal models are built during the early phases of the RE process [Dar93, Yu97, Dub98]. The basis for the argument is the driving role played by goals in that process; the soonest a goal is identified and validated, the best. This does not imply any sort of waterfall-like requirements elaboration process, however. As requirements "implement" goals much the same way as programs implement design specifications, there is some inevitable intertwining of goal identification and requirements elaboration [Lam95, Swa82]. Goals may thus sometimes be identified fairly lately in the RE process --especially when WHY questions about technical details or scenarios, initially taken for granted, are raised lately in the process.

3. Modeling goals

The benefit of goal modeling is to support heuristic, qualitative or formal reasoning schemes during requirements engineering (see Section 5). Goals are generally modelled by *intrinsic features* such as their type and attributes, and by their *links* to other goals and to other elements of a requirements model.

Goal types and taxonomies. Goals can be of different types. Several classification axes have been proposed in the literature.

Functional goals underlie services that the system is expected to deliver whereas *non-functional* goals refer to expected system qualities such as security, safety, performance, usability, flexibility, customizability, interoperability, and so forth [Kel90]. This typology is overly general and can be specialized. For example, *satisfaction* goals are functional goals concerned with satisfying agent requests; *information* goals are functional goals concerned with keeping such agents informed about object states [Dar93]. Non-functional goals can be specialized in a similar way. For example, *accuracy* goals are non-functional goals requiring the state of software objects to accurately reflect the state of the corresponding monitored/controlled objects in the environment [Myl92, Dar93] --such goals are often overlooked in the RE process; their violation may be responsible for major failures [Lam00a]. *Performance* goals are specialized into time and space performance goals, the former being specialized into *response time* and *throughput* goals [Nix93]. *Security* goals are specialized into *confidentiality*, *integrity* and *availability* goals [Amo94]; the latter can be specialized in turn until reaching domain-specific security goals. A rich taxonomy for non-functional goals can be found in [Chu00].

Another distinction often made in the literature is between *soft goals*, whose satisfaction cannot be established in a clear-cut sense [Myl92], and (hard) goals whose satisfaction can be established through verification techniques [Dar93, Dar96]. Soft goals are especially useful for comparing alternative goal refinements and chosing one that contributes the “best” to them, see below.

Another classification axis is based on types of temporal behaviour prescribed by the goal. [Dar93]. *Achieve* (resp. *cease*) goals generate system behaviours, in that they require some target property to be eventually satisfied in some future state (resp. denied); *maintain* (resp. *avoid*) goals retrict behaviours, in that they require some target property to be permanently satisfied in every future state (resp. denied) unless some other property holds. *Optimize* goals compare behaviours to favor those which better ensure some soft target property.

In a similar vein, [Sut93] proposes a classification according to desired system states (e.g., positive, negative, alternative, feedback, or exception-repair) and to goal level (e.g., policy level, functional level, domain level). [Ant94] makes a distinction beween objective goals, that refer to objects in the system, and adverbial goals, that refer to ways of achieving objective goals.

Goal types and taxonomies are used to define heuristics for goal acquisition, goal refinement, requirements derivation, and semi-formal consistency/completeness checking [Dar93, Sut93, Ant98, Chu00, Ant01], or to retrieve goal specifications in the context of specification reuse [Mas97].

Goal attributes. Beside their type, goals can also be intrinsically characterized by attributes such as their *name* and their *specification* (see Section 4). *Priority* is another important attribute that can be attached to goals [Dar93]. Qualitative values for this attribute allow mandatory or optional goals to be modelled with various degrees of optionality. Priorities are often used for resolving conflicts among goals [Rob89, Lam98b]. Other goal attributes that have been proposed include goal *utility* and *feasibility* [Rob89].

Goal Links. Many different types of links have been introduced in the literature to relate goals (a) with each other and (b) with other elements of requirements models. Such links form the basis for defining *goal structures*. We discuss inter-goal links first, and then links between goals and other elements of requirements models such as agents, scenarios, or operations.

Links between goals are aimed at capturing situations where goals positively or negatively *support* other goals. Directly borrowed from problem reduction methods in Artificial

Intelligence [Nil71], AND/OR graphs may be used to capture goal *refinement* links [Dar91, Dar93]. *AND-refinement* links relate a goal to a set of subgoals (called refinement); this means that satisfying all subgoals in the refinement is sufficient for satisfying the parent goal. *OR-refinement* links relate a goal to an alternative set of refinements; this means that satisfying one of the refinements is sufficient for satisfying the parent goal. In this framework, a *conflict* link between two goals is introduced when the satisfaction of one of them may prevent the other from being satisfied. Those link types are used to capture alternative goal refinements and potential conflicts, and to prove the correctness of goal refinements (see Section 5).

Weaker versions of those link types have been introduced to relate soft goals [Rob89, Myl92, Chu00] as the latter can rarely be said to be satisfied in a clear-cut sense. Instead of goal satisfaction, goal *satisficing* is introduced to express that subgoals are expected to achieve the parent goal within acceptable limits, rather than absolutely. A subgoal is then said to *contribute* partially to the parent goal, regardless of other subgoals; it may contribute *positively* or *negatively*. The semantic rules are now as follows. If a goal is AND-decomposed into subgoals and all subgoals are satisficed, then the parent goal is satisficeable; but if a subgoal is denied then the parent goal is deniable. If a goal contributes negatively to another goal and the former is satisficed, then the latter is deniable. These rules are used for qualitative reasoning about goal satisficing (see Section 5).

Beside inter-goal links, goals are in general also linked to other elements of requirements models. KAOS introduces AND/OR *operationalization* links to relate goals to the operations which ensure them through corresponding required pre-, post-, and trigger conditions [Lam98c, Lam00c] (the older notion of operationalization [Dar91, Dar93] was revised and simplified from practical experience). Others have used similar links between goals and operations, e.g., [Ant94, Ant98, Kai00]. In [Myl92], the inter-goal *contribution* link types are extended to capture the positive/negative contribution of requirements to goals; *argumentation* links are also introduced to connect supporting arguments to contribution links.

There has been a massive amount of work on linking goals and scenarios together --e.g., [Fic92, Dar93, Pot95, Lei97, Sut98, Ant98, Hau98, Lam98b, Rol98, Kai00, Ant01]. The obvious reason is that scenarios and goals have complementary characteristics; the former are concrete, narrative, procedural, and leave intended properties implicit; the latter are abstract, declarative, and make intended properties explicit. Scenarios and goals thus complement each other nicely for requirements elicitation and validation. By and large the link between a goal and a scenario is a *coverage link*; the main differences between the various modeling proposals lie in the fact that a scenario may be type-level or instance-level, may be an example or a counter-example of desired behavior, and may exercise a goal or an obtsacle to goal achievement.

Goal models may also be related to object models as goal formulations refer to specific objects, e.g., entities, relationships or agents [Dar93]. This link type allows pertinent object models to be systematically derived from goal models [Lam00c].

Various proposals have also been made to relate goals to agents. In KAOS, *responsibility* links are introduced to relate the goal and agent submodels. A goal may be assigned to alternative agents through OR responsibility links; this allows alternative boundaries to be explored between the software-to-be and its environment. “Responsibility” means that the agent is committed to restrict its behavior by performing the operations it is assigned to only under restricted conditions, namely, those prescribed by the required pre-, post-, and trigger conditions [Dar93]. This notion of responsibility derives from [Fea87, Fin87]; it is studied in depth in [Let01]. *Wish* links are also sometimes used in heuristics for agent assignment [Dar91]; e.g., one should avoid assigning a goal to an agent wishing other goals in conflict with that goal..

In the i* framework [Yu93, Yu97], various types of agent dependency links are defined to model situations where an agent depends on another for a goal to be achieved, a task to be achieved, or a resource to become available. For each type of dependency an operator is defined; operators may be combined to define plans that agents may use to achieve goals. The purpose of this modelling is to support various kinds of checks such as the viability of an agent's plan or the fulfilment of a commitment between agents. Although initially conceived for modeling the organizational environment of the software-to-be, the TROPOS project is currently aiming at propagating this framework to later stages of the software lifecycle, notably, for modeling agent-oriented software architectures.

Various authors have also suggested representing the links between goals and organizational policies, e.g., [Sib93, Fea93, Sut93].

At the process level, it may be useful for traceability purpose [Got95] to record which actor owns which goal or some view of it [Lam98b].

4. Specifying goals

Goals must obviously be specified precisely to support requirements elaboration, verification/validation, conflict management, negotiation, explanation and evolution.

An informal (but precise) specification should always be given to make it precise what the goal name designates [Zav97a].

Semi-formal specifications generally *declare* goals in terms of their type, attribute, and links (see Section 3). Such declarations may in general be provided alternatively using a textual or a graphical syntax (see, e.g., [Dar98]). In the NFR framework [Myl92], a goal is specified by the most specific subtype it is an instance of, parameters that denote the object attributes it refers to, and the degree of satisficing/denial by child goals. Semi-formal specifications often include keyword verbs with some predefined semantics. For example, Achieve, Maintain and Avoid verbs in KAOS specify a temporal logic pattern for the goal name appearing as parameter [Dar93]; they implicitly specify that a corresponding target condition should hold some time in the future, always in the

future unless some other condition holds, or never in the future. The intent is to provide a lightweight alternative to full formalization of the goal formulation, still amenable to some form of analysis. This basic set has been extended with qualitative verbs such as Improve, Increase, Reduce, Make, and so forth [Ant98]. In a similar spirit, goals in [Rol98] are represented by verbs with different parameters playing different roles with respect to the verb --e.g., target entities affected by the goal, beneficiary agents of the goal achievement, resource entities needed for goal achievement, source or destination of a communication goal, etc.

Formal specifications *assert* the goal formulation in a fully formal system amenable to analysis. In KAOS, such assertions are written in a real-time linear temporal logic heavily inspired from [Man92, Koy92] with the usual operators over past and future states, bound by time variables; semantically, they capture maximal sets of desired behaviors [Dar93, Let01]. The KAOS language is "2-button" in that the formal assertion layer is optional; it is used typically for critical aspects of the system only.

More formal specifications yield more powerful reasoning schemes at the price of higher specification effort and lower usability by non-experts; the various techniques briefly reviewed here should thus be seen as complementary means rather than alternative ones; their suitability may heavily depend on the specific type of system being considered.

5. Reasoning about goals

The ultimate purpose of goal modelling and specification is to suport some form of goal-based reasoning for RE subprocesses such as requirements elaboration, consistency and completeness checking, alternative selection, evolution management, and so forth.

5.1 Goal verification

One of the benefits of goal-oriented RE is that one can verify that the requirements entail the goals identified, and check that the set of requirements specified is sufficiently complete for the set of goals identified [Yue87]. More precisely, if *R* denotes the set of requirements, *As* the set of environmental assumptions, *D* the set of domain properties, and *G* the set of goals, the following satisfaction relation must hold for each goal *g* in *G*::

$$R, As, D \models g \quad \text{with} \quad R, As, D \not\models \textbf{false}$$

This may be checked informally, or formally if the goal specifications and domain properties are formalized. For temporal logic specifications one may rely on the proof theory of temporal logic and use tools such as, e.g., STeP [Man96].

A lightweight alternative is to use formal refinement patterns fo Achieve, Maintain and Avoid goals [Dar96]. Such patterns are proved correct and complete once for all; refinements in the goal graph are then verified by matching them to one applicable pattern from the library. The mathematical proof intricacies are thereby hidden. A frequently used pattern is the decomposition-by-milestone pattern that refines a parent Achieve goal

$$P \Rightarrow \Diamond Q$$

into two subgoals:

$$P \Rightarrow \Diamond \mathbf{R}\ ,\ \mathbf{R} \Rightarrow \Diamond Q$$

where the "◊" temporal operator means "sometime in the future". Another frequently used pattern is the decomposition-by-case pattern that refines the same parent Achieve goal into three subgoals:

$$P \wedge \mathbf{R} \Rightarrow \Diamond Q\ ,\ P \Rightarrow \Diamond R\ ,\ P \Rightarrow P\ W\ Q$$

where the "*W*" temporal operator means "always in the future unless".

The techniques above can be used for goals that can be said to be established in a clear-cut sense. For soft goals, the qualitative reasoning procedure provided by the NFR framework is particularly appropriate [Myl92]. This procedure determines the degree to which a goal is satisficed/denied by lower-level goals/requirements. A node or link in the goal graph is labelled *S* (satisficed) if it is satisficeable and not deniable; *D* (denied) if it is deniable but not satisficeable; *C* (conflicting) if it is both satisficeable and deniable; and *U* (undetermined) if it is neither satisficeable nor deniable. The general idea is to propagate such labels along satisficed links bottom-up, from lower-level nodes (i.e. requirements) to higher-level nodes (i.e. goals). Additional label values can be assigned at intermediate stages of the procedure, namely, U^+ (inconclusive positive support), U^- inconclusive negative support, and *?* (requiring user intervention to specify an appropriate label value). Rules for bottom-up propagation of labels are then defined accordingly. A example of application of this framework to performance goals can be found in [Nix93].

5.2 Goal validation

Goals can be validated by identifying or generating scenarios that are covered by them [Hau98]. One may even think of enacting such scenarios to produce animations [Hey98]. The scenario identification process is generally based on heuristics [Sut98, Ant98].

In [And89], plan-based techniques are used to tentatively generate scenarios showing that a goal can be achieved without reaching prohibited conditions. Goals, prohibited conditions and operations are specified formally by simple state predicates. An automated planner first produces a trial scenario to achieve the goal condition; it then checks for faults in the proposed scenario by looking for scenarios achieving the prohibited conditions; finally it assists the specifier in modifying the set of operations in case faults are found. [Fic92] explores this deficiency-driven paradigm further. The system is specified by a set of goals, formalized in some restricted temporal logic, a set of scenarios, expressed in a Petri net-like language, and a set of agents producing restricted scenarios to achieve the goals they are assigned to. The general approach consists of (a) trying to detect inconsistencies between scenarios and goals, and (b) applying operators that modify the specification to remove the inconsistencies. Step (a) is carried out by a planner that searches for behaviours leading to some goal violation. The operators offered to the analyst in Step (b) encode heuristics for specification debugging --e.g., introduce an agent whose responsibility is to prevent the state transitions that are the last step

in breaking the goal. There are operators for introducing new types of agents with appropriate responsibilities, splitting existing types, introducing communication and synchronization protocols between agents, weakening idealized goals, etc. The repeated application of deficiency detection and debugging operators allows the analyst to explore the design space and hopefully converge towards a satisfactory specification.

5.3 Goal-based requirements elaboration

The technique just sketched above is a first step towards making verification/validation contribute to the requirements elaboration process. The main reason for goal-oriented RE after all is to let goals help elaborating the requirements supporting them. A goal-based elaboration typically consists of a hybrid of top-down and bottom-up processes, plus additional processes driven by the handling of possible abnormal agent behaviors, the management of conflicting goals, the recognition of analogical situations from which specifications can be transposed, and so forth. Note, however, that for explanatory purpose the resulting requirements document is in general better presented in a top-down way.

Goal/requirement elicitation by refinement

An obvious (but effective) informal technique for finding out subgoals and requirements is to keep asking HOW questions about the goals already identified [Lam95, Lam00c].

Formal goal refinement patterns may also prove effective when goal specifications are formalized; typically, they help finding out subgoals that were overlooked but are needed to establish the parent goal. Consider a simple train control system, for example, and the functional goal of train progress through consecutive blocks:

Goal Achieve [TrainProgress]
FormalDef (∀ t:r Train, b: Block) [On (tr, b) ⇒ ◊ On (tr, b+1)]

A particular case that comes directly to mind is when block *b+1*'s signal is set to 'go'. Two subgoals coming naturally to mind are the following:

Goal Achieve [ProgressWhenGoSignal]
FormalDef ∀ tr: Train, b: Block
On (tr, b) ∧ *Go[b+1]* ⇒ ◊ On (tr, b+1)

Goal Achieve [SignalSetToGo]
FormalDef ∀ tr: Train, b: Block
On (tr, b) ⇒ ◊ Go[b+1]

This tentative refinement matches the decomposition-by-case pattern in Section 5.1 and therefore allows the following missing subgoal to be pointed out:

Goal Maintain [TrainWaiting]
FormalDef ∀ tr: Train, b: Block
On (tr, b) ⇒ On (tr, b) *W* On (tr, b+1)

Another effective way of driving the refinement process is based on the determination that an agent candidate to goal assignment cannot realize the goal, e.g., because it cannot monitor the variables appearing in the goal antecedent or control the variables appearing in the goal consequent. [Let01] gives a set of conditions for goal unrealizability; this set is proved complete and provides the basis for a rich, systematic set of agent-driven refinements tactics for generating realizable subgoals and auxiliary agents.

Goal/requirement elicitation by abstraction

An obvious (but effective) informal technique for finding out more abstract, parent goals is to keep asking WHY questions about operational descriptions already available [Lam95, Lam00c].

More sophisticated techniques have been devised to elicit goals from scenarios. Based on a bidirectional coupling between type-level scenarios and goal verb templates as discussed in Section 4, [Rol98] proposes heuristic rules for finding out alternative goals covering a scenario (corresponding to alternative values for the verb parameters), missing companion goals, or subgoals of the goal under consideration. On a more formal side, [Lam98c] describes an inductive learning technique that takes scenarios as examples and counterexamples of intended behavior and generates goal specifications in temporal logic that cover all the positive scenarios and exclude all the negative ones.

Note also that refinement patterns when applied in the reverse way correspond to abstraction patterns that may produce more coarse-grained goals.

Goal operationalization

A few efforts have been made to support the process of deriving pre-, post-, and trigger conditions on software operations so as to ensure the terminal goals in the refinement process. The principle is to apply derivation rules whose premise match the goal under consideration [Dar93, Let01]. Consider, for example, the following goal:

Goal Maintain [DoorsClosedWhileMoving]
FormalDef ∀ tr: Train, loc, loc': Location
At (tr, loc) ∧ o At (tr, loc') ∧ loc <> loc'
⇒ tr.Doors = 'closed' ∧ o (tr.Doors = 'closed')

where the "o" temporal operator means "in the next state". Applying the following derivation rule

G: P ∧ (P1 ∧ o P2 ⇒ Q1 ∧ o Q2) , *DomPre:* P1, *DomPost:* P2

ReqPre for *G*: Q1 , *ReqPost* for *G*: Q2

we derive the following operationalization:

Operation Move
Input tr: Train; loc, loc': Location ; **Output** At
DomPre At (tr, loc) ∧ loc <> loc'
DomPost At (tr, loc')
RequiredPre for DoorsClosedWhileMoving: *tr.Doors = 'closed'*
RequiredPost for DoorsClosedWhileMoving: *tr.Doors = 'closed'*

Analogical reuse

Goal-based specifications can also be acquired by retrieving structurally and semantically analog specifications in a repository of reusable specification components, and then transposing the specifications found according to the structural and semantic matching revealed by the retrieval process [Mas97].

Obstacle-driven elaboration

First-sketch specifications of goals, requirements and assumptions are often too ideal; they are likely to be violated from time to time in the running system due to unexpected behaviors of agents. The lack of anticipation of exceptional behaviors may result in unrealistic, unachievable and/or incomplete requirements.

Such exceptional behaviors are captured by assertions called *obstacles* to goal satisfaction. An obstacle *O* is said to obstruct a goal *G* in a domain *Dom* iff

$\{O, Dom\} \models \neg G$ *obstruction*

$Dom \not\models \neg O$ *domain consistency*

Obstacles thus need to be identified and resolved at RE time in order to produce robust requirements and hence more reliable software. The notion of obstacle was just mentioned in [Yue87]. It was elaborated further in [Pot95] where scenarios are shown to be a good vehicle for identifying goal obstructions. Some heuristics for identifying obstacles can be found in [Pot95] and [Ant98]. More formal techniques are described in [Lam98a] and then [Lam00a] for:

- the abductive generation of obstacles from goal specifications and domain properties,
- the systematic generation of various types of obstacle resolution, e.g., goal substitution, agent substitution, goal weakening, goal restoration, obstacle mitigation, or obstacle prevention.

Obstacles can also be resolved at run time in some cases, see [Fea98].

5.4 Conflict management

Requirements engineers live in a world where conflicts are the rule, not the exception [Eas94]. Conflicts generally arise from multiple viewpoints and concerns [Nus94]. They must be detected and eventually resolved even though they may be temporarily useful for eliciting further information [Hun98]. Various forms of conflict are studied in [Lam88b], in particular, a weak form called *divergence* which occurs frequently in practice.

The goals $G_1, ..., G_n$ are said to be *divergent* iff there exists a non-trivial *boundary condition B* such that :

$\{ B, \forall i\ G_i, Dom\} \models$ **false** *inconsistency*

$\{ B, \forall_{j \neq i}\ G_j, Dom\} \not\models$ **false** *minimality*

("Non-trivial" means that *B* is different from the bottom **false** and the complement $\neg \forall i\ G_i$). Note that the traditional case of conflict, in the sense of logical inconsistency, amounts to a particular case of divergence.

Divergences need to be identified and resolved at RE time in order to eventually produce consistent requirements. Formal and heuristic techniques are described in [Lam98b] for:

- the abductive generation of boundary conditions from goal specifications and domain properties,
- the systematic generation of various types of divergence resolution.

A qualitative procedure is suggested in [Rob89] for handling conflicts. The idea is to detect them at requirements level and characterize them as differences at goal level. The user of the procedure first identifies the requirements elements that correspond to each other in the various viewpoints at hand; conflict detection is then carried out by mapping syntactic differences between the corresponding requirements elements to differences in values of variables involved in the goals supported by these elements. Conflict resolution is attempted next by appealing to compromises (e.g., through compensations or restriction specialization), or goal substitutions. Finally, the conflict resolution at goal level is down propagated to the requirements level.

5.5 Goal-based negotiation

Conflict resolution often requires negotiation. [Boe95] proposes an iterative 3-step process model for goal-based negotiation of requirements. At each iteration of a spiral model for requirements elaboration,

(1) all stakeholders involved are identified together with their wished goals (called *win conditions*);

(2) conflicts between these goals are captured together with their associated risks and uncertainties;

(3) goals are reconciled through negotiation to reach a mutually agreed set of goals, constraints, and alternatives for the next iteration.

5.6 Alternative selection

Which goal refinement should be selected when alternative ones are identified? Which agent assignment should be selected when alternative ones are identified? This is by and large an open problem. There are local tactics of course, such as favoring alternatives with less critical obstacles or conflicts, but a systematic approach has not emerged so far in the RE literature.

One promising direction would be to use qualitative reasoning schhemes à la NFR [Myl92] to select an alternative refinement that contributes the best to the satisficing of soft goals related to cost, reliability, performance etc. Multicriteria analysis techniques could be helpful here.

6. A goal-oriented RE method in action

It is now time to demonstrate how some of the techniques reviewed above can fit together in a goal-oriented RE method. We come back to a case study we have already presented in [Lam00c] because it illustrates many of the issues raised here; the initial document is unbiased as it comes from an independent source involved in the development,; it is publically available [BAR99] --unlike most documents from the industrial projects we have been involved in; the system is a real, complex, real-time, safety-critical one (this allows one to suggest that goal-oriented RE is not only useful for business applications). The initial document focuses on the control of speed and acceleration of trains under responsibility of the Advanced Automatic Train Control being developed for the San Francisco Bay Area Rapid Transit (BART) system.

We follow the KAOS method [Dar93, Lam95, Lam00c] in order to incrementally elaborate four complementary submodels: (1) the goal model, (2) the object model; (3) the agent responsibility model, leading to alternative system boundaries; (4) the operation model. The goal refinement graph is elaborated by eliciting goals from available sources and asking *why* and *how* questions (*goal elaboration step*); objects, relationships and attributes are derived from the goal specifications (*object modeling step*); agents are identified, alternative responsibility assignments are explored, and agent interfaces are derived (*responsibility assignment step*);

operations and their domain pre- and postconditions are identified from the goal specifications, and strengthened pre-/postconditions and trigger conditions are derived so as to ensure the corresponding goals (*operationalization step*). These steps are not strictly sequential as progress in one step may prompt parallel progress in the next one or backtracking to a previous one.

The presentation will be sketchy for lack of space; the interested reader may refer to [Let01] for a much greater level of details.

Figure 1 - Preliminary goal graph for the BART system

Goal identification from the initial document

A first set of goals is identified from a first reading of the available source [BART99] by searching for intentional keywords such as “objective”, “purpose”, “intent”, “concern”, “in order to”, etc. A number of soft goals are thereby identified, e.g., “ServeMorePassengers”, “NewTracksAdded”, “Minimize[DevelopmentCosts]”, “Minimize[DistanceBetweenTrains]”, “SafeTransportation”, etc. These goals are qualitatively related to each other through support links: Contributes (+), ContributesStrongly (++), Conflicts (-), ConflictsStrongly (- -). These weights are used to select among alternatives. Where possible, keywords from the semi-formal layer of the KAOS language are used to indicate the goal category. The *Maintain* and *Avoid* keywords specify “always” goals having the temporal pattern $\Box(P \rightarrow Q)$ and $\Box(P \rightarrow \neg Q)$, respectively. The *Achieve* keyword specifies “eventually” goals having the pattern $P \Rightarrow \Diamond Q$. The “$\rightarrow$“ connective denotes logical implication; $\Box(P \rightarrow Q)$ is denoted by $P \Rightarrow Q$ for short.

Figure 1 shows the result of this first elicitation. Clouds denote soft-goals, parallelograms denote formalizable goals, arrows denote goal-subgoal links, and a double line linking arrows denotes an OR-refinement into alternative subgoals.

Formalizing goals and identifying objects

The object modeling step can start as soon as goals can be formulated precisely enough. The principle here is to identify objects, relationships and attributes from goal specifications. Consider, for example, the following goal at the bottom of Figure 1:

Goal Maintain[TrackSegmentSpeedLimit]
InformalDef *A train should stay below the maximum speed the track segment can handle.*
FormalDef $\forall$ tr: Train, s: TrackSegment :
On(tr, s) $\Rightarrow$ tr.Speed $\leq$ s.SpeedLimit

From the predicate, objects, and attributes appearing in this goal formalization we derive the following portion of the object model:

Similarly, the other goal at the bottom of Figure 5 is specified as follows:

Goal Maintain[WCS-DistBetweenTrains]
InformalDef *A train should never get so close to a train in front so that if the train in front stops suddenly (e.g., derailment) the next train would hit it.*
FormalDef $\forall$ tr1, tr2: Train :
Following(tr1, tr2) $\Rightarrow$ tr1.Loc - tr2.Loc > tr1.WCS-Dist

(The InformalDef statements in those goal definitions are taken literally from the initial document; *WCS-Dist* denotes the physical worst-case stopping distance based on the physical speed of the train.) This new goal specification allows the above portion of the object model to be enriched with *Loc* and *WCS-Dist* attributes for the *Train* object together with a reflexive *Following* relationship on it. The formalization of the goal Avoid[TrainEnterinClosedGate] in Figure 1 will further enrich the object model by elements that are strictly necessary to the goals considered. *Goals thus provide a precise driving criterion for identifying elements of the object model*.

Eliciting new goals through WHY questions

It is often the case that higher-level goals underpinning goals easily identified from initial sources are kept implicit in such sources. They may, however, be useful for finding out other important subgoals of the higher-level goal that were missing for the higher-level goal to be achieved.

As mentioned before, higher-level goals are identified by asking WHY questions about the goals available.

For example, asking a WHY question about the goal Maintain[WCS-DistBetweenTrains] yields the parent goal Avoid[Train-Collision]; asking a WHY question about the goal Avoid[TrainEnteringClosedGate] yields a new portion of the goal graph, shown in Figure 2.

In this goal subgraph, the companion subgoal Maintain[Gate-ClosedWhenSwitchInWrongPosition] was elicited *formally* by matching a formal refinement pattern to the formalization of the parent goal Avoid[TrainOnSwitchInWrongPosition], found by a WHY question, and to the formalization of the initial goal Avoid[TrainEnteringClosedGate] [Dar96, Let01]. The dot joining the two lower refinement links together in Figure 2

means that the refinement is (provably) complete.

Figure 2 - Enriching the goal graph by WHY elicitation

Eliciting new goals through HOW questions

Goals need to be refined until subgoals are reached that can be assigned to individual agents in the software-to-be and in the environment. Terminal goals become requirements in the former case and assumptions in the latter.

More concrete goals are identified by asking HOW questions. For example, a HOW question about the goal Maintain[WCS-DistBetweenTrains] in Figure 1 yields an extension of the goal graph shown in Figure 3.

Figure 3 - Goal refinement

The formalization of the three subgoals in Figure 3 may be used to prove that together they entail the parent goal Maintain[WCS-DistBetweenTrains] formalized before [Let01]. These subgoals need be refined in turn until assignable subgoals are reached. A complete refinement tree is given in Annex 1.

Identifying potential responsibility assignments

Annex 1 also provides a possible goal assignment among individual agents. This assignment seems the one suggested in the initial document [BAR99]. For example, the accuracy goal Maintain[AccurateSpeed/PositionEstimates] is assignable to the TrackingSystem agent; the goal Maintain[SafeTrainResponse-ToCommand] is assignable to the OnBoardTrainController agent; the goal Maintain[SafeCmdMsg] is assignable to the Speed/AccelerationControlSystem agent.

It is worth noticing that goal refinements and agent assignments are both captured by AND/OR relationships. Alternative refinements and assignments can be (and probably have been) explored. For example, the parent goal Maintain[WCS-DistBetweenTrains] in Figure 3 may alternatively be refined by the following three Maintain subgoals:

PreceedingTrainSpeed/PositionKnownToFollowingTrain
SafeAccelerationBasedOnPreceedingTrainSpeed/Position
NoSuddenStopOfPreceedingTrain

The second subgoal above could be assigned to the OnBoard-TrainController agent. This alternative would give rise to a fully distributed system.

As suggested before, qualitative reasoning techniques in the style of [Myl99] might be applied to the softgoals identified in Figure 1 to help making choices among alternatives.

Deriving agent interfaces

Let us now assume that the goal Maintain[SafeCmdMsg] at the bottom of the tree in Annex 1 has been actually assigned to the Speed/AccelerationControlSystem agent. The interfaces of this agent in terms of monitored and controlled variables can be derived from the formal specification of this goal (we just take its general form here for sake of simplicity):

Goal Maintain[SafeCmdMsg]
FormalDef $\forall$ cm: *CommandMessage*, ti1, ti2: *TrainInfo*
cm.Sent $\wedge$ cm.TrainID = ti1.TrainID $\wedge$ FollowingInfo (ti1, ti2)
$\Rightarrow$ cm.Accel $\leq$ F (ti1, ti2) $\wedge$ cm.Speed > G (ti1)

To fulfil its responsibility for this goal the Speed/Acceleration-ControlSystem agent must be able to *evaluate* the goal antecedent and *establish* the goal consequent. The agent's monitored variable is therefore *Train.Info* whereas its controlled variables are *CommandMessage.Accel* and *CommandMessage.Speed*. The latter will in turn become monitored variables of the OnBoardTrainController agent, by similar analysis. The technique for deriving the agent's monitored and controlled variables is fairly systematic, see [Let01] for details.

Identifying operations

The operationalization step starts by identifying the operations relevant to goals and defining their domain pre- and postconditions. Goals refer to specific state transitions; for each such transition an operation causing it is identified; its domain pre- and postcondition capture the state transition. For the goal Maintain[SafeCmdMsg] formalized above we get, for example,

Operation SendCommandMessage
Input Train {**arg** tr}
Output ComandMessage {**res** cm}
DomPre $\neg$ cm.Sent
DomPost cm.Sent $\wedge$ cm.TrainID = tr.ID

This definition minimally captures what any sending of a command to a train is about in the domain considered; it does not ensure any of the goals it should contribute to.

Operationalizing goals

The next operationalization sub-step is to strengthen such domain conditions so that the various goals linked to the operation are ensured. For goals assigned to software agents, this step produces *requirements* on the operations for the corresponding goals to be achieved. As mentioned before, derivation rules for an operationalization calculus are available [Dar93, Let01]. In our example, they yield the following requirements that strengthen the domain pre- and postconditions:

Operation SendCommandMessage
Input Train {**arg** tr}, TrainInfo; **Output** ComandMsg {**res** cm}
DomPre ... ; **DomPost** ...
ReqPost for SafeCmdMsg:
Tracking (ti1, tr) $\wedge$ Following (ti1, ti2)
$\rightarrow$ cm.Acc $\leq$ F (ti1, ti2) $\wedge$ cm.Speed $>$ G (ti1)
ReqTrig for CmdMsgSentInTime:
$\blacksquare_{\leq 0.5\ sec}\ \neg\ \exists$ cm2: CommandMessage:
cm2.Sent $\wedge$ cm2.TrainID = tr.ID

(The trigger condition captures an obligation to trigger the operation as soon as the condition gets true and provided the domain precondition is true. In the example above the condition says that no command has been sent in every past state up to one half-second [BAR99].)

Using a mix of semi-formal and formal techniques for goal-oriented requirements elaboration, we have reached the level at which most formal specification techniques would start.

Anticipating obstacles

As mentioned before, goals also provide a basis for early generation of high-level exceptions which, if handled properly at requirements engineering time, may generate new requirements for more robust systems.

The following obstacles were generated to obstruct the subgoal Achieve[CommandMsgIssuedInTime]:

CommandMsgNotIssued,
CommandMsgIssuedLate,
CommandMsgSentToWrongTrain

For the companion subgoal Achieve[CommandMsgDeliveredInTime] we similarly generated obstacles such as:

CommandMsgDeliveredLate,
CommandMsgCorrupted

The last companion subgoal Maintain[SafeCmdMsg] may be obstructed by the condition

UnsafeAcceleration,

and so on. The obstacle generation process for a single goal results in a goal-anchored fault-tree, that is, a refinement tree whose root is the goal negation. Compared with standard fault-tree analysis [Lev95], obstacle analysis is goal-oriented, formal, and produces obstacle trees that are provably complete with respect to what is known about the domain [Lam00a].

Alternative obstacle resolutions may then be generated to produce new or alternative requirements. For example, the obstacle CommandMsgSentLate above could be resolved by an alternative design in which accelerations are calculated by the on-board train controller instead; this would correspond to a *goal substitution* strategy. The obstacle UnsafeAcceleration above could be resolved by assigning the responsibility for the subgoal SafeAccelerationCommanded of the goal Maintain[SafeCmdMsg] to the VitalStationComputer agent instead [BART99]; this would correspond to an *agent substitution* strategy. An *obstacle mitigation* strategy could be applied to resolve the obstacle OutOfDateTrainInfo obstructing the accuracy goal Maintain[AccurateSpeed/PositionEstimates], by introducing a new subgoal of the goal Avoid[TrainCollisions], namely, the goal Avoid[CollisionWhenOutOfDateTrainInfo]. This new goal has to be refined in turn, e.g., by subgoals requiring full braking when the message origination time tag has expired.

Handling conflicts

The initial BART document suggests an interesting example of divergence [BART99, p.13]. Roughly speaking, the train commanded speed may not be too high, because otherwise it forces the distance between trains to be too high, in order to achieve the DistanceIncreasedWithCommandedSpeed subgoal of the SafeTransportation goal; on the other hand, the commanded speed may not be too low, in order to achieve the LimitedAccelerAbove7mphOfPhysicalSpeed subgoal of the SmoothMove goal. There seems to be a flavor of divergence here.

Figure 4 - Conflict in speed/acceleration control

We therefore look at the formalization of the suspect goals:

Goal Maintain [CmdedSpeedCloseToPhysicalSpeed]
FormalDef $\forall$ tr: Train
$tr.Acc_{CM} \geq 0$
$\Rightarrow tr.Speed_{CM} \leq tr.Speed + f(\text{dist-to-obstacle})$

and

Goal Maintain [CmdedSpeedAbove7mphOfPhysicalSpeed]
FormalDef $\forall$ tr: Train
$tr.Acc_{CM} \geq 0 \Rightarrow tr.Speed_{CM} > tr.Speed + 7$

These two goals are formally detected to be divergent using the techniques described in [Lam98b]. The generated boundary condition for making them logically inconsistent is

$\Diamond\ (\exists$ tr: Train$)\ (tr.Acc_{CM} \geq 0 \wedge f(\text{dist-to-obstacle}) \leq 7)$

The resolution operators from [Lam98b] may be used to generate possible resolutions; in this case one should keep the safety goal as it is and weaken the other conflicting goal to remove the divergence:

Goal Maintain [CmdedSpeedAbove7mphOfPhysicalSpeed]
FormalDef $\forall$ tr: Train
$tr.Acc_{CM} \geq 0 \Rightarrow tr.Speed_{CM} > tr.Speed + 7$
$\vee\ f(\text{dist-to-obstacle}) \leq 7$

7. Experience and tool support

The purrpose of this paper is obviously not to deliver an experience report. We would just like to mention here that

experience with goal-oriented requirements engineering is growing significantly, in different domain, different types of projects, and different project sizes. For example, Anton and colleagues have reported their experience with BPR applications [Ant94] and various electronic commerce systems [Ant98, Ant01]. Our understanding is that the NFR and i* frameworks have been experienced in real settings as well.

Our KAOS method has been used in 11 industrial projects to date. These include the goal-oriented reengineering of a complex, unintelligible requirements document for a phone system on TV cable; the goal-oriented modeling of a complex air traffic control application; the goal-oriented engineering of requirements for a variety of systems such as: a copyright management system for a major editor of cartoon strips, a management system for a hospital emergency service, a drug delivery management system for a big drug distributor, a new information system for a big daily newspaper, a web-based job information server, a web-based language translation system, and various e-learning systems. To give an idea, the copyright management system has 65 goals, 75 entity types and relationships, 11 agents, and 45 operations; the goal-oriented deliverable is 115 pages long. The size of the goal refinement graph for the other applications ranges from 50 to 100 goals and requirements.

Those projects could not have been undertaken without tool support. Our current GRAIL environment provides a graphical editor tightly coupled with a syntax-directed editor, an object-oriented specification database server supporting queries for model analysis, static semantics checkers, view filtering mechanisms, a HTML generator for model browsing in hypertext mode, and various types of report generators. Current efforts are devoted to an open, full Java version; the plan then is to integrate more formal support such as animators, model checkers, test data generators, formal verification tools, and so forth.

8. Goal orientation beyond RE

It has been suggested recently that the functional and (especially) non-functional goals elaborated in the RE process could be used for deriving and refining architectures [Lam00c] and for annotating design patterns [Chu00]. These are just preliminary efforts that should be expanded in a near future.

9. Conclusion

Goal-oriented requirements engineering has many advantages, some of which were recurrently felt in the aforementioned projects, to restate a few of them:

- object models and requirements can be derived systematically from goals;
- goals provide the rationale for requirements;
- a goal graph provides vertical traceability from high-level strategic concerns to low-level technical details; it allows evolving versions of the system under consideration to be integrated as alternatives into one single framework;
- goal AND/OR graphs provide the right abstraction level at which decision makers can be involved for important decisions;
- the goal refinement structure provides a comprehensible structure for the requirements document;
- alternative goal refinements and agent assignments allow alternative system proposals to be explored;
- goal formalization allows refinements to be proved correct and complete.

We hope to have convinced the reader that this area of RE is worth pursuing. There are many open issues to work on in the future, of course; the reader may refer to [Lam00c] for a discussion of them.

Acknowledgment. Discussions with Robert Darimont and Emmanuel Letier were a permanent source of inspiration and confrontation of some of the issues raised in this paper; they were in particular instrumental in developing KAOS specifications for various non-trivial systems, including the one outlined here [Let01]. I am also grateful to the KAOS/GRAIL crew at CEDITI for using some of the ideas presented here in industrial projects and providing regular feedback, among others, Emmanuelle Delor, Philippe Massonet, and André Rifaut. All the people whose work is mentioned in this paper had some influence on it in some way or another (whether they recognize and like it or not!).

References

[Amo94] E.J. Amoroso, Fundamentals of Computer Security. Prentice-Hall, 1994.

[And89] J.S. Anderson and S. Fickas, "A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem", *Proc. 5th Intl. Workshop on Software Specification and Design*, IEEE, 1989, 177-184.

[Ant94] A.I. Anton, W.M. McCracken, and C. Potts, "Goal Decomposition and Scenario Analysis in Business Process Reengineering, *Proc. CAISE'94*, LNCS 811, Springer-Verlag, 1994, 94-104.

[Ant98] A.I. Anton and C. Potts, “The Use of Goals to Surface Requirements for Evolving Systems”, *Proc. ICSE-98: 20th Intrnational Conference on Software Enginering*, Kyoto, April 1998.

[Ant01] A.I. Anton, R. Carter, A. Dagnino, J. Dempster and D.F. Siege, “Deriving Goals from a Use-Case Based Requirements Specification”, *Requirements Engineering Journal*, Vol. 6, 2001, 63-73.

[BAR99] Bay Area Rapid Transit District, Advance Automated Train Control System, Case Study Description. Sandia National Labs, http://www.hcecs.sandia.gov/bart.htm.

[Ber91] V. Berzins and Luqi, *Software Engineering with Abstractions*. Addison-Wesley, 1991.

[Boe95] B. W. Boehm, P. Bose, E. Horowitz, and Ming June Lee, “Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach”, *Proc. ICSE-17 - 17th Intl. Conf. on Software Engineering*, Seattle, 1995, pp. 243-253.

[Chu00] L. Chung, B. Nixon, E. Yu and J. Mylopoulos, *Non-functional requirements in software engineering*. Kluwer Academic, Boston, 2000.

[Dar91] A. Dardenne, S. Fickas and A. van Lamsweerde, “Goal-Directed Concept Acquisition in Requirements Elicitation”, *Proc. IWSSD-6 - 6th Intl. Workshop on Software Specification and Design*, Como, 1991, 14-21.

[Dar93] A. Dardenne, A. van Lamsweerde and S. Fickas, “Goal-Directed Requirements Acquisition”, *Science of Computer Programming*, Vol. 20, 1993, 3-50.

[Dar96] R. Darimont and A. van Lamsweerde, “Formal Refinement Patterns for Goal-Driven Requirements Elaboration”, *Proc. FSE’4 - Fourth ACM SIGSOFT Symp. on the Foundations of Software Engineering*, San Francisco, October 1996, 179-190.

[Dar98]R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde, “GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering”, *Proc. ICSE’98 - 20th Intl. Conf. on Software Engineering*, Kyoto, April 1998, vol. 2, 58-62. (Earlier and shorter version found in *Proc. ICSE’97 - 19th Intl. Conf. on Software Engineering*, Boston, May 1997, 612-613.)

[Dub98] E. Dubois, E. Yu and M. Petit, "From Early to Late Formal Requirements: A Process-Control Case Study”, *Proc. IWSSD’98 - 9th International Workshop on Software Specification and Design*, Isobe, IEEE CS Press, April 1998, 34-42.

[Dwy99] M.B. Dwyer, G.S. Avrunin and J.C. Corbett, “Patterns in Property Specifications for Finite-State Verification”, *Proc. ICSE-99: 21th Intrnational Conference on Software Enginering*, Los Angeles, 411-420.

[Eas94] S. Easterbrook, “Resolving Requirements Conflicts with Computer-Supported Negotiation”. In *Requirements Engineering: Social and Technical Issues*, M. Jirotka and J. Goguen (Eds.), Academic Press, 1994, 41-65.

[Fea87] M. Feather, “Language Support for the Specification and Development of Composite Systems”, *ACM Trans. on Programming Languages and Systems* 9(2), Apr. 87, 198-234.

[Fea93] M. Feather, "Requirements Reconnoitering at the Juncture of Domain and Instance", *Proc. RE’93 - 1st Intl. IEEE Symp. on Requirements Engineering*, Jan. 1993, 73-77.

[Fea98] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, “Reconciling System Requirements and Runtime Behaviour”, *Proc. IWSSD’98 - 9th International Workshop on Software Specification and Design*, Isobe, IEEE CS Press, April 1998.

[Fic92] S. Fickas and R. Helm, “Knowledge Representation and Reasoning in the Design of Composite Systems", *IEEE Trans. on Software Engineering*, June 1992, 470-482.

[Fin87] A. Finkelstein and C. Potts, "Building Formal Specifications Using Structured Common Sense", Proc. IWSSD-4 - 4th International Workshop on Software Specification and Design (Monterey, Ca.), IEEE, April 1987, 108-113.

[Fow97] M. Fowler, *UML Distilled*. Addison-Wesley, 1997.

[Got95] O. Gotel and A. Finkelstein, “Contribution Structures”, *Proc. RE’95 - 2nd Intl. IEEE Symp. on Requirements Engineering*, York, IEEE, 1995, 100-107.

[Gro01] D. Gross and E. Yu, “From Non-Functional Requirements to Design through Patterns”, *Requirements Engineering Journal* Vol. 6, 2001, 18-36.

[Hau98] P. Haumer, K. Pohl, and K. Weidenhaupt, “Requirements Elicitation and Validation with Real World Scenes”, *IEEE Trans. on Sofware. Engineering*, Special Issue on Scenario Management, December 1998, 1036-1054.

[Hey98] P. Heymans and E. Dubois, “Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Requirements”, *Requirements Engineering Journal* Vol. 3 No. 3-4, 1998, 202-218.

[Hic74]G.F. Hice, W.S. Turner, and L.F. Cashwell, *System Development Methodology*. North Holland, 1974.

[Hun98] A. Hunter and B. Nuseibeh, “Managing Inconsistent Specifications: Reasoning, Analysis and Action”, *ACM Transactions on Software Engineering and Methodology*, Vol. 7 No. 4. October 1998, 335-367.

[Jac95] M. Jackson, Software Requirements & Specifications - A Lexicon of Practice, Principles and Pejudices. ACM Press, Addison-Wesley, 1995.

[Jar93] M. Jarke and K. Pohl, “Vision-Driven Requirements Engineering”, *Proc. IFIP WG8.1 Working Conference on Information System Development Process*, North Holland, 1993, 3-22.

[Kai00] H. Kaindl, “A Design Process Based on a Model Combining Scenarios with Goals and Functions”, *IEEE Trans. on Systems, Man and Cybernetic*, Vol. 30 No. 5, September 2000, 537-551.

[Kel90] S.E. Keller, L.G. Kahn and R.B. Panara, "Specifying Software Quality Requirements with Metrics", in Tutorial: System and Software Requirements Enginering, R.H. Thayer and M. Dorfman, Eds., IEEE Computer Society Press, 1990, 145-163.

[Koy92] R. Koymans, *Specifying message passing and time-critical systems with temporal logic*, LNCS 651, Springer-Verlag, 1992.

[Lam95] A. van Lamsweerde, R. Darimont, and Ph. Massonet, "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt", *Proc. RE’95 - 2nd Intl. IEEE Symp. on Requirements Engineering*, March 1995, 194-203.

[Lam98a] A. van Lamsweerde and E. Letier, “Integrating Obstacles in Goal-Driven Requirements Engineering”, *Proc. ICSE-98: 20th Intrnational Conference on Software Enginering*, Kyoto, April 1998.

[Lam98b] A. van Lamsweerde, R. Darimont and E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Trans. on Sofware. Engineering*, Special Issue on Inconsistency Management in Software Development, November 1998.

[Lam98c] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios", *IEEE Trans. on Sofware. Engineering*, Special Issue on Scenario Management, December 1998, 1089-1114.

[Lam00a] A. van Lamsweerde and E. Letier, “Handling Obstacles in Goal-Oriented Requirements Engineering”, *IEEE Transactions on Software Engineering*, Special Issue on Exception Handling, 2000.

[Lam00b] A. van Lamsweerde, “Formal Specification: a Roadmap”. In *The Future of Software Engineering*, A. Finkelstein (ed.), ACM Press, 2000.

[Lam00c] A. van Lamsweerde, “Requirements Engineering in the Year 00: A Research Perspective”. Invited Keynote Paper, *Proc. ICSE’2000: 22nd International Conference on Software Engineering*, ACM Press, 2000, pp. 5-19.

[Lee91] J. Lee, "Extending the Potts and Bruns Model for Recording Design Rationale", *Proc. ICSE-13 - 13th Intl. Conf. on Software Engineering*, IEEE-ACM, 1991, 114-125.

[Lei97] J.C. Leite, G. Rossi, F. Balaguer, V. Maiorana, G. Kaplan, G. Hadad and A. Oliveiros, “Enhancing a Requirements Baseline with Scenarios”, *Requirements Engineering Journal* Vol. 2 No. 4, 1997, 184-198.

[Let01] E. Letier, *Reasoning about Agents in Goal-Oriented Requirements Engineering*. Ph. D. Thesis, University of Louvain, May 2001.

[Lev95] N. Leveson, *Safeware - System Safety and Computers*. Addison-Wesley, 1995.

[Man92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992.

[Man96] Z. Manna and the STep Group, “STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems”, *Proc. CAV’96 - 8th Intl. Conf. on Computer-Aided Verification*, LNCS 1102, Springer-Verlag, July 1996, 415-418.

[Mas97] P. Massonet and A. van Lamsweerde, “Analogical Reuse of Requirements Frameworks”, Proc. RE-97 - *3rd Int. Symp. on Requirements Engineering*, Annapolis, 1997, 26-37.

[Mos85] J. Mostow, "Towards Better Models of the Design Process", AI Magazine, Vol. 6, 1985, pp. 44-57.

[Mun81] E. Munford, "Participative Systems Design: Structure and Method", *Systems, Objectives, Solutions*, Vol. 1, North-Holland, 1981, 5-19.

[Myl92] Mylopoulos, J., Chung, L., Nixon, B., “Representing and Using Nonfunctional Requirements: A Process-Oriented Approach”, *IEEE Trans. on Sofware. Engineering*, Vol. 18 No. 6, June 1992, pp. 483-497.

[Myl99] J. Mylopoulos, L. Chung and E. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis", Communications of the ACM, Vol. 42 No. 1, January 1999, 31-37.

[Nil71] N.J. Nilsson, *Problem Solving Methods in Artificial Intelligence*. McGraw Hill, 1971.

[Nix93] B. A. Nixon, "Dealing with Performance Requirements During the Development of Information Systems", Proc. RE’93 - 1st Intl. IEEE Symp. on Requirements Engineering, Jan. 1993, 42-49.

[Nus94] B. Nuseibeh, J. Kramer and A. Finkelstein, "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications", *IEEE Transactions on Software Engineering*, Vol. 20 No. 10, October 1994, 760-773.

[Par95] D.L. Parnas and J. Madey, “Functional Documents for Computer Systems”, *Science of Computer Programming*, Vol. 25, 1995, 41-61.

[Pot94] C. Potts, K. Takahashi and A.I. Anton, "Inquiry-Based Requirements Analysis", *IEEE Software*, March 1994, 21-32.

[Pot95] C. Potts, “Using Schematic Scenarios to Understand User Needs”, *Proc. DIS’95 - ACM Symposium on Designing interactive Systems: Processes, Practices and Techniques*, University of Michigan, August 1995.

[Rob89] Robinson, W.N., “Integrating Multiple Specifications Using Domain Goals”, *Proc. IWSSD-5 - 5th Intl. Workshop on Software Specification and Design*, IEEE, 1989, 219-225.

[Rol98] C. Rolland, C. Souveyet and C. Ben Achour, “Guiding Goal Modeling Using Scenarios”, *IEEE Trans. on Sofware. Engineering*, Special Issue on Scenario Management, December 1998, 1055-1071.

[Ros77] D.T. Ross and K.E. Schoman, "Structured Analysis for Requirements Definition", *IEEE Transactions on Software Engineering*, Vol. 3, No. 1, 1977, 6-15.

[Rub92] K.S. Rubin and A. Goldberg, "Object Behavior Analysis", Communications of the ACM Vol. 35 No. 9, September 1992, 48-62.

[Som97] I. Sommerville and P. Sawyer, *Requirements Engineering: A Good Practice Guide*. Wiley, 1997.

[Sut93] A. Sutcliffe and N. Maiden, “Bridging the Requirements Gap: Policies, Goals and Domains”, *Proc. IWSSD-7 - 7th Intl. Workshop on Software Specification and Design*, IEEE, 1993.

[Sut98] A. Sutcliffe, “Scenario-Based Requirements Analysis”, *Requirements Engineering Journal* Vol. 3 No. 1, 1998, 48-65.

[Swa82] W. Swartout and R. Balzer, "On the Inevitable Intertwining of Specification and Implementation", Communications of the ACM, Vol. 25 No. 7, July 1982, 438-440.

[Yue87] K. Yue, “What Does It Mean to Say that a Specification is Complete?”, *Proc. IWSSD-4, Fourth International Workshop on Software Specification and Design*, Monterey, 1987.

[Yu93] E.S.K. Yu, "Modelling Organizations for Information Systems Requirements Engineering", *Proc. RE'93 - 1st Intl Symp. on Requirements Engineering*, IEEE, 1993, 34-41.

[Yu97] E. Yu, “Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering”, *Proc. RE-97 - 3rd Int. Symp. on Requirements Engineering*, Annapolis, 1997, 226-235.

[Zav97a] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering", *ACM Transactions on Software Engineering and Methodology*, 1997, 1-30.

[Zav97b] P. Zave, “Classification of Research Efforts in Requirements Engineering”, *ACM Computing Surveys*, Vol. 29 No. 4, 1997, 315-321.

ANNEX 1: GOAL REFINEMENT TREE AND RESPONSIBILITY ASSIGNMENT IN THE BART SYSTEM

Ontology-Driven Guidance for Requirements Elicitation

Stefan Farfeleder[1], Thomas Moser[2], Andreas Krall[1], Tor Stålhane[3], Inah Omoronyia[4], and Herbert Zojer[5]

[1] Institute of Computer Languages, Vienna University of Technology
{stefanf,andi}@complang.tuwien.ac.at
[2] Christian Doppler Laboratory "Software Engineering Integration for Flexible Automation Systems", Vienna University of Technology
thomas.moser@tuwien.ac.at
[3] Department of Computer and Information Science, Norwegian University of Science and Technology
stalhane@idi.ntnu.no
[4] Irish Software Engineering Research Centre, University of Limerick
inah.omoronyia@lero.ie
[5] Infineon Technologies Austria AG
herbert.zojer@infineon.com

Abstract. Requirements managers aim at keeping their sets of requirements well-defined, consistent and up to date throughout a project's life cycle. Semantic web technologies have found many valuable applications in the field of requirements engineering, with most of them focusing on requirements analysis. However the usability of results originating from such requirements analyses strongly depends on the quality of the original requirements, which often are defined using natural language expressions without meaningful structures. In this work we present the prototypic implementation of a semantic guidance system used to assist requirements engineers with capturing requirements using a semi-formal representation. The semantic guidance system uses concepts, relations and axioms of a domain ontology to provide a list of suggestions the requirements engineer can build on to define requirements. The semantic guidance system is evaluated based on a domain ontology and a set of requirements from the aerospace domain. The evaluation results show that the semantic guidance system effectively supports requirements engineers in defining well-structured requirements.

Keywords: requirements elicitation, domain ontology, elicitation guidance, requirements engineering.

1 Introduction

A major goal of requirements engineering is to achieve a common understanding between all project stakeholders regarding the set of requirements. Modern IT projects are complex due to the high number and complexity of requirements, as well as due to geographically distributed project stakeholders with different

G. Antoniou et al. (Eds.): ESWC 2011, Part II, LNCS 6644, pp. 212–226, 2011.
© Springer-Verlag Berlin Heidelberg 2011

backgrounds and terminologies. Therefore, adequate requirements management tools are a major contribution to address these challenges. Current requirements management tools typically work with a common requirements database, which can be accessed by all stakeholders to retrieve information on requirements content, state, and interdependencies.

Requirements management tools help project managers and requirements engineers to keep the overview on large amounts of requirements by supporting: (a) Requirements categorization by clustering requirements into user-defined subsets to help users find relevant requirements more quickly, e.g., by sorting and filtering attribute values; (b) Requirements conflict analysis (or consistency checking) by analyzing requirements from different stakeholders for symptoms of inconsistency, e.g., contradicting requirements; and (c) Requirements tracing by identifying dependencies between requirements and artifacts to support analyses for change impact and requirements coverage. Unfortunately, requirements management suffers from the following challenges and limitations:

- Incompleteness [5] of requirements categorization and conflict identification, in particular, when performed manually.
- High human effort for requirements categorization, conflict analysis and tracing, especially with a large number of requirements [5].
- Tracing on syntactic rather than on concept level: requirements are often traced on the syntactic level by explicitly linking requirements to each other. However, requirements engineers actually want to trace concepts, i.e., link requirements based on their meaning, which can be achieved only partially by information retrieval approaches like "keyword matching" [11][12].

The use of semantic technologies seems promising for addressing these challenges: Ontologies provide the means for describing the concepts of a domain and the relationships between these concepts in a way that allows automated reasoning [18]. Automated reasoning can support tasks such as requirements categorization, requirements conflict analysis, and requirements tracing. While these are very valuable efforts, we think what is missing here is additionally having a proactive and interactive guidance system that tries to improve requirements quality while actually eliciting requirements.

In this work we present the prototypic implementation of a semantic guidance system used to assist requirements engineers with capturing requirements using a semi-formal representation. Compared to the usual flow - write requirements, analyze requirements using the domain ontology, improve requirements - our approach directly uses the domain ontology knowledge in a single step. The semantic guidance system uses concepts, relations and axioms of domain ontologies to provide a list of suggestions the requirements engineer can build on to define requirements.

We evaluate the proposed semantic guidance system based on a domain ontology and a set of requirements from the aerospace domain. The evaluation results show that the semantic guidance system supports the requirements engineer in defining well-structured requirements. The tool managed to provide useful suggestions for filling missing parts of requirements in the majority of the cases (>85%).

This work is organized in the following way: Section 2 presents related work. Section 3 motivates our research; section 4 introduces our approach to ontology-based guidance. Section 5 presents an evaluation of the tool and section 6 concludes and gives ideas about future work.

2 Related Work

This section summarizes related work, going from the broad field of requirements engineering to the more specific areas of elicitation guidance and finally pattern-based requirements.

2.1 Requirements Engineering

Requirements Engineering is a discipline that deals with understanding, documenting, communicating and implementing customers' needs. Thus, insufficient understanding and management of requirements can be seen as the biggest cause of project failure. In order to improve this situation a systematic process to handle requirements is needed [8]. The main activities of a requirements engineering process can be defined as follows [15]:

- **Requirements Elicitation**. Requirements elicitation involves technical staff working with customers to find out about the application domain, the services the system should provide and the system's operational constraints. The goal is to gather raw requirements.
- **Requirements Analysis and Negotiation**. Requirements analysis and negotiation is an activity which aims to discover problems and conflicts with the requirements and reach agreement on changes to satisfy all system stakeholders (people that are affected by the proposed system). The final goal is to reach a common understanding of the requirements between all project participants.
- **Requirements Documentation and Validation**. The defined requirements, written down in a software requirements specification, are validated against criteria like correctness, completeness, consistency, verifiability, unambiguity, traceability, etc.
- **Requirements Management**. Requirements management consists of managing changes of requirements (keeping them consistent), e.g., by ensuring requirements traceability (identification of interdependencies between requirements, other requirements, and artifacts).

These four steps can be summarized as requirements development. In addition, requirements management is a supporting discipline to control all requirements and their changes during the development life cycle and to identify and resolve inconsistencies between the requirements and the project plan and work products. One important method of requirements management is requirements tracing. Traceability can be defined as the *degree to which a relationship between*

two or more products of the development process can be established [1]. Gotel [7] and Watkins [21] describe why requirements tracing can help project managers in verification, cost reduction, accountability, change management, identification of conflicting requirements and consistency checking of models.

2.2 Elicitation Guidance

There are several approaches to guide users to specify requirements. PROPEL [3] is a tool that provides guidance to define property specifications which are expressed as finite-state automata. For the definition of a property the user is guided by a *question tree*, a hierarchical sequence of questions. There are separate scope trees for a property's behavior and its scope. Based on the answers, the tool chooses an appropriate property template. A team of medical personnel and computer scientists used the tool to formulate properties of medical guidelines.

Kitamura et al. [14] present a requirements elicitation tool that improves requirements quality by analysis. The tool analyzes natural language requirements and finds domain ontology entities occurring in the statements. According to these occurrences and their relations in the ontology, requirements are analyzed in terms of completeness, correctness, consistency and unambiguity. Suggestions are provided to the user in order to improve these metrics, e.g., if the tool finds that a domain concept is not defined in the requirements set, the suggestion would be to add a requirement about the missing concept.

REAS [6] is a tool that interactively detects imprecisions in natural language requirements. It consists of a spelling checker, a rule imposer, a lexical analyzer and a parser. Some of the rules enforced by the tool are writing short requirements, using active voice instead of passive and avoiding possessive pronouns (e.g., "it" and "its"). If the tool detects a violation of a rule, the requirements engineer is asked to improve the offending expression.

Compared to our own approach, PROPEL only deals with a rather specific kind of requirements. The other two approaches try to identify weaknesses after the requirements have been defined and do not actively propose wording while writing them.

2.3 Pattern-Based Requirements

Hull, Jackson and Dick [9] first used the term *boilerplate* to refer to a textual requirement template. A boilerplate consists of a sequence of attributes and fixed syntax elements. As an example, a common boilerplate is "⟨*system*⟩ `shall` ⟨*action*⟩". In this boilerplate ⟨*system*⟩ and ⟨*action*⟩ are attributes and `shall` is a fixed syntax element. It is possible to combine several boilerplates by means of simple string concatenation. This allows keeping the number of required boilerplates low while at the same time having a high flexibility. During instantiation textual values are assigned to the attributes of the boilerplates; a boilerplate requirement is thus defined by its boilerplates and its attribute values. The

authors did not propose a fixed list of boilerplates[1] but instead envisioned a flexible language that can be adapted or enriched when necessary.

Stålhane, Omoronyia and Reichenbach [20] extended boilerplates with a domain ontology by linking attribute values to ontology concepts. They adapted the requirements analyses introduced by Kaiya [13] to boilerplate requirements and added a new analysis called opacity. The requirement language used in this work is based on their combination of boilerplates and the domain ontology.

Ibrahim et al. [10] use boilerplate requirements in their work about requirements change management. They define a mapping from boilerplate attributes to software design artifacts (e.g., classes, attributes, operations) and add traceability links between requirements and artifacts accordingly. There are several other pattern based languages similar to boilerplates, e.g., requirements based on EBNF grammars [19]. Denger et al. [4] propose natural language patterns to specify requirements in the embedded systems domain. They include a metamodel for requirement statements and one for events and reactions which they use to check the completeness of the pattern language. Compared to boilerplate requirements, their patterns seem to be a bit less generic, e.g., some of the nonfunctional requirements used in our evaluation would be impossible to express.

Matsuo, Ogasawara and Ohnishi [16] use controlled natural language for requirements, basically restraining the way in which simple sentences can be composed to more complex ones. They use a frame model to store information about the domain. There are three kind of frames. The noun frame classifies a noun into one of several predefined categories. The case frame classifies verbs into operations and contains the noun types which are required for the operation. Finally the function frame represents a composition of several simple operations. The authors use these frames to parse requirements specifications, to organize them according to different viewpoints and to check requirements completeness. In contrast to domain ontologies, the frame-based approach seems to be harder to understand and to adapt by non-experts.

3 Research Issues

There have been presented several approaches to use ontologies to analyze requirements. These approaches try to measure quality aspects like completeness, correctness and consistency on a set of requirements. In [20] there is an analysis called *opacity* that basically checks if, for two concepts occurring in a requirement, there is a relation between them in the domain ontology. A conclusion of our review of this analysis was that, rather than first writing an incorrect requirement, then analyzing and improving it, a better approach would be to actually suggest the very same domain information which is used for the opacity analysis to the requirements engineer in the first place. There are two points to this idea:

[1] J. Dick maintains a list of suggestions at `http://freespace.virgin.net/gbjedi/books/re/boilerplates.htm` though.

- We want a system that automatically proposes at least parts of the requirements by using information originating from a domain ontology.
- We want a system that exploits relations and axioms of the domain ontology, i.e., a system that is more powerful than just a simple dictionary. The relations and axioms of the domain ontology represent agreed upon knowledge of stakeholders; by using them we hope to improve the precision of requirements.

We believe that a tool supporting these two points will increase efficiency by speeding up the process to get a high-quality requirements specification.

A semantic guidance system implementing these two points was added to our boilerplates elicitation tool (DODT). To test our assumption, we used the tool on requirements from the domain of the *Doors Management System* (DMS). The DMS is a use case developed by EADS for experimenting with and evaluating requirements engineering and modeling techniques within the CESAR project[2]. The DMS controls the doors of an airplane; its main objective is to lock the doors of an airplane while it is moving (in the air or on the ground) and to unlock them when the airplane is parked on the ground. The system consists of sensors that measure the state of the doors, actuators to lock and unlock the doors and computing elements that control the entire system.

4 Guidance for Boilerplate Requirements

This section presents our approach for guiding the requirement engineer using information of a domain ontology.

4.1 Boilerplate Requirements Elicitation

Figure 1 shows how requirements elicitation works using the boilerplates method. To specify a requirement, one or more boilerplates are chosen by the requirements engineer. The attribute values refer to entities in the domain ontology. During instantiation the attributes of the chosen boilerplates are set and a final boilerplate-based requirement is produced. The semantic guidance system affects the domain ontology, the attribute values and the instantiation. Its purpose is to suggest potential and suitable values for the attributes to the requirements engineer.

Table 1 lists the boilerplate attributes implemented by the tool. We reduced the number of attributes compared to the original suggestions in [9] and [20]. This was done in accordance with user wishes who were uncomfortable with the subtile differences between similar attributes and had problems deciding on which to use.

[2] `http://cesarproject.eu/`

Fig. 1. Boilerplate Requirements Elicitation Flow

Table 1. Boilerplate Attributes and Values

Attribute	Description	Example Value
⟨*action*⟩	A behavior that is expected to be fulfilled by the system, or a capability	open the door
⟨*entity*⟩	A separate entity in the domain; not fitting into ⟨*user*⟩ or ⟨*system*⟩	door status
⟨*number*⟩	A numeric value denoting a quantity	100
⟨*operational condition*⟩	A condition or event that occurs during system operation	the user tries to open the door
⟨*system*⟩	Any part of the system; sub-class of entity	door
⟨*unit*⟩	Unit of measurement	millisecond
⟨*user*⟩	A person somehow interacting with the system, e.g., operating it; sub-class of entity	pilot

4.2 Domain Ontology

The domain ontology contains facts about the domain that are relevant to requirements engineering, i.e., facts that can be used to formulate and to analyze requirements. The domain ontology should be usable to specify requirements for several projects in the same domain. Thus adding concepts which are only relevant to a single product should be avoided. See section 6 for a discussion about combining several ontologies. This approach could be used to split the ontology into a common domain part and a product-specific one.

There are three kinds of facts about the domain stored in the domain ontology. The following list describes them. The tool uses an OWL[2] representation to store ontologies. A detailed mapping to OWL can be found in Table 2.

Table 2. Mapping of domain facts to OWL

Fact	OWL Expressions
Concept(name, definition)	Declaration(Class(*concept-iri*)) AnnotationAssertion(rdfs:label *concept-iri name*) AnnotationAssertion(rdfs:comment *concept-iri definition*)
Relation(subj, label, obj)	Declaration(ObjectProperty(*label-iri*)) AnnotationAssertion(rdfs:label *label-iri label*) SubClassOf(*subj-iri* ObjectAllValuesFrom(*label-iri obj-iri*))
SubClass(sub, super)	SubClassOf(*sub-iri super-iri*)
Equivalence(concept$_1$, concept$_2$)	EquivalentClasses(*concept$_1$-iri concept$_2$-iri*)
Deprecated(concept)	AnnotationAssertion(*deprecated-iri concept-iri* `1`)

Concept: A concept represents an entity in the problem domain. The entity can be material (e.g., a physical component of the system) or immaterial (e.g., a temporal state). OWL classes are used to represent concepts. The reason for using classes instead of individuals is the built-in support for sub-classing.

A concept has two attributes, its name and a textual definition. The definition is intended to provide the possibility to check whether the correct term is used.

Relation: A relation is a labeled directed connection between two concepts. A relation contains a label which is expected to be a verb. The label, the relation's source and destination concepts form a subject-verb-object triple. Relations are used for guidance (section 4.3). Relations map to OWL object properties and object property restrictions.

Axiom: There are two types of axioms that are relevant to guidance: sub-class and equivalence axioms. The first one specifies that one concept is a sub-class of another concept, e.g., *cargo door* is a sub-class of *door*. This information is used to "inherit" suggestions to sub-class concepts, e.g., the guidance system infers the suggestion *the user opens the cargo door* from the base class' *the user opens the door*.

The equivalence axiom is used to express that two concepts having different names refer to the same entity in the domain. An example from DMS is the equivalence of *aircraft* and *airplane*. Ideally each real-world phenomenon has exactly one name. However, due to requirements coming from different stakeholders or due to legacy reasons, at times several names are required. It is possible to mark a concept as being *deprecated*; the tool will warn about occurrences of such concepts and will suggest using an equivalent non-deprecated concept instead.

In this work we assume the pre-existence of a suitable domain ontology. See [17] for ways of constructing new domain ontologies. The tool contains an ontology

editor that is tailored to the information described here. We found this editor to be more user-friendly than generic OWL editors like Protégé[3].

4.3 Guidance

When filling the attributes of a boilerplate, the tool provides a list of suggestions to the requirements engineer. The provided guidance depends on the attribute the requirements engineer is currently filling, e.g., the suggestions for ⟨*system*⟩ will be completely different than for ⟨*action*⟩. The idea is to apply an attribute-based pre-filter to avoid overwhelming the user with the complete list of ontology entities. Typing characters further filters this list of suggestions to only those entries matching the typed string.

It is not mandatory to choose from the list of suggestions; the tool will not stop the requirements engineer from entering something completely different. In case information is missing from the domain ontology, an update of the ontology should be performed to improve the guidance for similar requirements. All changes to the domain ontology should be validated by a domain expert to ensure data correctness.

There are three types of suggestions for an attribute; Table 3 provides an overview over the suggestion types.

Concept: The tool suggests to use the name of a concept for an attribute. The tool generates two variants, just the plain name and once prefixed with the article "the". The idea is that most of the times using "the" will be appropriate but sometimes other determiners like "all" or "each" are more suitable and are typed in manually.

Verb-Object: The tool uses a relation from the domain ontology to suggest a verb phrase to the requirements engineer. The suggestion is the concatenation of the verb's infinitive form[4], the word "the" and the relation's destination object. This construction is chosen in order to be grammatically correct following a modal verb like "shall". An example from Figure 2 is the suggestion *check the door status*.

Subject-Verb-Object: For this kind of suggestion the entire relation including subject and object is taken into account. The suggestion text is "the", the subject, the verb conjugated into third person singular form, "the" and the object. An example from Figure 2 is the suggestion *the person tries to open the door*.

It is possible to combine several suggestions simply by selecting the first one, manually typing in "and" and selecting another suggestion.

For the classification of concepts into different attributes a separate ontology, the *attributes ontology*, is used. The attributes ontology contains an OWL class

[3] `http://protege.stanford.edu/`

[4] For building verb infinitives the *morphological analyzer* of the GATE project (`http://gate.ac.uk/`) is used.

Table 3. Suggestion Types

Type	Suggestion
Concept	*concept*
	`the` *concept*
Verb-Object	*verb (inf.)* `the` *object*
Subject-Verb-Object	`the` *subject verb (3rd sing.)* `the` *object*

Fig. 2. Domain Ontology and Attributes Ontology

per attribute and the sub-class axioms mentioned in Table 1. The domain ontology imports the attributes ontology to use its classes. Domain concepts are linked to attributes by means of sub-class axioms which are stored in the domain ontology.

An example for the semantic guidance system is given in Figure 2. The domain ontology is shown in the upper part of the figure, the attributes ontology is below. The concept *Doors Management System* is a sub-class of class *system*, which in turn allows the tool to suggest using the *Doors Management System* for a boilerplate containing the attribute ⟨*system*⟩. The blue regions represent verb-object and subject-verb-object suggestions in the domain ontology. Their mapping to the attributes ⟨*action*⟩ and ⟨*operational condition*⟩ is inferred automatically by the tool.

Figure 3 shows the boilerplates for two requirements and some of the suggestions provided by the guidance system. The information that *Doors Management System* is a *system* and that *second* and *millisecond* are values for attribute *unit* is stored in the domain ontology itself. The suggestions *check the door status* and *determine the door unlockability* are inferred from the domain ontology relations. The knowledge to suggest verb-object pairs for the attribute *action* is a built-in feature of the tool. The attribute *operational condition* turns out to be the most difficult one in terms of providing useful suggestions. The reason for this is that

Fig. 3. Boilerplates and Suggestions

there are many grammatical ways to describe conditions, a subject-verb-object triple being only one of them. Therefore the tool does not only suggest those triples for conditions; instead all concepts, verb-object pairs and subject-verb-object triples are provided in order to use those phrases in conditions.

5 Evaluation

As mentioned before we evaluated the semantic guidance system with a domain ontology and a set of requirements from the Doors Management System.

5.1 Setting

The use case contains a set of 43 requirements specified using natural language text. Various types of requirements are included: functional, safety, performance, reliability, availability and cost. Each requirement was reformulated into a boilerplate requirement using DODT. The semantic guidance system was used to assist in filling the boilerplate attributes.

The domain ontology for DMS was specifically developed for usage with the boilerplates tool. The data for the DMS ontology was initially provided by EADS and was then completed by the authors. Table 4 lists the number of concepts, relations and axioms of the DMS ontology.

Figure 4 shows the graphical user interface of the tool. At the top of the interface boilerplates can be selected. The center shows the currently selected boilerplates and text boxes for the attribute values of the requirements. The list of phrases below the text boxes are the suggestions provided by the semantic guidance system. Typing in the text boxes filters the list of suggestions. The tool shows the textual definitions of the concepts as tooltips. Selecting a list entry will add the text to the corresponding text box. The bottom of the interface lists all requirements. Expressions that refer to entities from the domain ontology are

underlined with green lines; fixed syntax elements of boilerplates with black. If nouns missing from the domain ontology were to be seen, they would be highlighted with the color red.

Table 5 present statistics about the suggestions produced by the guidance system for the DMS ontology.

Table 4. Ontology Measurements

Entity	Count
Concepts	107
Relations	70
Axioms	123
SubClass	108
to Concepts	15
to Attributes	93
Equivalence	15

Table 5. Guidance Suggestions

Type	Count
Concept	212
Verb-Object	69
Subject-Verb-Object	101
Total	382

Table 6. Evaluation Results

Item	Count
Requirements	43
Boilerplates	21
Attributes	120
Complete suggestions	36
Partial suggestions	39

Fig. 4. DODT Screenshot

5.2 Results

Table 6 lists the major results of the evaluation. For 43 requirements, we used 21 different boilerplates. The boilerplate which was used most often (16 times) is ⟨*system*⟩ `shall` ⟨*action*⟩. The 43 boilerplate requirements have a total of 120 attributes. For 36 attributes out of 120 (30%) the semantic guidance system was able to suggest the entire attribute value without any need for a manual change. For another 59 attributes (57.5%) the guidance could suggest at least parts of the attribute value. This leaves 25 attribute values (12.5%) for that the guidance was no help. For partial matches, these are some of the reasons the attribute values had to be modified:

- A different determiner is used than the suggested "the", e.g., "a", "each" or "all".
- The plural is used instead of singular.
- A combination of two or more suggestions is used.
- A subordinate clause is added, e.g., "each door that could be a hazard if it unlatches".

Reasons for no guidance are these:

- Numbers for the ⟨*number*⟩ attribute cannot be suggested.
- Words are used that do not exist in the domain ontology.

Future work will include setting up an evaluation to compare the elicitation time with and without the semantic guidance system. However, due to the high percentage where the guidance was able to help (>85%) we are confident that efficiency improved, even though the presentation of explicit numbers has to be postponed to future work.

We also hope to improve the quality of requirements using the tool. We did a qualitative comparison of the original DMS requirements and the boilerplate requirements. These are our findings:

- Boilerplate requirements encourage using the active voice. In our evaluation the original requirement "Information concerning the door status shall be sent from the Doors Management System to ground station..." was turned into "The Doors Management System shall send information concerning the door status to ground station..." 8 requirements were improved in this way. In some cases missing subjects were added.
- Requirements like "There shall be..." and "It shall not be possible to..." were changed into "The *subject* shall have" and "The *subject* shall not allow...". Such changes make it obvious what part of the system is responsible to fulfill the requirement. To determine the right value for *subject* the original stakeholders should be asked for clarification. Due to timing constraints this was not possible and plausible values were inserted by the authors.
- During the requirements transformation we found that the original requirements used different expressions for seemingly identical things, e.g., "provision to prevent pressurization" and "pressure prevention means" or "airplane"

and "aircraft". Such synonyms are either stored as an equivalence axiom in the domain ontology or, preferably, stakeholders agree upon the usage of one term.
- Using boilerplates improved the overall consistency of the requirements. The original requirements set contains a mixture of "must" and "shall". While using one or the other is probably a matter of taste, one form should be picked and used consistently.
- The guidance system corrected a few typographic errors in the original requirements, e.g., "miliseconds".

We found the tool to be easy to use and user-friendly. This sentiment is shared by the partners in the CESAR project who are also currently evaluating the tool.

6 Conclusion and Future Work

Requirements should be made as consistent, correct and complete as possible to prevent detecting and correcting errors in later design phases. With respect to the three points raised in the introduction about requirements engineering, this work intends to be the foundation for further analyses, e.g., by facilitating requirements categorization with ontology knowledge.

We presented a tool for the elicitation of boilerplate requirements that includes a semantic guidance system which suggests concept names and phrases that were built from relations and axioms of the domain ontology. The tool managed to provide useful suggestions in the majority of the cases (>85%). We realize that the tool needs to be evaluated in a larger context, i.e., more requirements and a larger ontology. We will address this in our future research work.

The selection of suitable boilerplates for a requirement is not always trivial and requires a bit of experience with the boilerplates method. Thus a feature we want to explore and possibly add to the tool is the semi-automatic conversion of natural language requirements into boilerplate requirements.

While creating the DMS ontology, we found there are concepts for which the suggestions provided by the semantic guidance system really help but which are not specific to the problem domain. The most prominent example are measurement units like "second" or "kg". So what we want to do is to collect such entities into a separate ontology and to extend the tool to be able manage several ontologies. This could be handled by adding OWL import declarations to the "main" domain ontology. Such domain-independent ontologies can then be easily reused for guidance in other domains.

Acknowledgments

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement N° 100016 and from specific national programs and/or funding authorities. This work has been supported by the Christian Doppler Forschungsgesellschaft and the BMWFJ, Austria.

References

1. IEEE Recommended Practice for Software Requirements Specifications. IEEE Std 830 (1998)
2. OWL 2 Web Ontology Language Direct Semantics. Tech. rep., W3C (2009), http://www.w3.org/TR/2009/REC-owl2-direct-semantics-20091027/
3. Cobleigh, R., Avrunin, G., Clarke, L.: User Guidance for Creating Precise and Accessible Property Specifications. In: 14th International Symposium on Foundations of Software Engineering, pp. 208–218. ACM, New York (2006)
4. Denger, C., Berry, D., Kamsties, E.: Higher Quality Requirements Specifications through Natural Language Patterns. In: 2003 IEEE International Conference on Software - Science, Technology and Engineering, pp. 80–90. IEEE, Los Alamitos (2003)
5. Egyed, A., Grunbacher, P.: Identifying Requirements Conflicts and Cooperation: How Quality Attributes and Automated Traceability Can Help. IEEE Software 21(6), 50–58 (2004)
6. Elazhary, H.H.: REAS: An Interactive Semi-Automated System for Software Requirements Elicitation Assistance. IJEST 2(5), 957–961 (2010)
7. Gotel, O., Finkelstein, C.: An Analysis of the Requirements Traceability Problem. In: 1st International Conference on Requirements Engineering, pp. 94–101 (1994)
8. Gottesdiener, E.: Requirements by Collaboration: Workshops for Defining Needs. Addison-Wesley, Reading (2002)
9. Hull, E., Jackson, K., Dick, J.: Requirements Engineering. Springer, Heidelberg (2005)
10. Ibrahim, N., Kadir, W., Deris, S.: Propagating Requirement Change into Software High Level Designs towards Resilient Software Evolution. In: 16th Asia-Pacific Software Engineering Conference, pp. 347–354. IEEE, Los Alamitos (2009)
11. Jackson, J.: A Keyphrase Based Traceability Scheme. IEEE Colloquium on Tools and Techniques for Maintaining Traceability During Design, 2/1–2/4 (1991)
12. Kaindl, H.: The Missing Link in Requirements Engineering. Software Engineering Notes 18, 30–39 (1993)
13. Kaiya, H., Saeki, M.: Ontology Based Requirements Analysis: Lightweight Semantic Processing Approach. In: 5th Int. Conf. on Quality Software, pp. 223–230 (2005)
14. Kitamura, M., Hasegawa, R., Kaiya, H., Saeki, M.: A Supporting Tool for Requirements Elicitation Using a Domain Ontology. Software and Data Technologies, 128–140 (2009)
15. Kotonya, G., Sommerville, I.: Requirements Engineering. John Wiley & Sons, Chichester (1998)
16. Matsuo, Y., Ogasawara, K., Ohnishi, A.: Automatic Transformation of Organization of Software Requirements Specifications. In: 4th International Conference on Research Challenges in Information Science, pp. 269–278. IEEE, Los Alamitos (2010)
17. Omoronyia, I., Sindre, G., Stålhane, T., Biffl, S., Moser, T., Sunindyo, W.: A Domain Ontology Building Process for Guiding Requirements Elicitation. In: 16th REFSQ, pp. 188–202 (2010)
18. Pedrinaci, C., Domingue, J., Alves de Medeiros, A.K.: A Core Ontology for Business Process Analysis. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 49–64. Springer, Heidelberg (2008)
19. Rupp, C.: Requirements-Engineering und -Management. Hanser (2002)
20. Ståhane, T., Omoronyia, I., Reichenbach, F.: Ontology-Guided Requirements and Safety Analysis. In: 6th International Conference on Safety of Industrial Automated Systems (2010)
21. Watkins, R., Neal, M.: Why and How of Requirements Tracing. IEEE Software 11(4), 104–106 (1994)

Bidirectional Requirements Traceability

By Linda Westfall

www.westfallteam.com

Traceability is one of the essential activities of good requirements management. Traceability is used to ensure that the right products are being built at each phase of the software development life cycle, to trace the progress of that development and to reduce the effort required to determine the impacts of requested changes. This article explores:

- What is traceability?
- Why is traceability a good practice?
- How is traceability performed?

What is Traceability?

The IEEE Standard Glossary of Software Engineering Terminology defines traceability as "the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another." [IEEE-610]

Traceability is used to track the relationship between each unique product-level requirement and its source. For example, a product requirement might trace from a business need, a user request, a business rule, an external interface specification, an industry standard or regulation, or to some other source.

Traceability is also used to track the relationship between each unique product-level requirement and the work products to which that requirement is allocated. For example, a single product requirement might trace to one or more architectural elements, detail design elements, objects/classes, code units, tests, user documentation topics, and/or even to people or manual processes that implements that requirement.

Good traceability practices allow for bidirectional traceability, meaning that the traceability chains can be traced in both the forwards and backwards directions as illustrated in Figure 1.

Forward traceability looks at:

- Tracing the requirements sources to their resulting product requirement(s) to ensure the completeness of the product requirement specification.
- Tracing each unique product requirement forward into the design that implements that requirement, the code that implements that design and the tests that validate that requirement and so on. The objective is to ensure that each requirement is implemented in the product and that each requirement is thoroughly tested.

Figure 1: Bidirectional (Forward & Backward) Traceability

Backwards traceability looks at:

Copyright © 2006 The Westfall Team. All Rights Reserved.

- Tracing each unique work product (e.g., design element, object/class, code unit, test) back to its associated requirement. Backward traceability can verify that the requirements have been kept current with the design, code, and tests.
- Tracing each requirement back to its source(s).

Why is Traceability a Good Practice?

The Software Engineering Institute (SEI) Capability Maturity Model Integration® (CMMI®) states that the purpose of the process area in "Requirements Management is to manage the requirements of the project's product and product components and to identify inconsistencies between those requirements and the project's plans and work products." [SEI-00]. One of the specific practices under the Requirements Management process area is to "Maintain Bidirectional Traceability of Requirements." [SEI-00] What is the benefit of putting in the effort to maintain bidirectional traceability?

Forward traceability ensures proper direction of the evolving product (that we are building the right product) and indicates the completeness of the subsequent implementation. For example, if a business rule can't be traced forward to one or more product requirements then the product requirements specification is incomplete and the resulting product may not meet the needs of the business. If a product requirement cannot be traced forward to its associated architectural design elements, then the architectural design is not complete and so on.

If, on the other hand, there are changes in the business environment (e.g., a business rule change or a standard change), then if good forward traceability has been maintained, that change can be traced forward to the associated requirements and all of the work products that are impacted by that change. This greatly reduces the amount of effort required to do a thorough job of impact analysis. It also reduces the risk that one of the effected work products is forgotten, resulting in an incomplete implementation of the change (i.e., a defect).

Backwards traceability helps ensure that the evolving product remains on the correct track with regards to the original and/or evolving requirements (that we are building the product right). The objective is to ensure that we are not expanding the scope of the project by adding design elements, code, tests or other work products that are not called out in the requirements (i.e., "gold plating"). If there is a change needed in the implementation or if the developers come up with a creative, new technical solution, that change or solution should be traced backwards to the requirements and the business needs to ensure that it is within the scope of the desired product. For example, if there is a work product element that doesn't trace backwards to the product requirements one of two things is true. The first possibility is that there is a missing requirement because the work product element really is needed. In this case, traceability has helped identify the missing requirement and can also be used to evaluate the impacts of adding that requirement to project plans and other work products (forward traceability again). The second possibility is that there is "gold plating" going on – something has been added that should not be part of the product. Gold plating is a high risk activity because project plans have not allocated time or resources to the work and the existence of that part of the product may not be well communicated to other project personnel (e.g., tester doesn't test it, it's not included in user documentation).

Another benefit of backward traceability comes when a defect is identified in one of the work products. For example, if a piece of code has a defect, the traceability matrix can be used to help determine the root cause of that defect. For example, is it just a code defect or does it trace back to a defect in the design or requirements? If it's a design or requirements defect, what other work products might be impacted by the defect?

Benefits of bi-directional requirements traceability include the ability to:

- Analyze the impact of a change
  - All work products affected by a changed requirement
  - All requirements affected by a change or defect in a work product
- Assess current status of the requirements and the project
  - Identify missing requirements

Copyright © 2006 The Westfall Team. All Rights Reserved.

– Identify gold plating

How is Traceability Performed?

The classic way to perform traceability is by constructing a traceability matrix. As illustrated in Table 1, a traceability matrix summarizes in matrix form the traceability from original identified stakeholder needs to their associated product requirements and then on to other work product elements. In order to construct a traceability matrix, each requirement, each requirements source and each work product element must have a unique identifier that can be used as a reference in the matrix. The requirement matrix has the advantage of being a single repository for documenting both forwards and backwards traceability across all of the work products.

Requirement Source	Product Requirements	HLD Section #	LLD Section #	Code Unit	UTS Case #	STS Case #	User Manual
Business Rule #1	R00120 Credit Card Types	4.1 Parse Mag Strip	4.1.1 Read Card Type	Read_Card _Type.c Read_Card _Type.h	UT 4.1.032 UT 4.1.033 UT 4.1.038 UT 4.1.043	ST 120.020 ST 120.021 ST 120.022	Section 12
			4.1.2 Verify Card Type	Ver_Card _Type.c Ver_Card _Type.h Ver_Card _Types. dat	UT 4.2.012 UT 4.2.013 UT 4.2.016 UT 4.2.031 UT 4.2.045	ST 120.035 ST 120.036 ST 120.037 ST 120.037	Section 12
Use Case #132 step 6	R00230 Read Gas Flow	7.2.2 Gas Flow Meter Interface	7.2.2 Read Gas Flow Indicator	Read_Gas _Flow.c	UT 7.2.043 UT 7.2.044	ST 230.002 ST 230.003	Section 21.1.2
	R00231 Calculate Gas Price	7.3 Calculate Gas price	7.3 Calculate Gas price	Cal_Gas_ Price.c	UT 7.3.005 UT 7.3.006 UT 7.3.007	ST 231.001 ST 231.002 ST 231.003	Section 21.1.3

Table 1: Example of a Traceability Matrix

Modern requirements management tools include traceability mechanisms as part of their functionality.

A third mechanism for implementing traceability is trace tagging. Again, each requirement, each requirement source and each work product element must have a unique identifier. In trace tagging however, those identifiers are used as tags in the subsequent work products to identify backwards tracing to the predecessor document. As illustrated in Figure 2 for example, the Software Design Specification

Figure 2: Example of Trace Tagging

Copyright © 2006 The Westfall Team. All Rights Reserved.

(SDS) includes tags that identify the requirements implemented by each uniquely identified design element and the Unit Test Specification (UTS) includes trace tags that trace back to the design elements that each test case verifies. This tagging propagates through the work product set with source code units that include trace tags back to design elements, integration test cases with tags back to architecture elements and system test cases with trace tags back to requirements as appropriate depending on the hierarchy of work products used on the product. Trace tags have the advantage of being part of the work products so a separate matrix isn't maintained. However, while backwards tracing is easy with trace tags, forward tracing is very difficult using this mechanism.

All traceability implementation techniques require the commitment of a cross-functional team of participants to create and maintain the linkages between the requirements, their source and their allocation to subsequent work products. The requirements analyst must initiate requirements traceability and document the original tracing of the product requirements to their source. As system and software architects create the high-level design, those practitioners add their information to the traceability documentation. Developers doing low-level design, code and unit testing add additional traceability information for the elements they create, as do the integration, system and alpha, beta and acceptance testers. For small projects, some of these roles may not exist or may be done by the sample practitioner, which limits the number of different people working with the traceability information. For larger projects, where traceability information comes from many different practitioners, it may be necessary to have someone who coordinates, documents and ensures periodic audits of the traceability information from all its various sources to achieve completeness and consistency.

References

IEEE-610 IEEE Standards Software Engineering, *IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610-1990*, The Institute of Electrical and Electronics Engineers, 1999, ISBN 0-7381-1559-2.

SEI-00 *CMMI^{SM} for Systems Engineering/Software Engineering, Version 1.02 (CMMI-SW/SW, V 1.02); CMMI Staged Representation, CMU/SEI-2000-TR-018, ESC-TR-2000-018; Continuous Representation, CMU/SEI-2000-TR-019, ESC-TR-2000-019;* Product Development Team; Software Engineering Institute; November 2000.

Copyright © 2006 The Westfall Team. All Rights Reserved.

SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2012; **22**:67–120
Published online 11 March 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.430

Regression testing minimization, selection and prioritization: a survey

S. Yoo*,† and M. Harman

King's College London, Centre for Research on Evolution, Search and Testing, Strand, London WC2R 2LS, U.K.

SUMMARY

Regression testing is a testing activity that is performed to provide confidence that changes do not harm the existing behaviour of the software. Test suites tend to grow in size as software evolves, often making it too costly to execute entire test suites. A number of different approaches have been studied to maximize the value of the accrued test suite: minimization, selection and prioritization. Test suite minimization seeks to eliminate redundant test cases in order to reduce the number of tests to run. Test case selection seeks to identify the test cases that are relevant to some set of recent changes. Test case prioritization seeks to order test cases in such a way that early fault detection is maximized. This paper surveys each area of minimization, selection and prioritization technique and discusses open problems and potential directions for future research. Copyright © 2010 John Wiley & Sons, Ltd.

Received 4 September 2008; Revised 4 January 2010; Accepted 6 January 2010

KEY WORDS: regression testing; test suite minimization; regression test selection; test case prioritization

1. INTRODUCTION

Regression testing is performed when changes are made to existing software; the purpose of regression testing is to provide confidence that the newly introduced changes do not obstruct the behaviours of the existing, unchanged part of the software. It iza complex procedure that is all the more challenging because of some of the recent trends in software development paradigms. For example, the component-based software development method tends to result in the use of many black-box components, often adopted from a third party. Any change in the third-party components may interfere with the rest of the software system, yet it is hard to perform regression testing because the internals of the third-party components are not known to their users. The shorter life-cycle of software development, such as the one suggested by the *agile* programming discipline, also imposes restrictions and constraints on how regression testing can be performed within limited resources.

Naturally, the most straightforward approach to this problem is to simply execute all the existing test cases in the test suite; this is called a *retest-all* approach. However, as software evolves, the test suite tends to grow, which means that it may be prohibitively expensive to execute the

*Correspondence to: S. Yoo, King's College London, Centre for Research on Evolution, Search and Testing, Strand, London WC2R 2LS, U.K.
†E-mail: Shin.Yoo@kcl.ac.uk

Contract/grant sponsor: EPSRC; contract/grant numbers: EP/D050863, GR/S93684, GR/T22872
Contract/grant sponsor: EU; contract/grant number: IST-33472
Contract/grant sponsor: DaimlerChrysler Berlin and Vizuri Ltd., London

Copyright © 2010 John Wiley & Sons, Ltd.

entire test suite. This limitation forces consideration of techniques that seek to reduce the effort required for regression testing in various ways.

A number of different approaches have been studied to aid the regression testing process. The three major branches include test suite minimization, test case selection and test case prioritization. Test suite minimization is a process that seeks to identify and then eliminate the obsolete or redundant test cases from the test suite. Test case selection deals with the problem of selecting a subset of test cases that will be used to test the *changed* parts of the software. Finally, test case prioritization concerns the identification of the 'ideal' ordering of test cases that maximize desirable properties, such as early fault detection. Existing empirical studies show that the application of these techniques can be cost-effective.

This paper surveys work undertaken in these three related branches of regression testing. Section 2 introduces the nomenclature. Section 3 describes different test suite minimization techniques as well as their efficiency and effectiveness. Section 4 examines test case selection techniques according to the specific analysis technique used, and evaluates the strengths and weaknesses of each approach. Section 5 introduces test case prioritization techniques. Section 6 introduces meta-empirical studies concerning evaluation methodologies and cost-effectiveness analysis of regression testing techniques. Section 7 presents a summary of the field and identifies trends and issues. Section 8 introduces some gaps in the existing literature, thereby suggesting potential directions for future work. Section 9 concludes.

1.1. Motivation

When writing a survey paper there are two natural questions that need to be asked:

1. Why is this the right set of topics for a survey?
2. Is there already a recent survey in this area?

The first question concerns the motivation for the scope chosen for the survey, whereas the second concerns the perceived need for such a survey, once a suitable scope is established. For this paper the scope has been chosen to include topics on test suite minimization, Regression Test Selection (RTS) and test case prioritization. The reason for the choice of this scope is that these three topics are related by a common thread of optimization; each is an approach that optimizes the application of an existing pool of test cases.

The fact that all three approaches assume that the existence of a pool of test cases distinguishes these topics from test case generation, which seeks to create pools of test data. The three topics form a coherent set of approaches, each of which shares a common starting point; that the tester has a pool of test cases that is simply too large to allow all cases to be applied to the System Under Test (SUT). Each of the three approaches denotes a different way of coping with this problem of scale.

The relationship between the three techniques goes deeper than the mere shared application to pools of test data. It is not only the sets of problems addressed by each that exhibit overlap, but also the solution approaches that are applied. There is an intimate relationship between solutions to the three related problems, as this survey reveals. For instance, one way of selecting (or minimizing) a set of n test cases from a test pool would be to prioritize the whole set and pick the first n in priority order. Of course, there may be more optimal choices, since prioritization has to contend with any possible choice of n, whereas selection merely requires that an optimal choice is found for a given value of n.

Turning to the second question, the previous papers closest to this survey are a survey of RTS techniques undertaken in 1996 [1] and a recent systematic review on RTS [2] that concerns a specific set of research questions rather than a survey of the whole area. No previous survey on regression testing considered minimization, selection and prioritization collectively. The present survey claims that these classes of techniques are closely related to each other and denote a coherent sub-area of study. Our survey also includes recent applications of techniques that were surveyed in the earlier work of Rothermel and Harrold [1]. There are existing comparative studies on RTS [3–7], but these papers only concern a small number of specific RTS techniques and do not provide an overview of the field.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

1.2. Selection of papers

This survey aims to collect and consider papers that deal with three regression testing techniques: test suite minimization, RTS and test case prioritization. Our intention is not to undertake a systematic review, but rather to provide a broad state-of-the-art view on these related fields. Many different approaches have been proposed to aid regression testing, which has resulted in a body of literature that is spread over a wide variety of domains and publication venues. The majority of surveyed literature has been published in the software engineering domain, and especially in the software testing and software maintenance literature. However, the regression testing literature also overlaps with those of programming language analysis, empirical software engineering and software metrics.

Therefore, the paper selection criteria on which this survey is based are the problems considered in papers, while focusing on the specific topics of minimization, selection and prioritization. The formal definitions of these problems are presented in Section 2.2. The selected papers are listed in the Appendix. Fast abstracts and short papers have been excluded.

2. BACKGROUND

This section introduces the basic concepts and definitions that form a nomenclature of regression testing and minimization, selection and prioritization techniques.

2.1. Regression testing

Regression testing is performed between two different versions of software in order to provide confidence that the newly introduced features of the SUT do not interfere with the existing features. While the exact details of the modifications made to SUT will often be available, they may not be easily available in some cases. For example, when the new version is written in a different programming language or when the source code is unavailable, modification data will also be unavailable.

The following notations are used to describe concepts in the context of regression testing. Let P be the current version of the program under test, and P' be the next version of P. Let S be the current set of specifications for P, and S' be the set of specifications for P'. T is the existing test suite. Individual test cases will be denoted by lower case: t. $P(t)$ stands for the execution of P using t as input.

2.2. Distinction between classes of techniques

It is necessary at this point to establish a clear terminological distinction between the different classes of techniques described in the paper. Test suite minimization techniques seek to reduce the size of a test suite by eliminating redundant test cases from the test suite. Minimization is sometimes also called 'test suite reduction', meaning that the elimination is permanent. However, these two concepts are essentially interchangeable because all reduction techniques can be used to produce a temporary subset of the test suite, whereas any minimization techniques can be used to permanently eliminate test cases. More formally, following Rothermel *et al.* [8], the test suite minimization is defined as follows:

Definition 1 (*Test Suite Minimization Problem*)
Given: A test suite, T, a set of test requirements $\{r_1, \ldots, r_n\}$, that must be satisfied to provide the desired 'adequate' testing of the program, and subsets of T, $T_1, \ldots, T_n$, one associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to achieve requirement r_i.
Problem: Find a representative set, T', of test cases from T that satisfies all r_is.

The testing criterion is satisfied when every test requirement in $\{r_1, \ldots, r_n\}$ is satisfied. A test requirement, r_i, is satisfied by any test case, t_j, that belongs to the T_i, a subset of T. Therefore, the representative set of test cases is the hitting set of the T_is. Furthermore, in order to maximize the effect of minimization, T' should be the minimal hitting set of the T_is. The minimal hitting set problem is an NP-complete problem as is the dual problem of the minimal set cover problem [9].

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

While test case selection techniques also seek to reduce the size of a test suite, the majority of selection techniques are *modification-aware*. That is, the selection is not only temporary (i.e. specific to the current version of the program), but also focused on the identification of the modified parts of the program. Test cases are selected because they are relevant to the changed parts of the SUT, which typically involves a white-box static analysis of the program code. Throughout this survey, the meaning of 'test case selection problem' is restricted to this modification-aware problem. It is also often referred to as the RTS problem. More formally, following Rothermel and Harrold [1], the selection problem is defined as follows (refer to Section 4 for more details on how the subset T' is selected):

Definition 2 (*Test Case Selection Problem*)
Given: The program, P, the modified version of P, P' and a test suite, T.
Problem: Find a subset of T, T', with which to test P'.

Finally, test case prioritization concerns ordering test cases for early maximization of some desirable properties, such as the rate of fault detection. It seeks to find the optimal permutation of the sequence of test cases. It does not involve selection of test cases, and assumes that all the test cases may be executed in the order of the permutation it produces, but that testing may be terminated at some arbitrary point during the testing process. More formally, the prioritization problem is defined as follows:

Definition 3 (*Test Case Prioritization Problem*)
Given: A test suite, T, the set of permutations of T, PT, and a function from PT to real numbers, $f : PT \rightarrow \mathbb{R}$.
Problem: To find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

This survey focuses on papers that consider one of these three problems. Throughout the paper, these three techniques will be collectively referred to as 'regression testing techniques'.

2.3. Classification of test cases

Leung and White present the first systematic approach to regression testing by classifying types of regression testing and test cases [10]. Regression testing can be categorized into *progressive* regression testing and *corrective* regression testing. Progressive regression testing involves changes of specifications in P', meaning that P' should be tested against S'. On the other hand, corrective regression testing does not involve changes in specifications, but only in design decisions and actual instructions. It means that the existing test cases can be reused without changing their input/output relation.

Leung and White categorize test cases into five classes. The first three classes consist of test cases that already exist in T.

- *Reusable*: Reusable test cases only execute the parts of the program that remain unchanged between two versions, i.e. the parts of the program that are common to P and P'. It is unnecessary to execute these test cases in order to test P'; however, they are called *reusable* because they may still be retained and reused for the regression testing of the future versions of P.
- *Retestable*: Retestable test cases execute the parts of P that have been changed in P'. Thus, retestable test cases should be re-executed in order to test P'.
- *Obsolete*: Test cases can be rendered obsolete because (1) their input/output relation is no longer correct due to changes in specifications, (2) they no longer test what they were designed to test due to modifications to the program or (3) they are 'structural' test cases that no longer contribute to structural coverage of the program.

The remaining two classes consist of test cases that have yet to be generated for the regression testing of P'.

- *New-structural*: New-structural test cases test the modified program constructs, providing structural coverage of the modified parts in P'.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

- *New-specification*: New-specification test cases test the modified program specifications, testing the new code generated from the modified parts of the specifications of P'.

Leung and White go on to propose a retest strategy, in which a *test plan* is constructed based on the identification of changes in the program and classification of test cases. Although the definition of a test plan remains informal, it provides a basis for the subsequent literature. It is of particular importance to regression test case selection techniques, since these techniques essentially concern the problem of identifying *retestable* test cases. Similarly, test suite minimization techniques concern the identification of *obsolete* test cases. Test case prioritization also can be thought of as a more sophisticated approach to the construction of a test plan.

It should be noted that the subsequent literature focusing on the idea of selecting and reusing test cases for regression testing is largely concerned with corrective regression testing only. For progressive regression testing, it is very likely that new test cases are required in order to test the new specifications. So far, this aspect of the overall regression testing picture has been a question mainly reserved for test data generation techniques. However, the early literature envisages a 'complete' regression testing strategy that should also utilize test data generation techniques.

3. TEST SUITE MINIMIZATION

Test suite minimization techniques aim to identify redundant test cases and to remove them from the test suite in order to reduce the size of the test suite. The minimization problem described by Definition 1 can be considered as the minimal hitting set problem.

Note that the minimal hitting set formulation of the test suite minimization problem depends on the assumption that each r_i can be satisfied by a single test case. In practice, this may not be true. For example, suppose that the test requirement is functional rather than structural and, therefore, requires more than one test case to be satisfied. The minimal hitting set formulation no longer applies. In order to apply the given formulation of the problem, the functional granularity of test cases needs to be adjusted accordingly. The adjustment process may be either that a higher level of abstraction would be required so that each test case requirement can be met with a single test scenario composed of relevant test cases, or that a 'large' functional requirement needs to be divided into smaller sub-requirements that will correspond to individual test cases.

3.1. Heuristics

The NP-completeness of the test suite minimization problem encourages the application of heuristics; previous work on test case minimization can be regarded as the development of different heuristics for the minimal hitting set problem [11–14].

Horgan and London applied linear programming to the test case minimization problem in their implementation of a data-flow-based testing tool, ATAC [13, 15]. Harrold *et al.* presented a heuristic for the minimal hitting set problem with the worst-case execution time of $O(|T| * \max(|T_i|))$ [12]. Here $|T|$ represents the size of the original test suite, and $\max(|T_i|)$ represents the cardinality of the largest group of test cases among $T_1, \ldots, T_n$.

Chen and Lau applied GE and GRE heuristics and compared the results to that of the Harrold–Gupta–Soffa (HGS) heuristic [11]. The GE and GRE heuristics can be thought of as variations of the greedy algorithm that is known to be an effective heuristic for the set cover problem [16]. Chen *et al.* defined *essential* test cases as the opposite of *redundant* test cases. If a test requirement r_i can be satisfied by one and only one test case, the test case is an essential test case. On the other hand, if a test case satisfies only a subset of the test requirements satisfied by another test case, it is a redundant test case. Based on these concepts, the GE and GRE heuristics can be summarized as follows:

- *GE heuristic*: first select all essential test cases in the test suite; for the remaining test requirements, use the additional greedy algorithm, i.e. select the test case that satisfies the maximum number of unsatisfied test requirements.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table I. Example test suite taken from Tallam and Gupta [18]. The early selection made by the greedy approach, t_1, is rendered redundant by subsequent selections, $\{t_2, t_3, t_4\}$.

	Testing requirements					
Test case	r_1	r_2	r_3	r_4	r_5	r_6
t_1	x	x	x			
t_2	x			x		
t_3		x			x	
t_4			x			x
t_5					x	

- *GRE heuristic*: first remove all redundant test cases in the test suite, which may make some test cases essential; then perform the GE heuristic on the reduced test suite.

Their empirical comparison suggested that no single technique is better than the other. This is a natural finding, because the techniques concerned are heuristics rather than precise algorithms.

Offutt *et al.* also treated the test suite minimization problem as the dual of the minimal hitting set problem, i.e. the set cover problem [14]. Their heuristics can also be thought of as variations of the greedy approach to the set cover problem. However, they adopted several different test case orderings, instead of the fixed ordering in the greedy approach. Their empirical study applied their techniques to a mutation-score-based test case minimization, which reduced sizes of test suites by over 30%.

Whereas other minimization approaches primarily considered code-level structural coverage, Marré and Bertolino formulated test suite minimization as a problem of finding a spanning set over a graph [17]. They represented the structure of the SUT using a *decision-to-decision* graph (ddgraph). A ddgraph is a more compact form of the normal CFG since it omits any node that has one entering edge and one exiting edge, making it an ideal representation of the SUT for branch coverage. They also mapped the result of data-flow analysis onto the ddgraph for testing requirements such as *def–use* coverage. Once testing requirements are mapped to entities in the ddgraph, the test suite minimization problem can be reduced to the problem of finding the minimal spanning set.

Tallam and Gupta developed the greedy approach further by introducing the *delayed greedy* approach, which is based on the Formal Concept Analysis of the relation between test cases and testing requirements [18]. A potential weakness of the greedy approach is that the early selection made by the greedy algorithm can eventually be rendered redundant by the test cases subsequently selected. For example, consider the test suite and testing requirements depicted in Table I, taken from Tallam and Gupta [18]. The greedy approach will select t_1 first as it satisfies the maximum number of testing requirements, and then continues to select t_2, t_3 and t_4. However, after the selection of t_2, t_3 and t_4, t_1 is rendered redundant. Tallam *et al.* tried to overcome this weakness by constructing a concept lattice, a hierarchical clustering based on the relation between test cases and testing requirements. Tallam *et al.* performed two types of reduction on the concept lattice. First, if a set of requirements covered by t_i is a superset of the set of requirements covered by t_j, then t_j is removed from the test suite. Second, if a set of test cases that cover requirement r_i is a subset of the set of test cases that cover requirement r_j, requirement r_i is removed. The concept lattice is a natural representation that supports the identification of these test cases. Finally, the greedy algorithm is applied to the transformed set of test cases and testing requirements. In the empirical evaluation, the test suites minimized by this 'delayed greedy' approach were either the same size or smaller than those minimized by the classical greedy approach or by the HGS heuristic.

Jeffrey and Gupta extended the HGS heuristic so that certain test cases are selectively retained [19, 20]. This 'selective redundancy' is obtained by introducing a secondary set of testing requirements. When a test case is marked as redundant with respect to the first set of testing requirements, Jeffrey and Gupta considered whether the test case is also redundant with respect to the second set of testing requirements. If it is not, the test case is still selected, resulting in a certain level of redundancy with respect to the first set of testing requirements. The empirical evaluation used

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

branch coverage as the first set of testing requirements and *all-uses* coverage information obtained by data-flow analysis. The results were compared to two versions of the HGS heuristic, based on branch coverage and *def–use* coverage. The results showed that, while their technique produced larger test suites, the fault-detection capability was better preserved compared to single-criterion versions of the HGS heuristic.

Whereas the selective redundancy approach considers the secondary criterion only when a test case is marked as being redundant by the first criterion, Black *et al.* considered a bi-criteria approach that takes into account both testing criteria [21]. They combined the *def–use* coverage criterion with the past fault-detection history of each test case using a weighted-sum approach and used Integer Linear Programming (ILP) optimization to find subsets. The weighted-sum approach uses weighting factors to combine multiple objectives. For example, given a weighting factor α and two objectives o_1 and o_2, the new and combined objective, o', is defined as follows:

$$o' = \alpha o_1 + (1-\alpha) o_2$$

Consideration of a secondary objective using the weighted-sum approach has been used in other minimization approaches [22] and prioritization approaches [23]. Hsu and Orso also considered the use of an ILP solver with multi-criteria test suite minimization [22]. They extended the work of Black *et al.* by comparing different heuristics for a multi-criteria ILP formulation: the weighted-sum approach, the prioritized optimization and a hybrid approach. In prioritized optimization, the human user assigns a priority to each of the given criteria. After optimizing for the first criterion, the result is added as a constraint, while optimizing for the second criterion, and so on. However, one possible weakness shared by these approaches is that they require additional input from the user of the technique in the forms of weighting coefficients or priority assignment, which might be biased, unavailable or costly to provide.

Contrary to these approaches, Yoo and Harman treated the problem of time-aware prioritization as a multi-objective optimization problem [24]. Instead of using a fitness function that combines selection and prioritization, they used a Pareto-efficient multi-objective evolutionary algorithm to simultaneously optimize for multiple objectives. You and Harman argued that the resulting Pareto-frontier not only provides solutions but also allows the tester to observe trade-offs between objectives, providing additional insights.

McMaster and Memon proposed a test suite minimization technique based on *call-stack coverage*. A test suite is represented by a set of unique maximum depth call stacks; its minimized test suite is a subset of the original test suite whose execution generates the same set of unique maximum depth call stacks. Note that their approach is different from simply using function coverage for test suite minimization. Consider two test cases, t_1 and t_2, respectively, producing call stacks $c_1 = \langle f_1, f_2, f_3 \rangle$ and $c_2 = \langle f_1, f_2 \rangle$. With respect to function coverage, t_2 is rendered redundant by t_1. However, McMaster and Memon regard c_2 to be unique from c_1. For example, it may be that t_2 detects a failure that prevents the invocation of function f_3. Once the call-stack coverage information is collected, the HGS heuristic can be applied. McMaster and Memon later applied the same approach to Graphical User Interface (GUI) testing [25]. It was also implemented for object-oriented systems by Smith *et al.* [26].

While most test suite minimization techniques are based on some kind of coverage criteria, there do exist interesting exceptions. Harder *et al.* approached test suite minimization using operational abstraction [27]. An operational abstraction is a formal mathematical description of program behaviour. While it is identical to formal specifications in form, an operational abstraction expresses dynamically observed behaviour. Harder *et al.* use the widely studied Daikon dynamic invariant detector [28] to obtain operational abstractions. Daikon requires executions of test cases for the detection of possible program invariants. Test suite minimization is proposed as follows: if the removal of a test case does not change the detected program invariant, it is rendered redundant. They compared the operational abstraction approach to branch coverage-based minimization. While their approach resulted in larger test suites, it also maintained higher fault-detection capability. Moreover, Harder *et al.* also showed that test suites minimized for coverage adequacy can often be improved by considering operational abstraction as an additional minimization criterion.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Leitner *et al.* propose a somewhat different version of the minimization problem [29]. They start from the assumption that they already have a *failing test case*, which is too complex and too long for the human tester to understand. Note that this is often the case with randomly generated test data; the test case is often simply too complex for the human to establish the cause of failure. The goal of minimization is to produce a shorter version of the test case; the testing requirement is that the shorter test case should still reproduce the failure. This minimization problem is interesting because there is no uncertainty about the fault-detection capability; it is given as a testing requirement. Leitner *et al.* applied the widely studied *delta-debugging* technique [30] to reduce the size of the failing test case.

Schroeder and Korel proposed an approach of test suite minimization for black-box software testing [31]. They noted that the traditional approach of testing black-box software with combinatorial test suites may result in redundancy, since certain inputs to the software may not affect the outcome of the output being tested. They first identified, for each output variable, the set of input variables that can affect the outcome. Then, for each output variable, an individual combinatorial test suite is generated with respect to only those input variables that may affect the outcome. The overall test suite is a union of all combinatorial test suites for individual output variables. This has a strong connection to the concept of Interaction Testing, which is discussed in detail in Section 5.2.

Other work has focused on model-based test suite minimization [32, 33]. Vaysburg *et al.* introduced a minimization technique for model-based test suites that uses dependence analysis of Extended Finite State Machines (EFSMs) [32]. Each test case for the model is a sequence of transitions. Through dependence analysis of the transition being tested, it is possible to identify the transitions that affect the tested transition. In other words, testing a transition T can be thought of as testing a set of other transitions that affect T. If a particular test case tests the same set of transitions as some other test case, then it is considered to be redundant. Korel *et al.* extended this approach by combining the technique with automatic identification of changes in the model [33]. The dependence analysis-based minimization technique was applied to the set of test cases that were identified to execute the modified transitions. Chen *et al.* extended Korel's model-based approach to incorporate more complex representations of model changes [34].

A risk shared by most test suite minimization techniques is that a *discarded* test case may detect a fault. In some domains, however, test suite minimization techniques can enjoy the certainty of guaranteeing that discarding a test case will not reduce the fault-detection capability. Anido *et al.* investigated test suite minimization for testing Finite State Machines (FSMs) in this context [35]. When only some components of the SUT need testing, the system can be represented as a composition of two FSMs: *component* and *context*. The context is assumed to be fault-free. Therefore, certain transitions of the system that concern only the context can be identified to be redundant. Under the 'testing in context' assumption (i.e. the context is fault-free), it follows that it is possible to guarantee that a discarded test case cannot detect faults.

Kaminski and Ammann investigated the use of a logic criterion to reduce test suites while guaranteeing fault detection in testing predicates over Boolean variables [36]. From the formal description of fault classes, it is possible to derive a hierarchy of fault classes [37]. From the hierarchy, it follows that the ability to detect a class of faults may guarantee the detection of another class. Therefore, the size of a test suite can be reduced by executing only those test cases for the class of faults that subsume another class, whenever this is feasible.

3.2. Impact on fault-detection capability

Although the techniques discussed so far reported reduced test suites, there has been a persistent concern about the effect that the test suite minimization has on the fault-detection capability of test suites. Several empirical studies were conducted to investigate this effect [8, 38–40].

Wong *et al.* studied 10 common Unix programs using randomly generated test suites; this empirical study is often referred to as the WHLM study [39]. To reduce the size of the test suites, they used the ATAC testing tool developed by Horgan and London [13, 15]. First, a large pool of test cases was created using a random test data generation method. From this pool, several test suites with different total block coverage were generated. After generating test suites randomly,

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

artificial faults were seeded into the programs. These artificial faults were then categorized into four groups. Faults in Quartile-I can be detected by [0–25)% of the test cases from the original test suite; the percentage for Quartile-II, III and IV is [25–50)%, [50–75)% and [75–100]%, respectively. Intuitively, faults in Quartile-I are harder to detect than those in Quartile-IV. The effectiveness of the minimization itself was calculated as follows:

$$\left(1-\frac{\text{number of test cases in the reduced test suite}}{\text{number of test cases in the original test suite}}\right)*100\%$$

The impact of test suite minimization was measured by calculating the reduction in fault detection effectiveness as follows:

$$\left(1-\frac{\text{number of faults detected by the reduced test suite}}{\text{number of faults detected by the original test suite}}\right)*100\%$$

By categorizing the test suites (by different levels of block coverage) and test cases (by difficulty of detection), they arrived at the following observation. First, the reduction in size is greater in test suites with a higher block coverage in most cases. This is natural considering that test suites with higher block coverage will require more test cases in general. The average size reduction for test suites with (50–55)%, (60–65)%,(70–75)%,(80–85)% and (90–95)% block coverage was 1.19, 4.46, 7.78, 17.44, 44.23%, respectively. Second, the fault-detection effectiveness was decreased by test case reduction, but the overall decrease in fault-detection effectiveness is not excessive and could be regarded as worthwhile for the reduced cost. The average effectiveness reduction for test suites with (50–55)%, (60–65)%,(70–75)%,(80–85)% and (90–95)% block coverage was 0, 0.03, 0.01, 0.38, 1.45%, respectively. Third, test suite minimization did not decrease the fault-detection effectiveness for faults in Quartile-IV at all, meaning that all faults in Quartile-IV had been detected by the reduced test suite. The average decrease in fault-detection effectiveness for Quartile-I, II and III was 0.39, 0.66, and 0.098%, respectively. The WHLM study concluded that, if the cost of testing is directly related to the number of test cases, then the use of the reduction technique is recommended.

Wong *et al*. followed up on the WHLM study by applying the ATAC tool to test suites of another, bigger C program; this empirical study is often referred to as the WHLP study [40]. The studied program, `space`, was an interpreter for the Array Description Language (ADL) developed by the European Space Agency. In the WHLP study, test cases were generated, not randomly, but from the operational profiles of `space`. That is, each test case in the test case pool was generated so that it matches an example of real usage of `space` recorded in an operational profile. From the test case pool, different types of test suites were generated. The first group of test suites was constructed by randomly choosing a fixed number of test cases from the test case pool. The second group of test suites was constructed by choosing test cases from the test case pool until a predetermined block coverage target was met. The faults in the program were not artificial, but real faults that were retrieved from development logs.

The results of the WHLP study confirmed the findings of the WHLM study. As in the WHLM study, test suites with low initial block coverage (50, 55, 60 and 65%) showed no decrease in fault-detection effectiveness, after test suite minimization. For both the fixed size test suites and fixed coverage test suites, the application of the test case reduction technique did not affect the fault-detection effectiveness in any significant way; the average effectiveness reduction due to test suite minimization was less than 7.28%.

While both the WHLM and WHLP studies showed that the impact of test suite minimization on fault-detection capability was insignificant, other empirical studies produced radically different findings. Rothermel *et al.* also studied the impact of test suite minimization on the fault-detection capability [38]. They applied the HGS heuristics to the Siemens suite [41], and later expanded this to include `space` [8]. The results from these empirical studies contradicted the previous findings of the WHLM and WHMP studies.

For the study of the Siemens suite [38], Rothermel *et al.* constructed test suites from the test case pool provided by the Siemens suite so that the test suites include varying amounts of redundant

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

test cases that do not contribute to the decision coverage of the test suite. The effectiveness and impact of reduction was measured using the same metrics that were used in the WHLM study.

Rothermel *et al.* reported that the application of the test suite minimization technique produced significant savings in test suite size. The observed tendency in size reduction suggested a logarithmic relation between the original test suite size and the reduction effectiveness. The results of logarithmic regression confirmed this.

However, Rothermel *et al.* also reported that, due to the size reduction, the fault-detection capabilities of test suites were severely compromised. The reduction in fault-detection effectiveness was over 50% for more than half of 1000 test suites considered, with some cases reaching 100%. Rothermel *et al.* also reported that, unlike the size reduction effectiveness, the fault-detection effectiveness did not show any particular correlation with the original test suite size.

This initial empirical study was subsequently extended [8]. For the Siemens suite, the results of the HGS heuristic were compared to random reduction by measuring the fault-detection effectiveness of randomly reduced test suites. Random reduction was performed by randomly selecting, from the original test suite, the same number of test cases as in the reduced version of the test suite. The results showed that random reduction produced larger decreases in fault-detection effectiveness. To summarize the results for the Siemens suite, the test suite minimization technique produced savings in test suite size, but at the cost of decreased fault-detection effectiveness; however, the reduction heuristic showed better fault-detection effectiveness than the random reduction technique.

Rothermel *et al.* also expanded the previous empirical study by including the larger program, `space` [8]. The reduction in size observed in the test suites of `space` confirmed the findings of the previous empirical study of the Siemens suite; the size reduction effectiveness formed a logarithmic trend, plotted against the original test suite size, similar to the programs in the Siemens suite. More importantly, the reduction in fault-detection effectiveness was less than those of the Siemens suite programs. The average reduction in fault-detection effectiveness of test suites reduced by the HGS heuristic was 8.9%, whereas. that of test suites reduced by random reduction was 18.1%.

Although the average reduction in fault-detection effectiveness is not far from that reported for the WHLP study in the case of `space`, those of the Siemens suite differed significantly from both the WHLP study and the WHLM study, which reported that the application of the minimization technique did not have significant impact on fault-detection effectiveness. Rothermel *et al.* [8] pointed out the following differences between these empirical studies as candidates for the cause(s) of the contradictory findings, which is paraphrased as follows:

1. *Different subject programs*: the programs in the Siemens suite are generally larger than those studied in both the WHLM and WHLP studies. Difference in program size and structure certainly could have impact on the fault-detection effectiveness.
2. *Different types of test suites*: the WHLM study used test suites that were not coverage-adequate and were much smaller compared to test suites used by Rothermel *et al.* The initial test pools used in the WHLM study also did not necessarily contain any minimum number of test cases per covered item. These differences could have contributed to less redundancy in test suites, which led to reduced likelihood that test suite minimization will exclude fault-revealing test cases.
3. *Different types of test cases*: the test suites used in the WHLM study contained a few test cases that detected all or most of the faults. When such strong test cases are present, reduced versions of the test suites may well show little loss in fault-detection effectiveness.
4 *Different types of faults*: the faults studied by Rothermel *et al.* were all Quartile-I faults according to the definition of the WHLM study, whereas only 41% of the faults studied in the WHLM study belonged to the Quartile-I group. By having more 'easy-to-detect' faults, the test suites used in the WHLM study could have shown less reduction in fault-detection effectiveness after test suite minimization.

Considering the two contradicting empirical results, it is natural to conclude that the question of evaluating the effectiveness of the test suite minimization technique is very hard to answer in general and for all testing scenarios. The answer depends on too many factors, such as the structure

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

of the SUT, the quality of test cases and test suites and the types of faults present. This proliferation of potential contributory factors makes it very difficult to generalize any empirical result.

The empirical studies from WHLM, WHLP and Rothermel *et al.* all evaluated the effectiveness of test suite minimization in terms of two metrics: percentage size reduction and percentage fault-detection reduction. McMaster and Memon noticed that neither metric considers the actual role each testing requirement plays on fault detection [42]. Given a set of test cases, TC, a set of known faults, KF and a set of testing requirements, CR, *fault correlation* for a testing requirement $i \in CR$ to fault $k \in KF$ is defined as follows:

$$\frac{|\{j \in TC | j \text{ covers } i\} \cap \{j \in TC | j \text{ detects } k\}|}{|\{j \in TC | j \text{ covers } i\}|}$$

The expected probability of finding a given fault k after test suite minimization is defined as the maximum fault correlation of all testing requirements with k. From this, the probability of detecting all known faults in KF is the product of expected probability of finding all $k \in KF$. Since CR is defined by the choice of minimization criterion, e.g. branch coverage or call-stack coverage, comparing the probability of detecting all known faults provides a systematic method of comparing different minimization criteria, without depending on a specific heuristic for minimization. The empirical evaluation of McMaster and Memon compared five different minimization criteria for the minimization of test suites for GUI-intensive applications: event coverage (i.e. each event is considered as a testing requirement), event interaction coverage (i.e. each pair of events is considered as a testing requirement), function coverage, statement coverage and call-stack coverage proposed in [43]. While call-stack coverage achieved the highest average probability of detecting all known faults, McMaster and Memon also found that different faults correlate more highly with different criteria. This analysis provides valuable insights into the selection of minimization criterion.

Yu *et al.* considered the effect of test suite minimization on fault localization [44]. They applied various test suite minimization techniques to a set of programs, and measured the impact of the size reduction on the effectiveness of coverage-based fault localization techniques. Yu *et al.* reported that higher reduction in test suite size, typically achieved by statement coverage-based minimization, tends to have a negative impact on fault localization, whereas minimization techniques that maintain higher levels of redundancy in test suites have negligible impact.

4. TEST CASE SELECTION

Test case selection, or the RTS problem, is essentially similar to the test suite minimization problem; both problems are about choosing a subset of test cases from the test suite. The key difference between these two approaches in the literature is whether the focus is upon the changes in the SUT. Test suite minimization is often based on metrics such as coverage measured from a single version of the program under test. By contrast, in RTS, test cases are selected because their execution is relevant to the changes between the previous and the current version of the SUT.

To recall Definition 2, T' ideally should contain all the *faults-revealing* test cases in T, i.e. the test cases that will reveal faults in P'. In order to define this formally, Rothermel and Harrold introduced the concept of a *modification-revealing* test case [45]. A test case t is modification-revealing for P and P' if and only if $P(t) \neq P'(t)$. Given the following two assumptions, it is possible to identify the fault-revealing test cases for P' by finding the modification-revealing test cases for P and P'.

- **P-Correct-for-T Assumption**: It is assumed that, for each test case $t \in T$, when P was tested with t, P halted and produced the correct output.
- **Obsolete-Test-Identification Assumption**: It is assumed that there exists an effective procedure for determining, for each test case $t \in T$, whether t is obsolete for P'.

From these assumptions, it is clear that every test case in T terminates and produces correct output for P, and is also supposed to produce the same output for P'. Therefore, a modification-revealing

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

test case t must be also fault-revealing. Unfortunately, it is not possible to determine whether a test case t is fault-revealing against P' or not because it is undecidable whether P' will halt with t. Rothermel considers a weaker criterion for the selection, which is to select all *modification-traversing* test cases in T. A test case t is modification-traversing for P and P' if and only if:

1. it executes new or modified code in P' or
2. it formerly executed code that had been deleted in P'.

Rothermel also introduced the third assumption, which is the Controlled-Regression-Testing assumption.

- *Controlled-Regression-Testing Assumption*: When P' is tested with t, all factors that might influence the output of P', except for the code in P', are kept constant with respect to their states when P was tested with t.

Given that the Controlled-Regression-Testing assumption holds, a non-obsolete test case t can thereby be modification-revealing only if it is also modification-traversing for P and P'. Now, if the P-Correct-for-T assumption and the Obsolete-Test-Identification assumption hold along with the Controlled-Regression-Testing assumption, then the following relation also holds between the subset of fault-revealing test cases, T_{fr}, the subset of modification-revealing test cases, T_{mr}, the subset of modification-traversing test cases, T_{mt}, and the original test suite, T:

$$T_{fr} = T_{mr} \subseteq T_{mt} \subseteq T$$

Rothermel and Harrold admitted that the Controlled-Regression-Testing assumption may not be always practical, since certain types of regression testing may make it impossible to control the testing environment, e.g. testing of a system ported to different operating system [1]. Other factors like non-determinism in programs and time dependencies are also difficult to control effectively. However, finding the subset of modification-traversing test cases may still be a useful approach in practice, because T_{mt} is the closest approximation to T_{mr} that can be achieved without executing all test cases. In other words, by finding T_{mt}, it is possible to exclude those test cases that are guaranteed not to reveal any fault in P'. The widely used term, *safe* RTS, is based on this concept [46]. A safe RTS technique is not safe from all possible faults; however, it is safe in a sense that, if there exists a modification-traversing test case in the test suite, it will definitely be selected.

Based on Rothermel's formulation of the problem, it can be said that test case selection techniques for regression testing focus on identifying the modification-traversing test cases in the given test suite. The details of the selection procedure differ according to how a specific technique defines, seeks and identifies modifications in the program under test. Various approaches have been proposed using different techniques and criteria including Integer Programming [47, 48], data-flow analysis [49–52], symbolic execution [53], dynamic slicing [54], CFG graph-walking [46, 55–57], textual difference in source code, [58, 59] SDG slicing [60], path analysis [61], modification detection [62], firewall [63–66], CFG cluster identification [67] and design-based testing [68, 69]. The following subsections describe these in more detail, highlighting their strengths and weaknesses.

4.1. Integer programming approach

One of the earliest approaches to test case selection was presented by Fischer *et al.*, who used Integer Programming (IP) to represent the selection problem for testing FORTRAN programs [47, 48]. Lee and He implemented a similar technique [70]. Fischer first defined a program segment as a single-entry, single-exit block of code whose statements are executed sequentially. Their selection algorithm relies on two matrices that describe the relation between program segments and test cases, as well as between different program segments.

For a program with m segments and n test cases, the IP formulation is given as the problem of finding the decision vector, $<x_1, \ldots, x_n>$, that minimizes the following objective function, Z:

$$Z = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

subject to:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \geq b_1$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \geq b_2$$
$$\vdots$$
$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \geq b_m$$

The decision vector, $<x_1, \ldots, x_n>$, represents the subset of selected test cases; x_i is equal to 1 if the ith test case is included; 0 otherwise. The coefficients, $c1, \ldots, c_n$, represent the cost of executing each corresponding test case; Fischer *et al.* used the constant value of 1 for all coefficients, treating all test cases as being equally expensive. The test case dependency matrix, $a_{11}, \ldots, a_{mn}$ represents the relations between test cases and the program segments. The element a_{ij} is equal to 1 if the ith program segment is executed by the test case j; 0 otherwise.

After deriving the series of inequalities, the set of b_k values are determined by using a reachability matrix that describes the program segments that are reachable from other segments. Using this approach, when the modified segments are known, it is possible to calculate all the segments that are reachable from the modified segments, which thus need to be tested at least once. The integer programming formulation is completed by assigning 1 to the b values for all the segments that need to be tested. The inequality, $a_{i1}x_1 + \cdots + a_{in}x_n \geq b_i$, thus ensures that at least one included test case covers the program element reachable from a change.

Hartmann and Robson implemented and extended a version of Fischer's algorithm in order to apply the technique to C [71–73]. They treat subroutines as segments, achieving subroutine coverage rather than statement coverage.

One weakness in Fischer's approach is its inability to deal with control-flow changes in P'. The test case dependency matrix, $a_{11}, \ldots, a_{mn}$, depends on the control-flow structure of the program under test. If the control-flow structure changes, the test case dependency matrix can be updated only by executing all the test cases, which negates the point of applying the selection technique.

4.2. Data-flow analysis approach

Several test case selection techniques have been proposed based on data-flow analysis [49–52]. Data-flow analysis-based selection techniques seek to identify new, modified or deleted definition–use pairs in P', and select test cases that exercise these pairs.

Harrold and Soffa presented data-flow analysis as the testing criterion for an incremental approach to unit testing during the maintenance phase [50]. Taha *et al.* built upon this idea and presented a test case selection framework based on an incremental data-flow analysis algorithm [52]. Harrold and Soffa developed both intra-procedural and inter-procedural selection techniques [51, 74]. Gupta *et al.* applied program slicing techniques to identify definition–use pairs that are affected by a code modification [49]. The use of slicing techniques enabled identification of definition–use pairs that need to be tested without performing a complete data-flow analysis, which is often very costly. Wong *et al.* combined a data-flow selection approach with a coverage-based minimization and prioritization to further reduce the effort [75].

One weakness shared by all data-flow analysis-based test case selection techniques is the fact that they are unable to detect modifications that are unrelated to data-flow change. For example, if P' contains new procedure calls without any parameter, or modified output statements that contain no variable uses, data-flow techniques may not select test cases that execute these.

Fisher II *et al.* applied the Data-flow-based RTS approach for test re-use in spreadsheet programs [76]. Fisher II *et al.* proposed an approach called What-You-See-Is-What-You-Test (WYSIWYT) to provide incremental, responsive and visual feedback about the *testedness* of cells in spreadsheets. The WYSIWYT framework collects and updates data-flow information incrementally as the user of the spreadsheet makes modifications to cells, using Cell Relation Graph.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Interestingly, the data-flow analysis approach to re-test spreadsheets is largely free from the difficulties associated with data-flow testing of procedural code, because spreadsheet programs lack tricky semantic features such as aliasing and unstructured control flow. This makes spreadsheet programs better suited to a data-flow analysis approach.

4.3. Symbolic execution approach

Yau and Kishmoto presented a test case selection technique based on symbolic execution of the SUT [53]. In symbolic execution of a program, the variables' values are treated as symbols, rather than concrete values [77]. Yau and Kishimoto's approach can be thought of as an application of symbolic execution and input partitioning to the test case selection problem. First, the technique statically analyses the code and specifications to determine the input partitions. Next, it produces test cases so that each input partition can be executed at least once. Given information on where the code has been modified, the technique then identifies the edges in the control-flow graph (CFG) that lead to the modified code. While symbolically executing all test cases, the technique determines test cases that traverse edges that do not reach any modification. The technique then selects all test cases that reach new or modified code. For the symbolic test cases that reach modifications, the technique completes the execution; the real test cases that match these symbolic test cases should be retested.

While it is theoretically powerful, the most important drawback of the symbolic execution approach is the algorithmic complexity of the symbolic execution. Yau and Kishmoto acknowledge that symbolic execution can be very expensive. Pointer arithmetic can also present challenging problems for symbolic execution-based approaches.

4.4. Dynamic slicing-based approach

Agrawal *et al.* introduced a family of test case selection techniques based on different program slicing techniques [54]. An *execution slice* of a program with respect to a test case is what is usually referred to as an execution trace; it is the set of statements executed by the given test case. A *dynamic slice* of a program with respect to a test case is the set of statements in the execution slice that have an influence on an output statement. Since an execution slice may contain statements that do not affect the program output, a dynamic slice is a subset of an execution slice. For example, consider the program shown in Figure 1. It contains two faults in line `S3` and `S11`, respectively. The execution slice of the program with respect to test case T_3 in Table II is shown in column `ES` of Figure 1. The dynamic slice of the program with respect to test case T_1 in Table II is shown in column `DS` of Figure 1.

In order to make the selection more precise, Agrawal *et al.* proposed two additional slicing criteria: a *relevant slice* and an *approximate relevant slice*. A relevant slice of a program with respect to a test case is the dynamic slice with respect to the same test case together with all the predicate statements in the program that, if evaluated differently, could have caused the program to produce a different output. An approximated relevant slice is a more conservative approach to include predicates that could have caused a different output; it is the dynamic slice with all the predicate statements in the execution slice. By including all the predicates in the execution slice, an approximated relevant slice caters for the indirect references via pointers. For example, consider the correction of `S3` in the program shown in Figure 1. The dynamic slice of T_4 does not include `S3` because the `class` value of T_4 is not affected by any of the lines between `S3` and `S8`. However, the relevant slice of T_4, shown in column `DS` of Figure 1, does include `S3` because it could have affected the output when evaluated differently.

The test suite and the previous version of the program under test are preprocessed using these slicing criteria; each test case is connected to a slice, *sl*, constructed by one of the four slicing criteria. After the program is modified, test cases for which *sl* contains the modified statement should be executed again. For example, assume that the fault in `S11`, detected by T_5, is corrected. The program should be retested with T_5. However, T_3 need not be executed because the execution slice of T_3, shown in column `ES` of Figure 1, does not contain `S11`. Similarly, assume that the fault in `S3`, detected by T_6, is corrected. The program should be retested with T_6. The execution

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

```
Slices
ES DS RS
x  x  x    S1: read(a,b,c);
x     x    S2: class := scalene;
x     x    S3: if a = b or b = a
           S4:      class := isosceles;
x     x    S5: if a * a = b * b + c * c
x          S6:      class := right
x  x  x    S7: if a = b and b = c
   x       S8:      class := equilateral
x  x  x    S9: case class of:
x         S10:      right        : area = b * c / 2;
   x      S11:      equilateral  : area = a * 2 * sqrt(3) / 4;
      x   S12:      otherwise    : s := (a + b + c) / 2;
      x   S13:                     area := sqrt(s * (s-a) * (s-b) * (s-c));
x         S14: end;
x         S15: write(class, area);
```

Figure 1. Example triangle classification program taken from Agrawal *et al.* [54]. Note that it is assumed that the input vector is sorted in descending order. It contains two faults. In `S3`, `b = a` should be `b = c`. In `S11`, `a * 2` should be `a * a`. The column `ES` represents the statements that belong to the execution slice with respect to test case T_3 in Table II. Similarly, column `DS` represents the dynamic slice with respect to test case T_1 in Table II and column `RS` the relevant slice with respect to test case T_4 in Table II.

Table II. Test cases used with the program shown in Figure 1, taken from Agrawal *et al.* [54]. Note that T_5 detects the fault in `S11`, because the value for area should be 3.90. Similarly, T_6 detects the fault in `S3`, because the value for class should be isosceles.

	Input			Output	
Test case	a	b	c	Class	Area
T_1	2	2	2	equilateral	1.73
T_2	4	4	3	isosceles	5.56
T_3	5	4	3	right	6.00
T_4	6	5	4	scalene	9.92
T_5	3	3	3	equilateral	2.60
T_6	4	3	3	scalene	4.47

slice technique selects all six test cases, T_1–T_6, after the correction of the fault in `S3` because the execution slices of all six test cases include `S3`. However, it is clear that T_1 and T_3 are not affected by the correction of `S3`; their `class` values are overwritten after the execution of `S3`. The dynamic slicing technique overcomes this weakness. The dynamic slice of T_1 is shown in column `DS` of Figure 1. Since `S3` does not affect the output of T_1, it is not included in the dynamic slice. Therefore, the modification of `S3` does not necessitate the execution of T_1.

Agrawal *et al.* first built their technique on cases in which modifications are restricted to those that do not alter the CFG of the program under test. As long as the CFG of the program remains the same, their technique is safe and can be regarded as an improvement over Fischer's integer programming approach. Slicing removes the need to formulate the linear programming problem, reducing the effort required from the tester. Agrawal *et al.* later relaxed the assumption about static CFGs in order to cater for modifications in the CFGs of the SUT. If a statement s is added to P, now the slice sl contains all the statements in P that uses the variables defined in s. Similarly, if a predicate p is added to P, the slice sl contains all the statements in P that are control-dependent on p. This does cater for the changes in the CFG to some degree, but it is not complete. For example, if the added statement is a simple output statement that does not define or use any variable, then this statement can still be modification-revealing. However, since the new statement does not contain any variable, its addition will not affect any of the existing slices, resulting in an empty selection.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

4.5. Graph-walk approach

Rothermel and Harrold presented regression test case selection techniques based on graph-walking of Control Dependence Graphs (CDGs), Program Dependence Graphs (PDGs), System Dependence Graphs (SDGs) and CFGs [46, 56, 57, 78]. The CDG is similar to PDG but lacks data dependence relations. By performing a depth-first traversal of the CDGs of both P and P', it is possible to identify points in a program through which the execution trace reaches the modifications [46]. If a node in the CDG of P is not lexically equivalent to the corresponding node in the CDG of P', the algorithm selects all the test cases that execute the control-dependence predecessors of the mismatching node. The CDG-based selection technique does not cater for inter-procedural regression test case selection; Rothermel and Harrold recommend application of the technique at the individual procedural level.

Rothermel and Harrold later extended the graph-walking approach to use PDGs for intra-procedural selection, and SDGs for inter-procedural selection [56]. A weakness of the CDG-based technique is that, due to the lack of data dependence, the technique will select test cases that execute modified definitions but not the actual uses of a variable. If the modified definition of a variable is never used, it cannot contribute to any different output, and therefore its inclusion is not necessary for safe regression testing. PDGs contain data dependence for a single procedure; SDGs extend this to a complete program with multiple procedures. By using these graphs, Rothermel and Harrold's algorithm is able to check whether a modified definition of a variable is actually used later.

Rothermel and Harrold later presented the graph-walking approach based on CFGs [57]. The CFG-based technique essentially follows the approach introduced for the CDG-based technique, but on CFGs rather than on CDGs. Since CFG is a much simpler representation of the structure of a program, the CFG-based technique may be more efficient. However, the CFG lacks data dependence information, so the CFG-based technique may select test cases that are not capable of producing different outputs from the original programs, as explained above. The technique has been evaluated against various combinations of subject programs and test suites [79]. Ball improved the precision of the graph-walk approach with respect to branch coverage [80].

Rothermel *et al.* extended the CFG-based graph-walk approach for object-oriented software using the Inter-procedural Control-Flow Graph (ICFG) [81]. The ICFG connects methods using *call* and *return* edges. Harrold *et al.* adopted a similar approach for test case selection for Java software, using the Java Interclass Graph as representation (JIG) [82]. Xu and Rountev later extended this technique to consider AspectJ programs by incorporating the interactions between methods and advices at certain join points into the CFG [83]. Zhao *et al.* also considered a graph representation of AspectJ programs to apply a graph-walk approach for RTS [84]. Beydeda and Gruhn extended the graph-walk approach by adding black-box data-flow information to the Class Control-Flow Graph (CCFG) to test object-oriented software [85].

Orso *et al.* considered using different types of graph representation of the system to improve the scalability of graph-walk approach [86]. Their approach initially relies on a high-level graph representation of SUT to identify the parts of the system to be further analysed. Subsequently, the technique uses more detailed graph representation to perform more precise selection.

One strength of the graph-walk approach is its generic applicability. For example, it has been successfully used in black-box testing of re-usable classes [87]. Martins and Vieira captured the behaviours of a re-usable class by constructing a directed graph called the Behavioural Control-Flow Graph (BCFG) from the Activity Diagram (AD) of the class. The BCFG is a directed graph, $G = (V, E, s, x)$, with vertices V, edges E, a unique entry vertex s and an exit vertex x. Each vertex contains a label that specifies the signature of a method; each edge is also labelled according to the corresponding guards in AD. A path in G from s to x represents a possible life history of an object. By mapping changes made to the object to its BCFG and applying the graph-walking algorithm, it is possible to select test cases based on the behavioural difference between two versions of the same object. This approach requires traceability between the behavioural model and the actual test cases, because test cases are selected, not based on their structural coverage, but based on their behavioural coverage measured on BCFG. ADs have also been directly used for RTS by Chen *et al.* [88].

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Orso *et al.* used a variation of the graph-walk approach to consider an RTS technique based on meta-data and specifications obtained from software components [89, 90]. They presented two different techniques based on meta-data: code-based RTS using component meta-data and specification-based RTS using component meta-data. For code-based RTS, it was assumed that each software component was capable of providing structural coverage information, which was fed into the graph-walk algorithm. For specification-based RTS, the component specification was represented in UML state-chart diagrams, which were used by the graph-walk algorithm.

The graph-walk algorithm has also been applied to test web services, despite the challenges that arise from the distributed nature of web services [91–95]. Several different approaches have been introduced to overcome these challenges. Lin *et al.* adopted the JIG-based approach after transforming the web services to a single-JVM local application [91]. Ruth *et al.* collected a coarse-grained CFG from developers of each web service that forms a part of the entire application [92, 93, 95]. Finally, Tarhini *et al.* utilized Timed Labelled Transition System (TLTS), which is a coarse-grained representation of web services that resemble a labelled state machine [94].

4.6. Textual difference approach

Volkolos and Frankl proposed a selection technique based on the textual difference between the source code of two versions of SUT [58, 59]. They identified modified parts of SUT by applying the `diff` Unix tool to the source code of different versions. The source code was pre-processed into canonical forms to remove the impact of cosmetic differences. Although their technique operates on a different representation of SUT, its behaviour is essentially very similar to that of the CFG-based graph-walk approach.

4.7. SDG slicing approach

Bates and Horwitz proposed test case selection techniques based on program slices from Program Dependency Graphs (PDGs) [60]. Bates and Horwitz approach the RTS problem in two stages. First, all the test cases that can be reused for P' need to be identified. Bates and Horwitz introduce the definition of an equivalent execution pattern. If statements s and s' belong to P and P', respectively, s and s' have *equivalent execution patterns* if and only if all of the following hold:

1. For any input file on which both P and P' terminate normally, s and s' are exercised the same number of times.
2. For any input file on which P terminates normally but P' does not, s' is exercised at most as many times as s is exercised.
3. For any input file on which P' terminates normally but P does not, s is exercised at most as many times as s' is exercised.

Using program slicing, Bates and Horwitz categorize statements into execution classes. Statement s from P and s' from P' belong to the same execution class if and only if any test that exercises s will also exercise s'.

Now, a statement s' in P' is *affected* by the modification if and only if one of the following holds:

1. There is no corresponding statement s in P.
2. The behaviour of s' is not equivalent to the corresponding statement s in P.

Equivalent behaviour is determined by PDG slice isomorphism; if the PDG slices of two statements are isomorphic, then those statements share an equivalent behaviour. For each affected statement in P', reusable test cases are selected based on the information retrieved from the identification stage.

While Bates and Horwitz's technique selects test cases for modified or newly added statements in P', it does not select tests that exercise statements that are deleted from P, and therefore is not safe.

Binkley [96, 97] presented a technique based on SDG slicing, which extends Bates and Horwitz's intra-procedural selection technique to inter-procedural test case selection. Binkley introduced the

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

concept of *common execution patterns*, which corresponds to the equivalent execution patterns of Bates and Horwitz, to capture the multiple invocations of a procedure.

4.8. Path analysis

Benedusi *et al.* applied path analysis for test case selection [61]. They construct *exemplar paths* from P and P' expressed in an algebraic expression. By comparing two sets of exemplar paths, they classified paths in P' as new, modified, cancelled or unmodified. Test cases and the paths they execute in P are known; therefore, they selected all the test cases that will traverse modified paths in P'.

One potential weakness of the path analysis approach of Benedusi *et al.* lies not in path analysis itself, but in the potentially over-specific definition of 'modification' used in the post-analysis selection phase. No test cases are selected for the paths that are classified as new or cancelled. However, new or cancelled paths denote modifications that represent differences between P and P'; test cases that execute new or cancelled paths in P' may be modification-revealing. As presented, therefore, the path analysis approach is not safe.

4.9. Modification-based technique

Chen *et al.* introduced a testing framework called TestTube, which utilizes a modification-based technique to select test cases [62]. TestTube partitions the SUT into *program entities*, and monitors the execution of test cases to establish connections between test cases and the program entities that they execute. TestTube also partitions P' into program entities, and identifies program entities that are modified from P. All the test cases that execute the modified program entities in P should be re-executed.

TestTube can be thought of as an extended version of the graph-walk approach. Both techniques identify modifications by examining the program source code, and select test cases that will execute the modified parts. TestTube extends the CDG-based graph-walk technique by introducing program entities that include both functions and entities that are not functions, i.e. variables, data types and pre-processor macros. Any test case that executes modified functions will be selected. Therefore, TestTube is a safe test case selection technique.

One weakness of TestTube is pointer handling. By including variable and data types as program entities, TestTube requires that all value creations and manipulations in a program can be inferred from source code analysis. This is only valid for languages without pointer arithmetic and type coercion. As a result, TestTube makes assumptions; for example, it assumes that all pointer arithmetics are well-bounded. If these assumptions do not hold then safety cannot be guaranteed.

4.10. Firewall approach

Leung and White introduced and later implemented what they called a *firewall* technique for regression testing of system integration [63–66]. The main concept is to draw a firewall around the modules of the system that need to be retested. They categorize modules into the following categories:

- *No Change*: Module A has not been modified, $NoCh(A)$.
- *Only Code Change*: Module A has the same specification but its code has been modified, $CodeCh(A)$.
- *Spec Change*: module A has modified specifications, $SpecCh(A)$.

If a module A calls a module B, there exist 9 possible pairings between the states of A and B. The integration between A and B can be ignored for regression testing if $NoCh(A) \wedge NoCh(B)$, leaving 8 pairings. If both A and B are modified either in code or in specifications, the integration tests between A and B should be executed again as well as the unit tests of A and B; this accounts for 4 of the remaining 8 pairings. The other 4 pairings are cases in which an unchanged module calls a changed module, or vice versa; these pairs form the boundary for integration testing, i.e. the so-called firewall.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

By considering modules as the atomic entities, Leung and White maintained a very conservative approach to test case selection. If a module has been modified, any test case that tests the integration of the modified module should be selected. Therefore, all modification-traversing test cases will be selected. However, their technique may also select other test cases that execute the modified module, but are not modification-traversing in any way. Leung and White also noted that, in practice, the test suite for system integration is often not very reliable. The low reliability means that it is more likely that there may still exist a fault-revealing test case that does not belong to the test suite, and therefore cannot be selected. Note that it is always a risk that a fault-revealing test case exists outside the given test suite in any type of testing, not only in integration testing. What Leung and White pointed out was that such a risk can be higher in system integration testing due to the generally low quality of test suites.

The Firewall approach has been applied to Object-oriented programs [98–100] and GUIs [101]. Firewall approach has also been successfully applied to RTS for black-box Commercial Off-the-Shelf components. Zheng *et al.* applied the firewall technique of Leung and White based on the information extracted from the deployed binary code [102–105]. Skoglund and Runeson applied the firewall approach to a large-scale banking system [106].

4.11. Cluster identification

Laski and Szemer presented a test case selection technique based on analysis of the CFG of the program under test [67]. Their technique identifies single-entry, single-exit subgraphs of CFG called clusters. Given a program P and its modified version P',

- each cluster in P encapsulates some modifications to P,
- there is a unique cluster in P' that corresponds to the cluster in P, and
- when clusters in each graph are replaced by single nodes, there is a one-to-one correspondence between nodes in both graphs.

The CFGs of the original program and the modified program are reduced using a set of operators such as node collapse and node removal. During the process, if the counterpart of a node from the CFG of the original program cannot be found in the CFG of the modified program, this node is labelled as 'MOD', indicating a modification at the node. Eventually, all the modifications will be enclosed in one or more MOD cluster nodes. As with other test case selection techniques, their technique requires that the tester records the execution history of each test case in the test suite. Once clustering is completed, test case selection is performed by selecting all the test cases for which the corresponding execution path enters any of the MOD clusters.

The strength of the cluster identification technique is that it guarantees to select all modification-traversing test cases regardless of the type of the modification, i.e. addition or deletion of statements and control structures. However, since the clusters can encapsulate much larger areas of the SUT than the scope of actual modification, the technique may also select test cases that are not modification-traversing. In this sense the approach sacrifices precision in order to achieve safety.

4.12. Design-based approach

Briand *et al.* presented a black-box, design level RTS approach for UML-based designs [68, 69]. Assuming that there is traceability between the design and regression test cases, it is possible to perform RTS of code-level test cases from the impact analysis of UML design models. Briand *et al.* formalized possible changes in UML models, and classified the relevant test cases into the categories defined by Leung and White [10]: obsolete, retestable and reusable. They implemented an automated impact analysis tool for UML and empirically evaluated it using both student projects and industrial case studies.

The results showed that the changes made to a model can have a widely variable impact on the resulting system, which, in turn, yields varying degrees of reduction of effort in terms of the number of selected test cases. However, Briand *et al.* noted that the automated impact analysis itself can be valuable, especially for very large systems, such as the cruise control and monitoring system they studied. The UML use-cases of the model of the system had 323 614 corresponding

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

test cases. UML-based models also have been considered by Dent *et al.* [107], Pilskalns *et al.* [108] and Farooq *et al.* [109] for RTS; Le Traon *et al.* [110] and Wu and Offutt [111] considered the use of UML models in the wider context of regression testing in general. Muccini *et al.* considered the RTS problem at the software architecture level, although they did not use UML for the representation [112, 113].

5. TEST CASE PRIORITIZATION

Test case prioritization seeks to find the ideal ordering of test cases for testing, so that the tester obtains maximum benefit, even if the testing is prematurely halted at some arbitrary point. The approach was first mentioned by Wong *et al.* [39]. However, in that work it was only applied to test cases that were already selected by a test case selection technique. Harrold and Rothermel [114, 115] proposed and evaluated the approach in a more general context.

For example, consider the test suite described in Table III. Note that the example depicts an ideal situation in which fault-detection information is known. The goal of prioritization is to maximize early fault detection. It is obvious that the ordering A-B-C-D-E is inferior to B-A-C-D-E. In fact, any ordering that starts with the execution of C-E is superior to those that do not, because the subsequence C-E detects faults as early as possible; should testing be stopped prematurely, this ensures that the maximum possible fault coverage will have been achieved.

Note that the problem definition concerns neither versions of the program under test, nor exact knowledge of modifications. Ideally, the test cases should be executed in the order that maximizes early fault detection. However, fault-detection information is typically not known until the testing is finished. In order to overcome the difficulty of knowing which tests reveal faults, test case prioritization techniques depend on surrogates, hoping that early maximization of a certain chosen surrogate property will result in maximization of earlier fault detection. In a controlled-regression-testing environment, the result of prioritization can be evaluated by executing test cases according to the fault-detection rate.

5.1. Coverage-based prioritization

Structural coverage is a metric that is often used as the prioritization criterion [115–121]. The intuition behind the idea is that early maximization of structural coverage will also increase the chance of early maximization of fault detection. Therefore, while the goal of test case prioritization remains that of achieving a higher fault-detection rate, prioritization techniques actually aim to maximize early coverage.

Rothermel *et al.* reported empirical studies of several prioritization techniques [115, 121]. They applied the same algorithm with different fault-detection rate surrogates. The considered surrogates were: branch-total, branch-additional, statement-total, statement-additional, Fault Exposing Potential (FEP)-total and FEP-additional.

The branch-total approach prioritizes test cases according to the number of branches covered by individual test cases, while branch-additional prioritizes test cases according to the additional

Table III. Example test suite with fault-detection information, taken from Elbaum *et al*. [116]. It is clearly beneficial to execute test case C first, followed by E.

	Fault revealed by test case									
Test case	1	2	3	4	5	6	7	8	9	10
A	x				x					
B	x				x	x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

number of branches covered by individual test cases. The statement-total and statement-additional approaches apply the same idea to program statements, rather than branches. Algorithmically, 'total' approaches are essentially instances of greedy algorithms, whereas 'additional' approaches are essentially instances of additional greedy algorithms.

The FEP of a test case is measured using program mutation. Program mutation introduces a simple syntactic modification to the program source, producing a mutant version of the program [122]. This mutant is said to be *killed* by a test case if the test case reveals the difference between the original program and the mutant. Given a set of mutants, the mutation score of a test case is the ratio of mutants that are killed by the test case to the total kill-able mutants. The FEP-total approach prioritizes test cases according to the mutation score of individual test cases, while the FEP-additional approach prioritizes test cases according to the additional increase in mutation score provided by individual test cases. Note that FEP criterion can be constructed to be at least as *strong* as structural coverage; to kill a mutant, a test case not only needs to achieve the coverage of the location of mutation but also to execute the mutated part with a set of test inputs that can kill the mutant. In other words, coverage is necessary but not sufficient to kill the mutants.

It is important to note that all the 'additional' approaches may reach 100% realization of the utilized surrogate before every test case is prioritized. For example, achieving 100% branch coverage may not require all the test cases in the test suite, in which case none of the remaining test cases can increase the branch coverage. Rothermel *et al.* reverted to the 'total' approach once such a condition is met.

The results were evaluated using the Average Percentage of Fault-Detection (APFD) metric. Higher APFD values denote faster fault-detection rates. When plotting the percentage of detected faults against the number of executed test cases, APFD can be calculated as the area below the plotted line. More formally, let T be the test suite containing n test cases and let F be the set of m faults revealed by T. For ordering T', let TF_i be the order of the first test case that reveals the ith fault. The APFD value for T' is calculated as follows [123]:

$$APFD = 1 - \frac{TF_1 + \cdots + TF_m}{nm} + \frac{1}{2n}$$

Note that, while APFD is commonly used to evaluate test case prioritization techniques, it is not the aim of test case prioritization techniques to maximize APFD. Maximization of APFD would be possible only when every fault that can be detected by the given test suite is already known. This would imply that all test cases have been already executed, which would nullify the need to prioritize. APFD is computed after the prioritization only to evaluate the performance of the prioritization technique.

Rothermel *et al.* compared the proposed prioritization techniques to random prioritization, optimal prioritization and no prioritization, using the Siemens suite programs. Optimal prioritization is possible because the experiment was performed in a controlled environment, i.e. the faults were already known. The results show that all the proposed techniques produce higher APFD values than random or no prioritization. The surrogate with the highest APFD value differed between programs, suggesting that there is no single best surrogate. However, on average across the programs, the FEP-additional approach performed most effectively, producing APFD value of 74.5% compared to the 88.5% of the optimal approach. It should still be noted that these results are dependent on many factors, including the types of faults used for evaluation and types of mutation used for FEP, limiting the scope for generalization.

Elbaum *et al.* extended the empirical study of Rothermel *et al.* by including more programs and prioritization surrogates [116]. Among the newly introduced prioritization surrogates, function-coverage and function-level FEP enabled Elbaum *et al.* to study the effects of granularity on prioritization. Function-coverage of a test case is calculated by counting the number of functions that the test case executes. Function-level FEP is calculated, for each function f and each test case t, by summing the ratio of mutants in f killed by t. Elbaum *et al.* hypothesized that approaches with coarser granularity would produce lower APFD values, which was confirmed statistically.

Jones and Harrold applied the greedy-based prioritization approach to Modified Condition/Decision Coverage (MC/DC) criterion [124]. MC/DC is a 'stricter form' of branch coverage; it

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

requires execution coverage at condition level. A condition is a Boolean expression that cannot be factored into simpler Boolean expressions. By checking each condition in decision predicates, MC/DC examines whether each condition independently affects the outcome of the decision [125]. They presented an empirical study that contained only an execution time analysis of the prioritization technique and not an evaluation based on fault-detection rate.

Srivastava and Thiagarajan combined the greedy-based prioritization approach with RTS [126]. They first identified the modified code blocks in the new version of the SUT by comparing its binary code to that of the previous version. Once the modified blocks are identified, test case prioritization is performed using greedy-based prioritization, but only with respect to the coverage of modified blocks.

Do and Rothermel applied coverage-based prioritization techniques to the JUnit testing environment, a popular unit testing framework [127]. The results showed that prioritized execution of JUnit test cases improved the fault-detection rate. One interesting finding was that the random prioritization sometimes resulted in an APFD value higher than the *untreated* ordering, i.e. the order of creation. When executed in the order of creation, newer unit tests are executed later. However, it is the newer unit tests that have a higher chance of detecting faults. The empirical results showed that random prioritization could exploit this weakness of untreated ordering in some cases.

Li *et al.* applied various meta-heuristics for test case prioritization [128]. They compared random prioritization, a hill climbing algorithm, a genetic algorithm, a greedy algorithm, the additional greedy algorithm and a two-optimal greedy algorithm. The greedy algorithm corresponds to the *total* approaches outlined above, whereas the additional greedy algorithm corresponds to the *additional* approaches outlined above. The two-optimal greedy is similar to the greedy algorithm except that it considers two candidates at the same time rather than a single candidate for the next order. They considered the Siemens suite programs and the program `space`, and evaluated each technique based on Average Percentage of Block Coverage (APBC) instead of APFD. The results showed that the additional greedy algorithm is the most efficient in general.

5.2. Interaction testing

Interaction testing is required when the SUT involves multiple combinations of different components. A common example would be configuration testing, which is required to ensure that the SUT executes correctly on different combinations of environment, such as different operating systems or hardware options. Each component that can be changed is called a *factor*; the number of choices for each factor is called the *level* of the corresponding factor. As the number of factors and levels of each factor increase, exhaustive testing of all possible combinations of factors becomes infeasible as it requires an exponentially large test suite.

Instead of testing exhaustively, *pairwise* interaction testing requires only that every individual pair of interactions between different factors are included at least once in the testing process. The reduction grows larger as more factors and levels are involved. More formally, the problem of obtaining interaction testing combinations can be expressed as the problem of obtaining a *covering array*, $CA(N; t, k, v)$, which is an array with N rows and k columns; v is the number of levels associated with each factor, and t is the *strength* of the interaction coverage (2 in the case of pairwise interaction testing).

One important research direction in interaction testing is how to efficiently generate an interaction test suite with high interaction coverage. This shares the same basic principles of test case prioritization. For example, the greedy approach aims to find, one by one, the 'next' test case that will most increase the k-way interaction coverage [129, 130], which resembles the greedy approach to test case prioritization. However, the similarities are not just limited to the generation of interaction test suites. Bryce and Colbourn assume that testers may value certain interactions higher than others [131, 132]. For example, an operating system with a larger user base may be more important than one with a smaller user base. After weighting each level value for each factor, they calculate the combined benefit of a given test by adding the weights of each level value selected for the test. They present a Deterministic Density Algorithm (DDA) that prioritizes interaction

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

tests according to their combined benefit. Qu *et al.* compared different weighting schemes used for prioritizing covering arrays [133, 134].

Bryce and Memon also applied the principles of interaction coverage to the test case prioritization of Event-Driven Software (EDS) [135]. EDS takes sequences of events as input, changes state and outputs new event sequences. A common example would be GUI-based programs. Bryce and Memon interpreted t-way interaction coverage as sequences that contain different combinations of events over t unique GUI windows. Interaction coverage-based prioritization of test suites was compared to different prioritization techniques such as unique event coverage (the aim is to cover as many unique events as possible, as early as possible), longest to shortest (execute the test case with the longest event sequence first) and shortest to longest (execute the test case with the shortest event sequence first). The empirical evaluation showed that interaction coverage-based testing of EDS can be more efficient than the other techniques, provided that the original test suite contains higher interaction coverage. Note that Bryce and Memon did not try to generate additional test cases to improve interaction coverage; they only considered permutations of existing test cases.

5.3. Prioritization approaches based on other criteria

While the majority of existing prioritization literature concerns structural coverage in some form or another, there are prioritization techniques based on other criteria [136–139].

Distribution-based Approach: Leon and Podgurski introduced distribution-based filtering and prioritization [136]. Distribution-based techniques minimize and prioritize test cases based on the distribution of the profiles of test cases in the multi-dimensional profile space. Test case profiles are produced by the dissimilarity metric, a function that produces a real number representing the degree of dissimilarity between two input profiles. Using this metric, test cases can be clustered according to their similarities. The clustering can reveal some interesting information. For example:

- Clusters of similar profiles may indicate a group of redundant test cases
- Isolated clusters may contain test cases inducing unusual conditions that are perhaps more likely to cause failures
- Low density regions of the profile space may indicate uncommon usage behaviours

The first point is related to reduction of effort; if test cases in a cluster are indeed very similar, it may be sufficient to execute only one of them. The second and third points are related to fault-proneness. Certain unusual conditions and uncommon behaviours may tend to be harder to reproduce than more common conditions and behaviours. Therefore, the corresponding parts of the program are likely to be tested less than other, more frequently used parts of the program. Assigning a high priority to test cases that execute these unusual behaviours may increase the chance of early fault detection. A good example might be exception handling code.

Leon and Podgurski developed new prioritization techniques that combine coverage-based prioritization with distribution-based prioritization. This hybrid approach is based on the observation that *basic coverage maximization* performs reasonably well compared to *repeated coverage maximization*. Repeated coverage maximization refers to the prioritization technique of Elbaum *et al.* [116], which, after realizing 100% coverage, repeatedly prioritizes test cases starting from 0% coverage again. In contrast, basic coverage maximization stops prioritizing when 100% coverage is achieved. Leon and Podgurski observed that the fault-detection rate of repeated coverage maximization is not as high as that of basic coverage maximization. This motivated them to consider a hybrid approach that first prioritizes test cases based on coverage, then switches to distribution-based prioritization once the basic coverage maximization is achieved. They considered two different distribution-based techniques. The one-per-cluster approach samples one test case from each cluster, and prioritizes them according to the order of cluster creation during the clustering. The failure-pursuit approach behaves similarly, but it adds the k closest neighbours of any test case that finds a fault. The results showed that the distribution-based prioritization techniques could outperform repeated coverage maximization.

Human-based Approach: Tonella *et al.* combined Case-Based Reasoning (CBR) with test case prioritization [137]. They utilized a machine learning technique called *boosting*, which is a

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

framework to combine simple learners into a single, more general and effective learner [140]. They adopted a boosting algorithm for ranking learning called *Rankboost* [141]. The algorithm takes a test suite, T, an initial prioritization index, f, and a set of pairwise priority relations between test cases, Φ, as input. The pairwise priority relation is obtained from comparisons of test cases made by the human tester. The output is a ranking function $H : T \rightarrow \mathbb{R}$ such that, with test cases t_1 and t_2, $t_1 \prec t_2$ if $H(t_1) > H(t_2)$. The ranking function H is then used to prioritize test cases.

They used the statement coverage metric and the cyclomatic complexity computed for the functions executed by test cases as the initial prioritization index. The test suite of the `space` program was considered. In order to measure the human effort required for the learning process, different test suite sizes were adopted, ranging from 10 to 100 test cases. The results were compared to other prioritization techniques including optimal ordering, random prioritization, statement coverage prioritization and additional statement coverage prioritization (the latter two correspond to statement-total and statement-additional respectively).

The results showed that, for all test suite sizes, the CBR approach was outperformed only by the optimal ordering. The number of pairwise relations entered manually showed a linear growth against the size of test suites. Tonella *et al.* reported that for test suites of `space` with fewer than 60 test cases, the CBR approach can be more efficient than other prioritization techniques with limited human effort. Note that empirical evaluation was performed based on an *ideal* user model, i.e. it was assumed that the human tester always makes the correct decision when comparing test cases. One notable weakness of this approach was that it did not scale well. The input from the human tester becomes inconsistent beyond a certain number of comparisons, which in turn limits the size of the learning samples for CBR.

Yoo *et al.* tried to improve the scalability of human-based prioritization approaches by combining pairwise comparisons of test cases with a clustering technique [138]. While the prioritization is still based on the comparisons made by the human tester, the tester is presented with clusters of similar test cases instead of individual test cases. The prioritization between clusters (*inter-cluster* prioritization) is, therefore, performed by the tester. However, the prioritization within each cluster (*intra-cluster* prioritization) is performed based on coverage. After both layers of prioritization are complete, the final ordering of test cases is determined by selecting the test case with the highest priority, determined by intra-cluster prioritization, from the next cluster in the order determined by inter-cluster prioritization, until all the clusters are empty. This is called the Interleaved Clusters Prioritization (ICP) technique.

With the use of clustering, Yoo *et al.* were able to reduce the size of the prioritization problem so that they could apply a more expensive pairwise approach called Analytic Hierarchy Process (AHP). AHP is a pairwise comparison technique developed by the Operations Research community [142] and has been successfully applied to Requirements Engineering [143]. The combination of AHP and ICP has been empirically evaluated for programs and test suites of various sizes, using a more realistic user model (with errors). The results showed that this combination of techniques can be much more effective than coverage-based prioritization. One surprising finding was that sometimes, an error rate higher than 50%, i.e. the human tester making wrong comparisons half the time, did not prevent this technique from achieving higher APFD than coverage-based prioritization. Yoo *et al.* explained this unexpected finding by showing that a certain amount of improvement derived from the effect of clustering. This confirms the argument of Leon and Podgurski about the benefits of distribution-based approach [136]; the clustering sometimes enables the early execution of a fault-revealing test case that would have been assigned low priority due to its low contribution to code coverage.

Probabilistic Approach: Kim and Porter proposed a history-based approach to prioritize test cases that are already selected by RTS [144]. If the number of test cases selected by an RTS technique is still too large, or if the execution costs are too high, then the selected test cases may have to be further prioritized. Since the relevance to the recent change in SUT is assumed by the use of an RTS technique, Kim *et al.* focus on the execution history of each test case, borrowing from statistical quality control. They define the probabilities of each test case tc to be selected at time t as $P_{tc,t}(H_{tc}, \alpha)$, where H_{tc} is a set of t timed observations $\{h_1, \ldots, h_t\}$ drawn from previous

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

runs of tc and α is a smoothing constant. Then the probability $P_{tc,t}(H_{tc}, \alpha)$ is defined as follows:

$$P_0 = h_1$$
$$P_k = \alpha h_k + (1-\alpha) P_{k-1} \quad (0 \leq \alpha \leq 1, k \geq 1)$$

Different definitions of H_{tc} result in different prioritization approaches. For example, Kim *et al.* define Least Recently Used (LRU) prioritization by using test case execution history as H_{tc} with α value that is as close to 0 as possible. The empirical evaluation showed that the LRU prioritization approach can be competitive in a severely constrained testing environment, i.e. when it is not possible to execute all test cases selected by an RTS technique.

Mirarab and Tahvildari took a different probabilistic approach to test case prioritization using Bayesian Networks [145]. The Bayesian Network model is built upon changes in program elements, fault proneness of program elements and probability of each test case to detect faults. Mirarab and Tahvildari extended the approach by adding a feedback route to update the Bayesian Network as prioritization progresses [146]. For example, if a test case covers a set of program elements, the probability of selecting other test cases that cover the same elements will be lowered. Note that this corresponds to the 'additional' approach described by Rothermel *et al.* [115, 121].

History-based Approach: Sherriff *et al.* presented a prioritization technique based on association clusters of software artefacts obtained by a matrix analysis called singular value decomposition [147]. The prioritization approach depends on three elements: association clusters, relationship between test cases and files and a modification vector. Association clusters are generated from a change matrix using SVD; if two files are often modified together as a part of a bug fix, they will be clustered into the same association cluster. Each file is also associated with test cases that affect or execute it. Finally, a new system modification is represented as a vector in which the value indicates whether a specific file has been modified. Using the association clusters and the modification vector, it is then possible to assign each file with a priority that corresponds to how closely the new modification matches each test case. One novel aspect of this approach is that any software artefact can be considered for prioritization. Sherriff *et al.* noted that the faults that are found in non-source files, such as media files or documentation, can be as severe as those found in source code.

Requirement-based Approach: Srikanth *et al.* presented requirement-based test case prioritization [139]. Test cases are mapped to software requirements that are tested by them, and then prioritized by various properties of the mapped requirements, including customer-assigned priority and implementation complexity. One potential weakness of this approach is the fact that requirement properties are often estimated and subjective values. Krishnamoorthi and Sahaaya developed a similar approach with additional metrics [148].

Model-based Approach: Korel *et al.* introduced a model-based prioritization approach [149–151]. Their initial approach was called *selective* prioritization, which was strongly connected to RTS [149]. Test cases were classified into a high priority set, TS_H, and a low priority set, TS_L. They defined and compared different definitions of high and low priority test cases, but essentially a test case is assigned high priority if it is relevant to the modification made to the model. The initial selective prioritization process consists of the random prioritization of TS_H followed by the random prioritization of TS_L. Korel *et al.* developed more sophisticated heuristics based on the dependence analysis of the models [150, 151].

Other Approaches: The use of mutation score for test case prioritization has been analysed by Rothermel *et al.* along with other structural coverage criteria [115, 121]. Hou *et al.* considered interface-contract mutation for the regression testing of component-based software and evaluated it with the *additional* prioritization technique [152].

Sampath *et al.* presented the prioritization of test cases for web applications [153]. The test cases are, in this case, recorded user sessions from the previous version of the SUT. Session-based test cases are thought to be ideal for testing web applications because they tend to reflect the actual usage patterns of real users, thereby making for realistic test cases. They compared different criteria for prioritization such as the number of HTTP requests per test case, coverage of parameter values,

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

frequency of visits for the pages recorded in sessions and the number of parameter values. The empirical evaluations showed that prioritized test suites performed better than randomly ordered test suites, but also that there is not a single prioritization criterion that is always best. However, the 2-way parameter-value criterion, the prioritization criterion that orders tests to cover all pairwise combinations of parameter-values between pages as soon as possible, showed the highest APFD value for 2 out of 3 web applications that were studied.

Fraser and Wotawa introduced a model-based prioritization approach [154]. Their prioritization technique is based on the concept of *property relevance* [155]. A test case is relevant to a model property if it is theoretically possible for the test case to violate the property. The relevance relation is obtained by the use of a model-checker, which is used as the input to the greedy algorithm. While they showed that property-based prioritization can outperform coverage-based prioritization, they noted that the performance of property-based prioritization is heavily dependent on the quality of the model specification.

A few techniques and analyses used for test suite minimization or RTS problem have also been applied to test case prioritization. Rummel *et al.* introduced a prioritization technique based on data-flow analysis by treating each *du* pair as a testing requirement to be covered [156]. Smith *et al.* introduced a prioritization technique based on a call-tree model, which they also used for test suite minimization [26]. They prioritized test cases according to the number of call-tree paths covered by each test case. Jeffrey and Gupta prioritized test cases using relevant slices [157], which was also used for RTS [54]. Each test case was associated with output statements, from which relevant slices were calculated. Then test cases were prioritized according to the sum of two elements: the size of the corresponding relevant slice and the number of statements that are executed by the test case but do not belong to the relevant slice. Both elements were considered to correlate to the chance of revealing a fault introduced by a recent change.

5.4. Cost-aware test case prioritization

Unlike test suite minimization and RTS, the basic definition of test case prioritization does not involve filtering out test cases, i.e. it is assumed that the tester executes the entire test suite following the order given by the prioritization technique. This may not be feasible in practice due to limited resources. A number of prioritization techniques addressed this problem of the need to be cost-aware [23, 24, 118, 158].

With respect to cost-awareness, the basic APFD metric has two limitations. First, it considers all faults to be equally severe. Second, it assumes that every test case costs the same in resources. Elbaum *et al.* extended the basic APFD metric to APFD_c so that the metric incorporates not just the rate of fault detection but also the severity of detected faults and the expense of executing test cases [118]. An ordering of test cases according to the APFD_c metric detects more severe faults at a lower cost. More formally, let T be the set of n test cases with costs $t_1, \ldots, t_n$, and let F be the set of m faults with severity values $f_1, \ldots, f_m$. For ordering T', let TF_i be the order of the first test case that reveals the ith fault. APFD_c of T' is calculated as following:

$$APFD_c = \frac{\sum_{i=1}^{m} (f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i}$$

Elbaum *et al.* applied random ordering, additional statement coverage prioritization, additional function coverage prioritization and additional fault index prioritization techniques to `space`, which contains faults discovered during the development stage. They adopted several different models of test case cost and fault severity, including uniform values, random values, normally distributed values and models taken from the Mozilla open source project. The empirical results were achieved by synthetically adding cost severity models to `space`. This enabled them to observe the impact of different severity and cost models. They claimed two practical implications. With respect to test case cost, they proposed the use of many small test cases rather than a few large test cases. Clearly the number of possible prioritizations is higher with a test suite that contains many small test cases, compared to one with a small number of large test cases. It was also claimed that having different models of fault severity distribution can also impact the efficiency of testing.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

This is true only when the prioritization technique considers the fault-detection history of previous tests.

Elbaum *et al.* compared two different severity distribution models: linear and exponential. In the linear model, the severity values grow linearly as the severity of faults increases, whereas they grow exponentially in the exponential model. If the previous fault-detection history correlates to the fault detection capability of the current iteration of testing, the exponential model ensures that test cases with a history of detecting more severe faults are executed earlier.

Walcott *et al.* presented a time-aware prioritization technique [23]. Time-aware prioritization does not prioritize the entire test suite; it aims to produce a subset of test cases that are prioritized and can be executed within the given time budget. More formally, it is defined as follows:

Given: A test suite, T, the collection of all permutations of elements of the power set of permutations of T, $perms(2^T)$, the time budget, $t_{\max}$, a time function $time: perms(2^T) \rightarrow \mathbb{R}$ and a fitness function $fit: perms(2^T) \rightarrow \mathbb{R}$:

Problem: Find the test tuple $\sigma_{\max} \in perms(2^T)$ such that $time(\sigma_{\max}) \leq t_{\max}$ and $\forall \sigma' \in perms(2^T)$, where $\sigma_{\max} \neq \sigma'$ and $time(\sigma') \leq t_{\max}$, $fit(\sigma_{\max}) > fit(\sigma')$.

Intuitively, a time-aware prioritization technique selects and prioritizes test cases at the same time so that the produced ordered subset yields higher fault-detection rates within the given time budget. Walcott *et al.* utilized a genetic algorithm, combining selection and prioritization into a single fitness function. The selection component of the fitness function is given higher weighting so that it dominates the overall fitness value produced. The results of the genetic algorithm were compared to random ordering, reverse ordering and the optimal ordering. The results showed that time-aware prioritization produces higher rates of fault detection compared to random, initial and reverse ordering. However, Walcott *et al.* did not compare the time-aware prioritization to the existing, non-time-aware prioritization techniques. Note that non-time-aware prioritization techniques can also be executed in 'time-aware' manner by stopping the test when the given time budget is exhausted.

While Yoo and Harman studied test suite minimization [24], their multi-objective optimization approach is also relevant to the cross-cutting concern of cost-awareness. By using multi-objective optimization heuristics, they obtained a Pareto-frontier which represents the trade-offs between the different criteria, including cost. When there is a constraint on cost, the knowledge of Pareto-frontier can provide the tester with more information to achieve higher coverage. The tester can then prioritize the subset selected by observing the Pareto-frontier.

The cost-constraint problem has also been analysed using ILP [159, 160]. Hou *et al.* considered the cost-constraint in web service testing [159]. Users of web services are typically assigned with a usage quota; testing a system that uses web services, therefore, has to consider the remaining quota for each web service. The ILP approach was later analysed in more generic context using execution time of each test as a cost factor [160].

Do and Rothermel studied the impact of time constraints on the cost-effectiveness of existing prioritization techniques [158]. In total, six different prioritization approaches were evaluated: original order, random order, total block coverage, additional block coverage, Bayesian Network approach without feedback, Bayesian Network approach with feedback. They considered four different time constraints, each of which allows $\{25\%, 50\%, 75\%, 100\%\}$ of time required for the execution of all test cases. Each prioritization approach was evaluated under these constraints using a cost-benefit model. The results showed that, although time constraints affect techniques differently, it is always beneficial to adopt some prioritization when under time constraints. The original ordering was always affected the most severely.

6. META-EMPIRICAL STUDIES

Recently, the meta-empirical study of regression testing techniques has emerged as a separate subject in its own right. It addresses cross-cutting concerns such as cost-benefit analysis of regression testing techniques and the studies of evaluation methodology for these techniques. Both studies

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

seek to provide more confidence in efficiency and effectiveness of regression testing techniques. Work in these directions is still in the early stages compared to the bodies of work available for minimization, selection or prioritization techniques. However, it is believed that these studies will make significant contributions towards the technology transfer.

Empirical evaluation of any regression testing technique is inherently a *post-hoc* process that assumes the knowledge of a set of known faults. Without the *a priori* knowledge of faults, it would not be possible to perform a controlled experiment comparing different regression testing techniques. This poses a challenge to the empirical evaluation of techniques, since the availability of fault data tends to be limited [161].

Andrews *et al.* performed an extensive comparison between real faults and those seeded by mutation [161]. One concern when using mutation faults instead of real faults is that there is no guarantee that the detection of mutation faults can be an accurate predictor of the detection of real faults. After considering various statistical data such as the ratio and distribution of fault detection, Andrews *et al.* concluded that mutation faults can indeed provide a good indication of the fault detection capability of the test suite, assuming that mutation operators are carefully selected and equivalent mutants are removed. However, they also note that, while mutation faults were not easier to detect than real faults, they were also not harder to detect. Do and Rothermel extended this study by focusing the comparison on the result of test case prioritization techniques [162, 163]. Here, they considered whether evaluating prioritization techniques against mutation faults and seeded faults differs. Based on the comparison of these two evaluation methods, it was concluded that mutation faults can be safely used in place of real or hand-seeded faults.

Although it was not their main aim, Korel *et al.* made an important contribution to the empirical evaluation methodology of regression testing techniques through the empirical evaluation of their prioritization techniques [149–151]. They noted that, in order to compare different prioritization techniques in terms of their rate of fault detection, they need to be evaluated using all possible prioritized sequences of test cases that may be generated by each technique. Even deterministic prioritization algorithms, such as the greedy algorithm, can produce different results for the same test suite if some external factors change; for example, if the ordering of the initial test suite changes, there is a chance that the greedy algorithm will produce a different prioritization result. Korel *et al.* argued, therefore, that the rate of fault detection should be measured in average across all possible prioritized sequences. They introduced a new metric called Most Likely average Position, which measures the average relative position of the first test case that detects a specific fault.

Elbaum *et al.* extended the empirical studies of prioritization techniques with the Siemens suite and `space` [116] by performing statistical analysis of the variance in APFD [117]. The APFD values were analysed against various program, change and test metrics. Program metrics included mean number of executable statements, mean function size across all functions, etc. Change metrics included number of functions with at least one changed statement, number of statements inserted or deleted, etc. Test metrics included number of tests in the test suite, percentage of tests reaching a changed function, mean number of changed functions executed by a test over a test suite, etc. The aim was to identify the source of variations in results. Elbaum *et al.* reported that the metrics that reflected normalized program characteristics (such as mean function size across the program) and characteristics of test suites in relation to programs (such as mean percentage of functions executed by a test over a test suite) were the primary contributors to the variances in prioritization. While they reported that this finding was not the anticipated one, it showed that the prioritization results are the product of closely coupled interactions between programs under test, changes and test suites.

Empirical evaluation of different techniques can benefit from a shared evaluation framework. Rothermel and Harrold presented a comparison framework for RTS techniques [45], which was used to compare different RTS techniques [1]. While minimization and prioritization techniques lack such a framework, certain metrics have been used as a *de facto* standard evaluation framework. Rate of reduction in size and rate of reduction in fault-detection capability have been widely used to evaluate test suite minimization techniques [12, 18–21, 25, 26, 38–40, 43]. Similarly, APFD [116] has been widely used to evaluate prioritization techniques [23, 116, 117, 120, 121, 123, 127, 136–138, 145, 146, 154, 156, 157, 159, 160, 162–165].

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Rothermel *et al.* studied the impact of test suite granularity and test input grouping on the cost-effectiveness of regression testing [120, 165]. They first introduced the concept of *test grains*, which is the smallest unit of test input that is executable and checkable. Test cases are constructed by grouping test grains. Based on this, they defined test suite granularity as the number of test grains in a test case, and test input grouping as the way test grains are added to each test case, e.g. randomly or grouped by their functionality. They reported that having a coarse-grained test suite did not significantly compromise the fault-detection capability of the test suite, but resulted in decreased total execution time. The savings in execution time can be explained by the fact that a coarse-grained test suite contains fewer test cases, thereby reducing the set-up time and other overheads that occur between execution of different test cases. However, they did not consider the cost of the test oracle. It is not immediately obvious whether the cost of a test oracle would increase or decrease as the test suite granularity increases. This oracle cost could affect the overall cost-effectiveness.

Kim *et al.* studied the impact of test application frequency on the cost-effectiveness of RTS techniques [166, 167]. Their empirical studies showed that the frequency of regression test application has a significant impact on the cost-effectiveness of RTS techniques. They reported that RTS techniques tend to be more cost-effective when the frequency of test application is high. It implies that only a small amount of changes are made between tests, which makes RTS more effective. However, as intervals between tests grow, changes are accumulated and RTS techniques tend to select more and more test cases, resulting in low cost-effectiveness. One interesting finding is that, as intervals between tests grow, random re-testing tends to work very well. With small testing intervals, the random approach fails to focus on the modification. As testing intervals increase, more parts of SUT need to be re-tested, improving the effectiveness of the random approach. Elbaum *et al.* studied the impacts of changes in terms of the quantitative nature of modifications [164]. They investigated how the cost-effectiveness of selection and prioritization techniques is affected by various change metrics such as percentage of changed lines of code (LOC), average number of LOC changed per function, etc. Their empirical analysis confirmed that the differences in these metrics can make a significant impact on the cost-effectiveness of techniques. However, they also reported that simple size of change, measured in LOC, was not a predominant factor in determining the cost-effectiveness of techniques. Rather, it was the distribution of changes and the ability of test cases to reach these changes.

Elbaum *et al.* also presented a technique for selecting the most cost-effective prioritization technique [168]. They applied a set of prioritization techniques to the same set of programs, and analysed the resulting APFD metric values. Different techniques perform best for different programs; they applied the classification tree technique to predict the best-suited technique for a program. Note that the term 'cost-effectiveness' in this work means the efficiency of a prioritization technique measured by the APFD metric; the computational cost of applying these techniques was not considered.

Rosenblum and Weyuker introduced a coverage-based cost-effective predictor for RTS techniques [169]. Their analysis is based on the coverage relation between test cases and program entities. If each program entity has a uniformly distributed probability of being changed in the next version, it is possible to predict the average number of test cases to be selected by a safe RTS technique using coverage relation information. They evaluated their predictor with the TestTube RTS tool [62], using multiple versions of the `KornShell` [170] and an I/O library for Unix, `SFIO` [171], as subjects. Their predictor was reasonably accurate; for example, it predicted an average of 87.3% of the test suite to be selected for `KornShell`, when TestTube selected 88.1%. However, according to the cost model of Leung and White [172], the cost of coverage analysis for RTS per test case was greater than the cost of execution per test case, indicating that TestTube was not cost-effective. Harrold *et al.* introduced an improved version of the cost-effective predictor of Rosenblum *et al.* for more accurate cost-effectiveness prediction of version-specific RTS [173]. They evaluated their predictor using TestTube and another RTS tool, DejaVu [57].

Modelling the cost-effectiveness of regression testing techniques has emerged as a research topic. This is motivated by the observation that any analysis of cost-effectiveness should depend

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

on some model. Leung and White introduced an early cost-model for regression testing strategies and compared the cost models of the *retest-all* strategy and the selective retesting strategy [172]. Malishevsky *et al.* presented detailed models of cost-benefit trade-offs for regression testing techniques [119]. They applied their models to the regression testing of `bash`, a popular Unix shell [174], with different ratio values of $f/(e+c)$, where f is the cost of omitting one fault, e is the additional cost per test and c is the result-validation cost per test. The results implied that if a regression testing technique does not consider f, it may overestimate the cost-effectiveness of a given technique. The cost model of Malichevsky *et al.* has been extended and evaluated against the prioritization of JUnit test cases [175]. Smith and Kapfhammer studied the impact of the incorporation of cost into test suite minimization [176]. Existing minimization heuristics including HGS [12], delayed greedy [18] and 2-optimal greedy algorithm [128] were extended to incorporate the execution cost of each test case. Do and Rothermel considered the impact of time constraints on selection and prioritization techniques across multiple consecutive versions of subject programs to incorporate software life-cycle factors into the study [177].

Reflecting the complexity of regression testing process, cost-effectiveness models often need to be sophisticated in order to incorporate multiple variables [119, 175–177]. However, complexity can be a barrier to uptake. Do and Rothermel introduced an approach based on statistical sensitivity analysis to simplify complicated cost models [178]. Their approach fixed certain cost factors that were deemed to be the least significant by the sensitivity analysis. The empirical evaluation showed that, while certain levels of simplification can still preserve the accuracy of the model, over-simplification may be risky.

7. SUMMARY & DISCUSSION

7.1. Analysis of current global trends in the literature

This paper has produced a survey of 159 papers on test suite minimization, RTS and test case prioritization. This number includes papers on methodologies of empirical evaluation and comparative studies. Data summarizing the results in these papers are shown in Tables AI–AIV in the Appendix. Note that the last category consists of papers on cross-cutting concerns for empirical studies, such as methodologies of empirical evaluation and analyses of cost-effectiveness, as well as purely comparative studies and surveys. Figure 2 plots the number of surveyed papers for each year since 1977, when Fischer published his paper on RTS using a linear programming approach [47]. The observed trend in the number of publications shows that the field continues to grow.

Figure 3 shows the chronological trend in the number of studies for each of the topics in this paper. In this figure, the papers have been classified into four different categories. The first three

Figure 2. Number of surveyed papers in each year since 1977. The field is still growing.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Figure 3. Relative research interest in each subject. Papers that consider more than one subject were counted multiple times.

categories contain papers on minimization, selection and prioritization, respectively. The fourth category contains papers on empirical evaluation and comparative studies, including previous surveys. Papers that consider more than one subject are represented in each category for which they are relevant; for example, a survey on RTS [1] is counted in both the selection category and the comparative studies category. Therefore, while the graph closely resembles Figure 2, it is not a representation of the number of publications. Rather, the figure should be read as a guide to the trends in study topics over time.

Considering that many different RTS approaches were introduced in the late 80s and 90s, recent research on RTS techniques has been mostly concerned with the application and evaluation of the graph-walk approach [83, 84, 87, 90–95]. As the figure reveals, the interest in test case prioritization has been steadily growing since the late 90s. In Figures 2 and 3, the data for 2009 is, of course, partial.

Whereas most of the early papers on RTS were theoretical (Table AII), empirical evaluation of regression testing techniques has recently received a burgeoning interest. Not only are there more publications on pure empirical/comparative studies (as can be observed in Figure 3), but recent studies of regression testing techniques tend to evaluate the suggested techniques empirically, as can be observed in Tables AI–AIII.

However, the scale of empirical studies seems to remain limited. Figure 4 shows the maximum size of SUTs (measured in LOC) and test suites (measured as the number of test cases) studied empirically in the literature. For both data, only the empirical studies that explicitly note the size of subject SUTs and test suites have been included. When only the average size of test suites is given, the maximum average size of studied test suites has been used. For the maximum size of SUTs, only the empirical studies that use source code as test subjects have been included; for example, studies of regression testing of UML models are not included. For about 60% of empirical studies, the largest SUT studied is smaller than 10 000 LoC. For about 70% of empirical studies, the largest test suite studied contains fewer than 1000 test cases.

Figure 5 shows the origins of subject SUTs studied in the literature. For detailed information about the classification, refer to the Appendix. Programs available from the Software Infrastructure Repository (SIR) [179] account for over 50% of subjects of empirical studies of regression testing techniques. The predominant programming language is C, followed by Java. Considering that the first paper appeared in 2002, model-based techniques have shown significant growth. Although there are several papers on RTS for web applications [91–95], empirical evaluation of these techniques remains limited.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Figure 4. Maximum size of SUT and test suites studied in the literature.

Figure 5. Origins of subject SUTs observed in the literature.

7.2. *State-of-the-art, trends and issues*

7.2.1. State-of-the-art Among the class of RTS techniques, the graph-walk approach seems to be the predominant technique in the literature. Although it was originally conceived for CDGs and PDGs [46, 56], the most widely used form of graph-walk approach works on CFGs [57, 79]. Its popularity can be observed from the fact that the approach has been applied to many forms of graph representation of SUT that are not CFGs [82–84, 87]. Indeed, the strength of the graph-walk approach lies not only in the fact that it is intuitive but also in the generic applicability of the technique to any graph representation of systems.

By studying the literature, it becomes clear that two ideas played essential roles in the development of RTS techniques: Leung and White's early idea of regression testing and test case classification [10], and Rothermel and Harrold's definition of a *safe* RTS [45]. Collectively, these two ideas provided a profound theoretical framework that can be used to evaluate RTS techniques.

The existence of such a theoretical framework is what differentiate RTS techniques from test suite minimization and test case prioritization. For RTS techniques, it is possible to define what a *safe* technique should do because the RTS problem is specifically focused on the modifications between two versions of SUT. Minimization and prioritization techniques, on the other hand, are forced to rely on surrogate metrics for real fault-detection capability. Therefore, it may not be clear what 'safe' minimization or prioritization techniques would mean. A minimization technique would not be safe unless the surrogate metric perfectly captures the fault-detection capability of the test suite; however, the empirical studies so far have shown that there is no such single metric.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

For prioritization, the concept of 'safe' and 'non-safe' does not apply directly since the aim of the technique is to permute the test suite and not to select a subset out of it.

Naturally, the history of minimization and prioritization literature is an on-going exploration of different heuristics and surrogate metrics. It is interesting to note that the greedy algorithm, a good approximation for the set cover problem and, therefore, test suite minimization problem, is also an efficient heuristic for test case prioritization precisely because of its *greedy* nature. In other words, *as much as possible*, *as soon as possible*. As a result of this, the greedy approach and its variations have a strong presence in the literature on both test suite minimization [11, 14, 18] and test case prioritization [115, 116, 121]. Recently, there are other approaches to test suite minimization and test case prioritization that aim to overcome the uncertainty of surrogates, e.g. the use of multiple minimization criteria [21, 22, 24] and the use of expert knowledge for test case prioritization [137, 138].

7.2.2. Trends Emphasis on Models: Whereas most of the early regression testing techniques concerned code-based, white-box regression testing, the model-based regression testing approach has been of more recent growing interest [32–34, 68, 69, 88, 107–111]. UML models and EFSMs are often used.

New Domains: While the majority of existing literature on regression testing techniques concerns stand-alone programs written in C and Java, there are a growing number of other domains that are being considered for regression testing. For example, spreadsheets [76], GUIs [25, 180] and web applications [91–95, 153] have been considered.

Multi-criteria Regression Testing: In minimization and prioritization, it is known that there is no single surrogate metric that correlates to fault prediction capability for all programs [8, 38–40, 115, 121]. One potential way of overcoming this limitation is to consider multiple surrogates simultaneously [21, 22] using classical multi-objective optimization techniques, such as the weighted-sum approach [21] or the prioritized approach [22]. Expert domain knowledge has also been used in addition to software metrics for test case prioritization [137, 138]. Finally, Multi-Objective Evolutionary Algorithms (MOEAs) have been used to deal with multiple objectives [24].

Cost-awareness: The cost of regression testing is a cross-cutting concern for all three classes of regression testing techniques. Studies of cost-aware regression testing can be classified into two categories. First, there is work that aims to incorporate the cost of testing directly into regression testing techniques at technical level; for example, cost-aware test case prioritization [23, 118] or minimization techniques that provide the tester with a series of alternative subsets of the original test suites that can be executed in different amounts of time [24]. Second, there are empirical evaluations of regression testing techniques that consider whether the application of these techniques is indeed cost-effective in the wider context of the overall software life-cycle [120, 158, 165, 169, 177].

7.2.3. Issues Limited Subjects: In Figure 5, subject programs from SIR [179] account for almost 60% of the subjects of empirical studies observed in the regression testing technique literature. While this is certainly evidence that SIR has been of tremendous value to the research community, it also means that many regression testing techniques are being evaluated against a limited set of programs and test suites. By no means is this a criticism of SIR itself or work that is based on subjects from SIR; rather, the dependency on SIR shows how time consuming and difficult it is to collect multiple versions of program source code, their test suites and associated fault information, let alone to make it publicly available. The SIR commonality also supports and facilitates comparisons. However, the community faces a possible risk of 'over-fitting' the research of these techniques to those programs that are easily available. Open source software projects are often suggested as an alternative source of data, but from the results of this survey analysis it seems that their use for research on regression testing techniques is limited.

Two potential approaches to the issue of limited subjects can be envisioned. The first approach is to design a method that will allow a realistic simulation of real software faults. There is early work in this direction. Andrews *et al.* considered whether mutation faults are similar to real faults [161];

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Do and Rothermel studied the same question especially in the context of regression testing [163]. Jia and Harman used a search-based approach to obtain higher-order mutation faults that are more 'subtle' and, therefore, potentially harder to detect than first-order mutation faults [181].

The second approach is to engage more actively with industry and open source communities. This is not an easy task, because the information about testing and software faults tends to be sensitive, particularly to commercial organizations. The research community faces the challenge of convincing the wider practitioner community of the cost-effectiveness and usefulness of these techniques. This is closely related to the issue of technology transfer, to which the paper now turns.

Technology Transfer: A detailed observation of the literature suggests that the community may have reached a stage of maturity that, in order to progress to the next level of achievement, technology transfer to industry will play an important role. While the research community may not have found the 'silver bullet', most empirical evaluation of the proposed techniques suggests that application of minimization, selection and prioritization techniques does make a difference from (1) uncontrolled regression testing, i.e. retest-all approach and un-prioritized regression testing, and (2) random approaches.

However, empirical evaluation and application of regression testing techniques at industrial level seems to remain limited [182]. Out of the 159 papers listed in Tables AI–AIV, only 31 papers list a member of industry as an author or a co-author. More importantly, only 12 papers consider industrial software artefacts as a subject of the associated empirical studies [68, 69, 88, 99–102, 104–106, 126, 147]. This suggests that a large-scale industrial uptake of these techniques has yet to occur.

8. FUTURE DIRECTIONS

This section discusses some of the possible future directions in the field of regression testing techniques. While it is not possible to predict the future direction that a field of study will follow, it was possible to identify some trends in literature, which may suggest and guide the direction of future research.

8.1. Orchestrating regression testing techniques with test data generation

Automatic test data generation has made advances in both functional and non-functional testing [183]. Superficially test data generation is the counterpart to regression testing techniques; it creates test cases while regression testing seeks to mange them. However, *because* these two activities are typically located at the opposite ends of the testing process, they may become close when the testing process repeats.

Orchestrating regression testing with test data generation has a long heritage. From the classification of test cases by Leung and White [10], it follows that regression testing involves two different needs that are closely related to test data generation: repairing obsolete test cases for corrective regression testing and generating additional test data for progressive regression testing. The second problem, in particular, has been referred to as the test suite augmentation problem [184, 185]. There was early work on both problems. Memon and Soffa considered automatic repair of GUI test cases that were made obsolete by the changes in the GUI [180], which was later extended by Memon [186]. Similarly, Alshahwan and Harman focused on the use of user session data for regression testing of web applications, and how the test cases can be automatically repaired for the next version [187]. Apiwattanapong *et al.* identified the testing requirements that are needed to test the new and modified parts of an SUT so that additional test data can be generated automatically [184, 185].

However, it is believed that there are additional areas that may be synergetic. For example, test data generation might possibly refer to the test cases selected and unselected during the last iteration in order to identify the part of the SUT to focus on. Similarly, regression testing techniques can use the additional information provided by test data generation techniques in order to make regression testing more efficient and effective. For example, there are test data generation

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

techniques that target a specific concern in the SUT, such as detection of the presence of a memory leak. The additional information about the *intention* behind each test case could be used to enrich the minimization, selection and prioritization process.

8.2. Multi-objective regression testing

Regression testing is a complex and costly process that may involve multiple objectives and constraints. For example, the cost of executing a test case is usually measured as the time taken to execute the test case. However, there may be a series of different costs involved in executing a test case, such as setting up the environment or preparing a test input, each of which may be subject to a different constraint. Existing techniques also assume that test cases can be executed in any given order without any change to the cost of execution, which seems unrealistic. Test cases may have dependency relations between them. It may also be possible to lower the cost of execution by grouping test cases that share the same test environment, thereby saving set-up time.

Considering the complexity of real-world regression testing, existing representations of problems in regression testing may be over-simplistic. Indeed, most of the published empirical studies rely on relatively small-scale academic examples. Even when real-world programs are studied, they tend to be individual programs, not a software system as a whole. Larger software systems do not simply entail larger problem size; they may denote a different level of complexity.

It is believed that, in order to cope with the complexity of regression testing, regression testing techniques may need to become multi-objective. There is existing, preliminary work that seeks to represent the cost of test case execution as an additional constraint using evolutionary algorithms [23, 24]. Multi-objective optimization heuristics may provide the much-needed flexibility that is required for the representation of problems with high complexity.

Another benefit of moving to a multi-objective paradigm is the fact that it provides additional insight into the regression testing problem by allowing the testers to observe the inherent trade-offs between multiple constraints. This is not possible with the so-called *classical* approaches to multi-objective problems that either consider one objective at a time [22] or conflate multiple objectives into a single objective using weighting [21, 23]; these approaches may be based on *multiple criteria*, but they are not truly multi-objective in the sense that they all produce a single solution. The result of a multi-objective optimization is often a set of solutions that do not dominate each other, thereby forming the trade-offs between constraints. The insight into the trade-offs may provide additional information that is hard to obtain manually.

8.3. Problem of test oracle and its cost

Test oracles present a set of challenging problems for software testing. It is difficult to generate them automatically, they often require human efforts to verify and the cost of this effort is hard to estimate and measure. The oracle cost has been considered as a part of cost models [178], but has not been considered as a part of the process of minimization, selection and prioritization itself. Since regression testing techniques seek to efficiently re-use existing test cases, information about the cost of verifying the output observed with the existing test suite may be collected across versions. This can be incorporated into the existing regression testing techniques.

While a test oracle and its cost may be seen as yet another additional objective that can be considered using a multi-objective approach, it can be argued that the test oracle will present many interesting and exciting research questions in the context of regression testing and, thus, deserves a special treatment in its own right. This is because, compared to other testing cost such as the physical execution time of test cases, the test oracle cost is closely related to the *quality* of testing. Moreover, unlike some costs that can be reduced by using more advanced hardware, the cost of oracle verification derives from human effort and is, therefore, harder to reduce. These characteristics make the issues related to test oracles challenging but interesting research subjects.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

8.4. Consideration of other domains

The majority of regression testing techniques studied in this survey concern white-box structural regression testing of code-level software artefacts. However, other domains are emerging as new and exciting research subjects.

Recently, Service-Oriented Architectures (SOAs) have been of keen interest both for academic researchers and industrialists. In the SOA paradigm, software system is built, or *composed*, by orchestrating a set of web services, each of which takes charge of a specific task. Several approaches have been introduced to address the issue of regression testing of web services, most of which seek to apply the same technique developed for traditional applications to web services [91–95]. However, the inherently distributed nature of an SOA system presents several challenges that are alien to traditional regression testing techniques.

Web services often reside in remote locations and are developed by a third-party, making it hard to apply the traditional white-box regression testing techniques that require analysis of source code. Modifications can happen across multiple services, which can make fault localization difficult. High interactivity in web applications may result in complex test cases that may involve human interaction. Finally, distributed systems often contain concurrency issues. Traditional regression testing techniques assume that the program produces deterministic output. This may not be adequate for testing applications with concurrency. Answers to these specific issues in regression testing of web applications are still in the early stage of development.

Model-based regression testing techniques have also received growing interests [68, 69, 107–111]. It is believed that the model-based regression testing techniques will be of crucial importance in the future for the following reasons:

- *Higher level regression testing*: These techniques can act as a medium between requirement/specification and testing activities, bringing regression testing from the structural level to functional level.
- *Scalability*: in dealing with software systems of industrial scale, model-based techniques will scale up better than code-based techniques.

However, there are a few open research questions. First, there is the well-known issue of traceability. Unless the traceability from requirements and specifications to code-level artefacts and test cases is provided, the role of model-based regression testing techniques will be severely limited. Second, there is the issue of test adequacy: if a test adequacy A is appropriate for a model M, which test adequacy should be used to test the program P that has been automatically generated from M? Does A still apply to P? If so, does it follow that M being adequate for A means P will be adequate for A as well?

There are also other interesting domains to consider. Testing of GUIs has received growing interest, not only in the context of regression testing [25, 135, 180], but also in the context of testing in general [188, 189]. Regression testing GUIs present a different set of challenges to code-based structural regression testing since GUIs are usually generated in a visual programming environment; they are often subject to frequent changes and, not being well-typed, do not readily facilitate static analysis.

8.5. Non-functional testing

A majority of existing regression testing techniques rely upon structural information about the SUT, such as data-flow analysis, CFG analysis, program slices and structural coverage. The impact that non-functional property testing will have on regression testing techniques has not been fully studied. Existing techniques were able to map the problems in the regression testing to well-formed abstract problems using the properties of structural information. For example, test suite minimization could be mapped to the minimal hitting set problem or the set coverage problem, precisely because the techniques were based on the concept of 'coverage'. Similarly, graph-walking approaches to test case selection were made possible because the changes between different versions were defined by structural differences in CFGs.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Imagine regression testing techniques for non-functional properties. What would be the minimized test suite that can test the power consumption of an embedded system? How would test cases be prioritized to achieve an efficient and effective stress testing of a web application? These questions remain largely unanswered and may require approaches that are significantly different from existing paradigms.

8.6. Tool support

Closely related to the issue of technology transfer is the issue of tool support. Without readily available tools that implement regression testing techniques, practical adoption will remain limited. One potential difficulty of providing tool support is the fact that, unlike unit testing for which there exists a series of frameworks based on the xUnit architecture, there is not a common framework for the regression testing process in general. The closest to a common ground for regression testing would be an Integrated Development Environment (IDE), such as Eclipse, with which the xUnit architecture is already integrated successfully. A good starting point for regression testing techniques may be the management framework of unit test cases, built upon xUnit architecture and IDEs.

9. CONCLUSION

This paper provides both a survey and a detailed analysis of trends in regression test case selection, minimization and prioritization. The paper shows how the work on these three topics is closely related and provides a survey of the landscape of work on the development of these ideas, their applications, empirical evaluation and open problems for future work.

The analysis of trends reported in the paper reveals some interesting properties. There is evidence to suggest that the topic of test case prioritization is of increasing importance, judging by the shift in emphasis towards it that is evident in the literature. It is also clear that the research community is moving towards assessment of the complex trade-offs and balances between different concerns, with an increase in work that considers the best way in which to incorporate multiple concerns (cost and value for instance) and to fully evaluate regression testing improvement techniques.

This focus on empirical methodology is one tentative sign that the field is beginning to mature. The trend analysis also indicates a rising profile of publication, providing evidence to support the claim that the field continues to attract growing attention from the wider research community, which is a positive finding for those working on regression test case selection and minimisation and, in particular those working on prioritization problems.

Our survey also provides evidence to indicate that there is a preponderance of empirical work that draws upon a comparatively small set of subjects (notably those available through the SIR repository). This is a testament to the importance of this source of case study material. It is valuable because it allows for cross comparison of results and replication, which is essential for the development of any science. However, it may potentially suggest a risk of over-fitting.

APPENDIX A

The following tables contain detailed information about publications on each class of technique: test suite minimization in Table AI, RTS in Table AII, test case prioritization in Table AIII and empirical/comparative studies in Table AIV. Note that publications that consider more than one class of techniques appear in multiple tables for ease of reference. For example, a survey of Regression Test Selection [1] appears in both Tables AII and AIV.

Information on the maximum size of SUT (measured in LOC) and the maximum size of test suites were collected from papers only when they were explicitly noted. Some papers that considered multiple test suites for a single program contained only the average, in which case the size of the test suite with largest average size was recorded. When the studied SUT cannot be measured in

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table AI. Summary of publications on test suite minimization.

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
Horgan *et al.* [13]	1992	1000	26					•			
Harrold *et al.* [12]	1993	N/A	19					•			
Offutt *et al.* [14]	1995	48	36.8					•			
Chen *et al.* [11]	1996	N/A*	—								
Rothermel *et al.* [38]	1998	516	260	•							
Wong *et al.* [39]	1998	842	33					•			
Wong *et al.* [40]	1999	6218	200		•						
Schroeder and Korel [31]	2000	Theory	—								
Korel *et al.* [33]	2002	Theory	—								
Malishevsky *et al.* [119]	2002	65 632	1168			•					
Rothermel *et al.* [120]	2002	68 782	1985			•		•			
Rothermel *et al.* [8]	2002	516	260	•							
Vaysburg *et al.* [32]	2002	Theory	—								
Anido *et al.* [35]	2003	Theory	—								
Harder *et al.* [27]	2003	6218	169	•	•						
Marré and Bertolino [17]	2003	516	5542	•							
Black *et al.* [21]	2004	512	5542	•							
Rothermel *et al.* [165]	2004	68 782	1985			•		•			
Jeffrey and Gupta [19]	2005	516	135	•							
McMaster and Memon [43]	2005	6218	4712		•						
Tallam anf Gupta [18]	2006	6218	539	•	•						
Chen *et al.* [34]	2007	Theory	—								
Hou *et al.* [152]	2007	5500	183						•		
Jeffrey and Gupta [20]	2007	6218	1560	•	•						
Leitner *et al.* [29]	2007	N/A†	—								
McMaster and Memon [42]	2007	11 803	1500						•		
Smith *et al.* [26]	2007	1455	N/A						•		
Yoo *et al.* [24]	2007	6218	169	•	•						

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table AI. *Continued.*

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
McMaster and Memon [25]	2008	11 803	1500						•		
Yu *et al.* [44]	2008	6218	13 585	•	•						
Zhong *et al.* [6]	2008	26 824	N/A	•				•			
Hsu *et al.* [22]	2009	1 892 226	5542	•		•			•		
Kaminski *et al.* [36]	2009	N/A‡	—								
Smith and Kapfhammer [176]	2009	6822	110						•		

*Chen *et al.* [11][1] evaluated their heuristics using simulation rather than real data.

†Leitner *et al.* [29][2] minimized the length of a unit test case, not a test suite.

‡Kaminski *et al.* [36][3] applied logical reduction to a set of 19 boolean predicates taken from the avionic software.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table AII. Summary of publications on Regression Test Selection (RTS).

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
Fischer [47]	1977	Theory	—								
Fischer *et al.* [48]	1981	Theory	—								
Yau and Kishimoto [53]	1987	Theory	—								
Benedusi *et al.* [61]	1988	Theory	—								
Harrold and Soffa [50]	1988	Theory	—								
Harrold and Soffa [74]	1989	Theory	—								
Harrold and Soffa [51]	1989	Theory	—								
Hartmann and Robson [71]	1989	Theory	—								
Leung and White [10]	1989	Theory	—								
Taha *et al.* [52]	1989	Theory	—								
Hartmann and Robson [73]	1990	Theory	—								
Hartmann and Robson [72]	1990	Theory	—								
Lee and He [70]	1990	Theory	—								
Leung and White [63]	1990	Theory	—								
Leung and White [64]	1990	550	235							•	
Horgan and London [15]	1991	1000	26					•			
Gupta *et al.* [49]	1992	Theory	—								
Laski and Zermer [67]	1992	Theory	—								
White and Leung [65]	1992	Theory	—								
Agrawal *et al.* [54]	1993	Theory	—								
Bates Horwitz [60]	1993	Theory	—								
Rothermel and Harrold [46]	1993	Theory	—								
White *et al.* [66]	1993	Theory	—								
Chen *et al.* [62]	1994	11 000	39					•			
Rothermel and Harrold [56]	1994	Theory	—								
Rothermel and Harrold [45]	1994	Theory	—								
Binkley [96]	1995	Theory	—								
Kung *et al.* [98]	1995	N/A*	N/A								
Rothermel and Harrold [1]	1996	Survey	—								
Rothermel [55]	1996	516	5542	•				•			

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table AII. *Continued.*

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
Binkley [97]	1997	623	—†					•			
Rosenblum and Rothermel [5]	1997	49 316	1033	•				•			
Rosenblum and Weyuker [169]	1997	N/A‡	N/A								
Rothermel and Harrold [57]	1997	516	5542	•							
Rothermel and Harrold [78]	1997	512	5542	•							
Wong *et al.* [75]	1997	6218	1000		•						
Vokolos and Frankl [58]	1997	N/A§	N/A					•			
Ball [80]	1998	Theory	—								
Graves *et al.* [7]	1998	516	398	•							
Rothermel and Harrold [79]	1998	516	5542	•							
Vokolos and Frankl [59]	1998	6218	100		•						
Harrold [114]	1999	Theory	—								
Kim *et al.* [166]	2000	6218	4361	•	•						
Le Traon *et al.* [110]	2000	N/A ¶	N/A								
Rothermel *et al.* [81]	2000	24 849	317					•			
Beydeda and Gruhn [85]	2001	Theory	—								
Bible *et al.* [4]	2001	49 316	1033	•	•			•			
Harrold *et al.* [82]	2001	N/A	189				•		•		
Harrold *et al.* [173]	2001	516	19	•							
Jones and Harrold [124]	2001	6218	4712	•	•						
Orso *et al.* [89]	2001	6035	138				•				
Briand *et al.* [68]	2002	N/A‖	596							•	
Chen *et al.* [88]	2002	N/A	306						•		
Fisher II *et al.* [76]	2002	N/A**	493							•	
Malishevsky *et al.* [119]	2002	65 632	1168			•					
Rothermel *et al.* [120]	2002	68 782	1985			•		•			
Elbaum *et al.* [164]	2003	65 632	1168			•					
Wu and Offutt [111]	2003	Theory	—								
White *et al.* [101]	2003	N/A††	N/A							•	
Deng *et al.* [107]	2004	Theory	—								
Rothermel *et al.* [165]	2004	68 782	1985			•		•			

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table AII. *Continued.*

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/C++/C#	Other Java programs	Models and other languages	Web applications
Orso *et al.* [86]	2004	532 000	707						•		
White and Robinson [99]	2004	N/A[††]	N/A						•		
Kim *et al.* [167]	2005	6218	4361	•	•						
Martins and Vieira [87]	2005	902	N/A					•			
Muccini *et al.* [112]	2005	N/A[‖]								•	
Skoglund and Runeson [106]	2005	1 200 000	N/A						•		
Do and Rothermel [177]	2006	80 400	1533				•				
Lin *et al.* [91]	2006	N/A[‡‡]	N/A								
Pilskalns *et al.* [108]	2006	N/A	52							•	
Tarhini *et al.* [94]	2006	Theory	—								
Zhao *et al.* [84]	2006	Theory	—								
Zheng *et al.* [102]	2006	757 000	592					•			
Zheng *et al.* [103]	2006	757 000	592					•			
Muccini *et al.* [113]	2006	N/A[‖]	N/A							•	
Farooq *et al.* [109]	2007	Theory	—								
Orso *et al.* [90]	2007	6035	567				•			•	
Ruth and Tu [92]	2007	Theory	—								
Ruth *et al.* [95]	2007	Theory	—								
Ruth and Tu [93]	2007	Theory	—								
Sherriff *et al.* [147]	2007	Theory	—								
Xu and Rountev [83]	2007	3423	63						•		

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table AII. *Continued*.

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
Zheng *et al.* [104]	2007	757 000	592					•			
Zheng *et al.* [105]	2007	757 000	31					•			
Fahad and Nadeem [3]	2008	Survey	—								
White *et al.* [100]	2008	Over 1MLoC	N/A					•			
Briand *et al.* [69]	2009	N/A‖	323 614							•	

*Kung *et al.* applied their technique to InterView C++ library, which contains 147 files and over 140 classes [98].

†Binkley [97] applied his technique to five C programs (the largest of which had 623 statements), but only identified the statements that require retesting without considering test suites.

‡Rosenblum and Weyuker [169] evaluated their cost-effectiveness predictor using 31 versions of the `KornShell` and a single version of the `SFIO` (Unix library), but exact versions, sizes of SUT and sizes of test suites were not specified.

§Vokolos and Frankl evaluated their textual difference selection technique using a small C function in addmon family of tools, `power` [58].

¶Le Traon *et al.* [110] presented a case study of a model of packet-switched data transport service, the size of which was not specified.

‖Briand *et al.* studied UML models rather than real systems, the biggest of which contained either 9 classes with 70 methods [68] or 32 classes with 64 methods [69]. Muccini *et al.* studied an RTS technique at software architecture level and presented case studies for the architecture model of an elevator system and a cargo router system [112, 113].

**Fisher II *et al.* [76] evaluated their retesting strategy for spreadsheets with a spreadsheet containing 48 cells, 248 expressions and 100 predicates.

††White *et al.* applied the firewall approach to GUI program with 246 GUI objects [101]. White and Robinson applied the firewall approach to a real time system developed by ABB [99].

‡‡ Lin *et al.* [91] applied their technique to a Java Interclass Graph (JIG) with over 100 nodes.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table AIII. Summary of publications on test case prioritization.

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	space	SIR Unix utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
Harrold [114]	1999	Theory	—								
Rothermel *et al.* [115]	1999	516	19	•							
Elbaum *et al.* [116]	2000	6218	169	•	•						
Elbaum *et al.* [117]	2001	6218	169	•	•						
Elbaum *et al.* [118]	2001	6218	166		•						
Jones *et al.* [124]	2001	6218	4712	•	•						
Rothermel *et al.* [121]	2001	6218	169	•	•						
Elbaum *et al.* [123]	2002	6218	169	•	•						
Kim *et al.* [144]	2002	6218	226	•	•						
Malishevsky *et al.* [119]	2002	65 632	1168			•					
Rothermel *et al.* [120]	2002	68 782	1985			•		•			
Srivastava *et al.* [126]	2002	18 000 000*	3128					•			
Elbaum *et al.* [164]	2003	65 632	1168			•					
Leon *et al.* [136]	2003	N/A†	3333					•	•		
Do *et al.* [127]	2004	80 400	877				•				
Elbaum *et al.* [168]	2004	68 000	1985.32			•		•			
Rothermel *et al.* [165]	2004	68 782	1985			•		•			
Bryce *et al.* [131]	2005	N/A‡	—								
Do *et al.* [162]	2005	80 400	877								
Korel *et al.* [149]	2005	800	980							•	
Rummel *et al.* [156]	2005	N/A§	21						•		
Srikanth *et al.* [139]	2005	2500	50						•		
Bryce *et al.* [132]	2006	N/A‡	—								
Do *et al.* [177]	2006	80 400	1533				•				
Do *et al.* [175]	2006	80 400	877				•				
Do *et al.* [163]	2006	80 400	877				•				
Jeffrey *et al.* [157]	2006	516	N/A	•							
Tonella *et al.* [137]	2006	6218	169		•						

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table AIII. *Continued.*

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
Walcott *et al.* [23]	2006	1808	53						•		
Bryce *et al.* [135]	2007	N/A‡	—								
Fraser *et al.* [154]	2007	N/A§	246								•
Hou *et al.* [152]	2007	5500	183						•		
Korel *et al.* [150]	2007	1416	1000							•	
Li *et al.* [128]	2007	11 148	4350	•	•						
Mirarab *et al.* [145]	2007	124 000	105				•				
Qu *et al.* [133]	2007	17 155	796			•					
Smith *et al.* [26]	2007	1455	N/A						•		
Do *et al.* [158]	2008	80 400	912				•				
Hou *et al.* [159]	2008	N/A¶	1000								
Korel *et al.* [151]	2008	1416	1439	•						•	
Mirarab *et al.* [146]	2008	80 400	912				•				
Qu *et al.* [134]	2008	107 992	975			•					
Sampath *et al.* [153]	2008	9401	890								
Krishnamoorthi *et al.* [148]	2009	6000	N/A						•		
Smith *et al.* [176]	2009	6822	110						•		
Yoo *et al.* [138]	2009	122 169	1061		•	•					
Zhang *et al.* [160]	2009	5361	209						•		

*Srivastava and Thiagarajan [126] considered the biggest software system so far, an office productivity application with over 18 million LoC. However, the technique took the executable binary as input, not the source code. The compiled application was 8.8 Mb in size, with a 22 Mb symbol table.

†Leon *et al.* [136] considered three compilers: `javac`, `jikes` and `gcc`. The sizes of the source code were not specified.

‡Bryce *et al.* [131, 132, 135] studied interaction coverage prioritization, for which the LoC metric is not appropriate.

§These papers considered models rather than real software systems. Rummel *et al.* [156] applied their technique to a model with 3 classes and 21 methods. Fraser *et al.* [154] did not specify the size of the studied models and test suites.

¶Hou *et al.* [159] evaluated their technique using a web application composed of 12 web services.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table AIV. Summary of publications on empirical evaluation and comparative studies.

Reference	Year	Max. SUT size (LoC)	Max. Test suite size	Siemens suite	`space`	SIR `Unix` utilities	SIR Java Programs	Other C/ C++/C#	Other Java programs	Models and other languages	Web applications
Leung *et al.* [172]	1991	Theory	—								
Rothermel *et al.* [45]	1994	Theory	—								
Rothermel *et al.* [1]	1996	Survey	—								
Rosenblum *et al.* [5]	1997	49 316	1033	•				•			
Rosenblum *et al.* [169]	1997	N/A*	N/A								
Rothermel *et al.* [78]	1997	512	5542	•							
Graves *et al.* [7]	1998	516	398	•							
Kim *et al.* [166]	2000	6218	4361	•	•						
Bible *et al.* [4]	2001	49 316	1033	•	•			•			
Harrold *et al.* [173]	2001	516	19	•							
Malishevsky *et al.* [119]	2002	65 632	1168			•					
Rothermel *et al.* [120]	2002	68 782	1985			•		•			
Elbaum *et al.* [164]	2003	65 632	1168			•					
Elbaum *et al.* [168]	2004	68 000	1985.32			•		•			
Rothermel *et al.* [165]	2004	68 782	1985			•		•			
Do *et al.* [162]	2005	80 400	877								
Do *et al.* [179]	2005	Theory	—								
Kim *et al.* [167]	2005	6218	4361	•	•						
Do *et al.* [177]	2006	80 400	1533				•				
Do *et al.* [175]	2006	80 400	877				•				
Do *et al.* [163]	2006	80 400	877				•				
Do *et al.* [158]	2008	80 400	912				•				
Do *et al.* [178]	2008	80 400	912				•				
Engström *et al.* [2]	2008	Systematic Review	—								
Fahad *et al.* [3]	2008	Survey	—								
Zhong *et al.* [6]	2008	26 824	N/A	•				•			
Smith *et al.* [176]	2009	6822	110						•		

*Rosenblum *et al.* [169] evaluated their cost-effectiveness predictor using 31 versions of the `KornShell` and a single version of the `SFIO` (Unix library), but exact versions, sizes of SUT and sizes of test suites were not specified.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

LOC, the detailed information was provided in footnotes. Tables also contain information about the origins of the studied SUT, which are classified as follows:

- *Siemens suite* [41]: All or part of the following set of C programs—`printtokens`, `printtokens2`, `schedule`, `schedule2`, `replace`, `tcas`, `totinfo`, available from SIR [179].
- `space`: An interpreter for ADL, developed by European Space Agency. Available from SIR.
- *Unix utilities in SIR*: All or part of the following set of C programs—`flex`, `grep`, `gzip`, `sed`, `vim`, `bash`, available from SIR.
- *Java programs in SIR*: All or part of the following set of Java programs—`siena`, `ant`, `jmeter`, `jtopas`, `xml-security`, `nanoxml`, available from SIR.
- *Other C/C++/C# programs*: Programs written in C/C++/C# that are not available from SIR.
- *Other Java programs*: Programs written in Java that are not available from SIR.
- *Models and programs in other languages*: models including state machines and UML diagrams. There are also a very few empirical studies that consider programs written in other languages, e.g. Pascal.
- *Web applications*: web applications and web services.

ACKNOWLEDGEMENTS

Shin Yoo is supported by the EPSRC SEBASE project (EP/D050863). Mark Harman is supported by EPSRC Grants EP/D050863, GR/S93684 & GR/T22872, by EU grant IST-33472 (EvoTest) and also by the kind support of DaimlerChrysler Berlin and Vizuri Ltd., London.

REFERENCES

1. Rothermel G, Harrold MJ. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering* 1996; **22**(8):529–551.
2. Emelie EE, Skoglund M, Runeson P. Empirical evaluations of regression test selection techniques: A systematic review. *ESEM '08*: *Proceedings of the Second ACM—IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM: New York, NY, U.S.A., 2008; 22–31.
3. Fahad M, Nadeem A. A survey of UML based regression testing. *Intelligent Information Processing* 2008; **288**:200–210.
4. Bible J, Rothermel G, Rosenblum DS. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Transactions on Software Engineering and Methodology* 2001; **10**(2):149–183.
5. Rosenblum D, Rothermel G. A comparative study of regression test selection techniques. *Proceedings of the 2nd International Workshop on Empirical Studies of Software Maintenance*. IEEE Computer Society Press: Silver Spring, MD, 1997; 89–94.
6. Zhong H, Zhang L, Mei H. An experimental study of four typical test suite reduction techniques. *Information and Software Technology* 2008; **50**(6):534–546.
7. Graves T, Harrold MJ, Kim JM, Porter A, Rothermel G. An empirical study of regression test selection techniques. *Proceedings of the 20th International Conference on Software Engineering* (*ICSE 1998*). IEEE Computer Society Press: Silver Spring, MD, 1998; 188–197.
8. Rothermel G, Harrold M, Ronne J, Hong C. Empirical studies of test suite reduction. *Software Testing, Verification, and Reliability* 2002; **4**(2):219–249.
9. Garey MR, Johnson DS. *Computers and Intractability*: *A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company: New York, NY, 1979.
10. Leung HKN, White L. Insight into regression testing. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1989*). IEEE Computer Society Press: Silver Spring, MD, 1989; 60–69.
11. Chen TY, Lau MF. Dividing strategies for the optimization of a test suite. *Information Processing Letters* 1996; **60**(3):135–141.
12. Harrold MJ, Gupta R, Soffa ML. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology* 1993; **2**(3):270–285.
13. Horgan J, London S. ATAC: A data flow coverage testing tool for c. *Proceedings of the Symposium on Assessment of Quality Software Development Tools*. IEEE Computer Society Press: Silver Spring, MD, 1992; 2–10.
14. Offutt J, Pan J, Voas J. Procedures for reducing the size of coverage-based test sets. *Proceedings of the 12th International Conference on Testing Computer Software*. ACM Press: New York, 1995; 111–123.

Copyright © 2010 John Wiley & Sons, Ltd.

15. Horgan JR, London S. Data flow coverage and the C language. *Proceedings of the Symposium on Testing, Analysis, and Verification* (*TAV4*). ACM Press: New York, 1991; 87–97.
16. Papadimitriou CH, Steiglitz K. *Combinatorial Optimization*. Courier Dover Publications: Mineola, NY, 1998.
17. Marré M, Bertolino A. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering* 2003; **29**(11):974–984.
18. Tallam S, Gupta N. A concept analysis inspired greedy algorithm for test suite minimization. *SIGSOFT Software Engineering Notes* 2006; **31**(1):35–42.
19. Jeffrey D, Gupta N. Test suite reduction with selective redundancy. *Proceedings of the 21st IEEE International Conference on Software Maintenance 2005* (*ICSM'05*). IEEE Computer Society Press: Silver Spring, MD, 2005; 549–558.
20. Jeffrey D, Gupta N. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering* 2007; **33**(2):108–123.
21. Black J, Melachrinoudis E, Kaeli D. Bi-criteria models for all-uses test suite reduction. *Proceedings of the 26th International Conference on Software Engineering* (*ICSE 2004*). ACM Press: New York, 2004; 106–115.
22. Hsu HY, Orso A. MINTS: A general framework and tool for supporting test-suite minimization. *Proceedings of the 31st International Conference on Software Engineering* (*ICSE 2009*). IEEE Computer Society: Silver Spring, MD, 2009; 419–429.
23. Walcott KR, Soffa ML, Kapfhammer GM, Roos RS. Time aware test suite prioritization. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 2006*). ACM Press: New York, 2006; 1–12.
24. Yoo S, Harman M. Pareto efficient multi-objective test case selection. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 2007*). ACM Press: New York, 2007; 140–150.
25. McMaster S, Memon A. Call-stack coverage for gui test suite reduction. *IEEE Transactions on Software Engineering* 2008; **34**(1):99–115.
26. Smith A, Geiger J, Kapfhammer GM, Soffa ML. Test suite reduction and prioritization with call trees. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2007*). ACM Press: New York, 2007.
27. Harder M, Mellen J, Ernst MD. Improving test suites via operational abstraction. *Proceedings of the 25th International Conference on Software Engineering* (*ICSE 2003*). IEEE Computer Society: Silver Spring, MD, 2003; 60–71.
28. Ernst MD, Cockrell J, Griswold WG, Notkin D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 2001; **27**(2):99–123.
29. Leitner A, Oriol M, Zeller A, Ciupa I, Meyer B. Efficient unit test case minimization. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2007*). ACM Press: New York, 2007; 417–420.
30. Zeller A. Yesterday, my program worked. Today, it does not why? *SIGSOFT Software Engineering Notes* 1999; **24**(6):253–267.
31. Schroeder PJ, Korel B. Black-box test reduction using input-output analysis. *SIGSOFT Software Engineering Notes* 2000; **25**(5):173–177.
32. Vaysburg B, Tahat LH, Korel B. Dependence analysis in reduction of requirement based test suites. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 2002*). ACM Press: New York, 2002; 107–111.
33. Korel B, Tahat L, Vaysburg B. Model based regression test reduction using dependence analysis. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2002*). IEEE Computer Society: Silver Spring, MD, 2002; 214–225.
34. Chen Y, Probert RL, Ural H. Regression test suite reduction using extended dependence analysis. *Proceedings of the 4th International Workshop on Software Quality Assurance* (*SOQUA 2007*). ACM Press: New York, 2007; 62–69.
35. Anido R, Cavalli AR, Lima Jr LP, Yevtushenko N. Test suite minimization for testing in context. *Software Testing, Verification and Reliability* 2003; **13**(3):141–155.
36. Kaminski GK, Ammann P. Using logic criterion feasibility to reduce test set size while guaranteeing fault detection. *Proceedings of the International Conference on Software Testing, Verification, and Validation 2009* (*ICST 2009*). IEEE Computer Society: Silver Spring, MD, 2009; 356–365.
37. Lau MF, Yu YT. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering Methodology* 2005; **14**(3):247–276.
38. Rothermel G, Harrold MJ, Ostrin J, Hong C. An empirical study of the effects of minimization on the fault detection capabilities of test suites. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1998*). IEEE Computer Press: Silver Spring, MD, 1998; 34–43.
39. Wong WE, Horgan JR, London S, Mathur AP. Effect of test set minimization on fault detection effectiveness. *Software Practice and Experience* 1998; **28**(4):347–369.
40. Wong WE, Horgan JR, Mathur AP, Pasquini A. Test set size minimization and fault detection effectiveness: A case study in a space application. *The Journal of Systems and Software* 1999; **48**(2):79–89.
41. Hutchins M, Foster H, Goradia T, Ostrand T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. *Proceedings of the 16th International Conference on Software Engineering* (*ICSE 1994*). IEEE Computer Press: Silver Spring, MD, 1994; 191–200.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

42. McMaster S, Memon AM. Fault detection probability analysis for coverage-based test suite reduction. *Proceedings of the 21st IEEE International Conference on Software Maintenance* (*ICSM'07*). IEEE Computer Society: Silver Spring, MD, 2007.
43. McMaster S, Memon AM. Call stack coverage for test suite reduction. *Proceedings of the 21st IEEE International Conference on Software Maintenance* (*ICSM'05*). IEEE Computer Society: Washington, DC, U.S.A., 2005; 539–548.
44. Yu Y, Jones JA, Harrold MJ. An empirical study of the effects of test-suite reduction on fault localization. *Proceedings of the International Conference on Software Engineering* (*ICSE 2008*). ACM Press: New York, 2008; 201–210.
45. Rothermel G, Harrold MJ. A framework for evaluating regression test selection techniques. *Proceedings of the 16th International Conference on Software Engineering* (*ICSE 1994*). IEEE Computer Press: Silver Spring, MD, 1994; 201–210.
46. Rothermel G, Harrold MJ. A safe, efficient algorithm for regression test selection. *Proceedings of the International Conference on Software Maintenance* (*ICSM 2003*). IEEE Computer Press: Silver Spring, MD, 1993; 358–367.
47. Fischer K. A test case selection method for the validation of software maintenance modifications. *Proceedings of the International Computer Software and Applications Conference*. IEEE Computer Press: Silver Spring, MD, 1977; 421–426.
48. Fischer K, Raji F, Chruscicki A. A methodology for retesting modified software. *Proceedings of the National Telecommunications Conference*. IEEE Computer Society Press: Silver Spring, MD, 1981; 1–6.
49. Gupta R, Harrold MJ, Soffa ML. An approach to regression testing using slicing. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1992*). IEEE Computer Press: Silver Spring, MD, 1992; 299–308.
50. Harrold MJ, Soffa ML. An incremental approach to unit testing during maintenance. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1998*). IEEE Computer Press: Silver Spring, MD, 1988; 362–367.
51. Harrold MJ, Soffa ML. Interprocedual data flow testing. *Proceedings of the 3rd ACM SIGSOFT Symposium on Software Testing*, *Analysis*, *and Verification* (*TAV3*). ACM Press: New York, 1989; 158–167.
52. Taha AB, Thebaut SM, Liu SS. An approach to software fault localization and revalidation based on incremental data flow analysis. *Proceedings of the International Computer Software and Applications Conference* (*COMPSAC 1989*). IEEE Computer Press: Silver Spring, MD, 1989; 527–534.
53. Yau SS, Kishimoto Z. A method for revalidating modified programs in the maintenance phase. *Proceedings of the International Computer Software and Applications Conference* (*COMPSAC 1987*). IEEE Computer Press: Silver Spring, MD, 1987; 272–277.
54. Agrawal H, Horgan JR, Krauser EW, London SA. Incremental regression testing. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1993*). IEEE Computer Society: Silver Spring, MD, 1993; 348–357.
55. Rothermel G. Efficient, effective regression testing using safe test selection techniques. *PhD Thesis*, University of Clemson, May 1996.
56. Rothermel G, Harrold MJ. Selecting tests and identifying test coverage requirements for modified software. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 1994*). ACM Press: New York, 1994; 169–184.
57. Rothermel G, Harrold MJ. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 1997; **6**(2):173–210.
58. Vokolos F, Frankl P. Pythia: A regression test selection tool based on text differencing. *Proceedings of the International Conference on Reliability Quality and Safety of Software Intensive Systems*. Chapman & Hall, 1997.
59. Vokolos F, Frankl P. Empirical evaluation of the textual differencing regression testing technique. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 1998*). IEEE Computer Press: Silver Spring, MD, 1998; 44–53.
60. Bates S, Horwitz S. Incremental program testing using program dependence graphs. *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press: New York, 1993; 384–396.
61. Benedusi P, Cmitile A, De Carlini U. Post-maintenance testing based on path change analysis. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1988*). IEEE Computer Press: Silver Spring, MD, 1988; 352–361.
62. Chen YF, Rosenblum D, Vo KP. Testtube: A system for selective regression testing. *Proceedings of the 16th International Conference on Software Engineering* (*ICSE 1994*). ACM Press: New York, 1994; 211–220.
63. Leung HKN, White L. Insights into testing and regression testing global variables. *Journal of Software Maintenance* 1990; **2**(4):209–222.
64. Leung HKN, White L. A study of integration testing and software regression at the integration level. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1990*). IEEE Computer Press: Silver Spring, MD, 1990; 290–301.
65. White LJ, Leung HKN. A firewall concept for both control-flow and data-flow in regression integration testing. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1992*). IEEE Computer Press: Silver Spring, MD, 1992; 262–271.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

66. White LJ, Narayanswamy V, Friedman T, Kirschenbaum M, Piwowarski P, Oha M. Test manager: A regression testing tool. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1993*). IEEE Computer Press: Silver Spring, MD, 338–347.
67. Laski J, Szermer W. Identification of program modifications and its applications in software maintenance. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1992*). IEEE Computer Press: Silver Spring, MD, 1992; 282–290.
68. Briand LC, Labiche Y, Buist K, Soccar G. Automating impact analysis and regression test selection based on UML designs. *Proceedings of the International Conference on Software Maintenance* (*ICSM 2002*). IEEE Computer Society: Silver Spring, MD, 2002; 252–261.
69. Briand LC, Labiche Y, He S. Automating regression test selection based on UML designs. *Journal of Information and Software Technology* 2009; **51**(1):16–30.
70. Lee JAN, He X. A methodology for test selection. *The Journal of Systems and Software* 1990; **13**(3):177–185.
71. Hartmann J, Robson DJ. Revalidation during the software maintenance phase. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1989*). IEEE Computer Press: Silver Spring, MD, 1989; 70–80.
72. Hartmann J, Robson DJ. Retest-development of a selective revalidation prototype environment for use in software maintenance. *Proceedings of the International Conference on System Sciences*, vol. 2. IEEE Computer Press: Silver Spring, MD, 1990; 92–101.
73. Hartmann J, Robson DJ. Techniques for selective revalidation. *IEEE Software* 1990; **7**(1):31–36.
74. Harrold MJ, Soffa ML. An incremental data flow testing tool. *Proceedings of the 6th International Conference on Testing Computer Software* (*ICTCS 1989*). ACM Press: New York, 1989.
75. Wong WE, Horgan JR, London S, Bellcore HA. A study of effective regression testing in practice. *Proceedings of the 8th International Symposium on Software Reliability Engineering* (*ISSRE 1997*). IEEE Computer Society: Silver Spring, MD, 1997; 264–275.
76. Fisher II M, Jin D, Rothermel G, Burnett M. Test reuse in the spreadsheet paradigm. *Proceedings of the International Symposium on Software Reliability Engineering* (*ISSRE 2002*). IEEE Computer Society: Silver Spring, MD, 2002; 257–268.
77. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Press, 10 December 1990.
78. Rothermel G, Harrold MJ. Experience with regression test selection. *Empirical Software Engineering*: *An International Journal* 1997; **2**(2):178–188.
79. Rothermel G, Harrold MJ. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering* 1998; **24**(6):401–419.
80. Ball T. On the limit of control flow analysis for regression test selection. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 1998*). ACM Press: New York, 1998; 134–142.
81. Rothermel G, Harrold MJ, Dedhia J. Regression test selection for C++ software. *Software Testing*, *Verification and Reliability* 2000; **10**(2):77–109.
82. Harrold MJ, Jones JA, Li T, Liang D, Orso A, Pennings M, Sinha S, Spoon S. Regression test selection for Java software. *ACM Conference on Object-oriented Programming*, *Systems*, *Languages*, *and Applications* (*OOPSLA 2001*). ACM Press: New York, 2001; 312–326.
83. Xu G, Rountev A. Regression test selection for AspectJ software. *Proceedings of the 29th International Conference on Software Engineering* (*ICSE 2007*). IEEE Computer Society: Silver Spring, MD, 2007; 65–74.
84. Zhao J, Xie T, Li N. Towards regression test selection for aspect-oriented programs. *Proceedings of the 2nd Workshop on Testing Aspect-oriented Programs* (*WTAOP 2006*). ACM Press: New York, 2006; 21–26.
85. Beydeda S, Gruhn V. Integrating white- and black-box techniques for class-level regression testing. *Proceedings of the 25th IEEE International Computer Software and Applications Conference* (*COMPSAC 2001*). IEEE Computer Society Press: Silver Spring, MD, 2001; 357–362.
86. Orso A, Shi N, Harrold MJ. Scaling regression testing to large software systems. *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*FSE 2004*). ACM Press: New York, 2004; 241–251.
87. Martins E, Vieira VG. Regression test selection for testable classes. *Dependable Computing—EDCC 2005* (*Lecture Notes in Computer Science*, vol. 3463/2005). 2005; 453–470.
88. Chen Y, Probert RL, Sims DP. Specification-based regression test selection with risk analysis. *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research* (*CASCON 2002*). IBM Press, 2002; 1–14.
89. Orso A, Harrold MJ, Rosenblum DS, Rothermel G, Soffa ML, Do H. Using component metadata to support the regression testing of component-based software. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2001*). IEEE Computer Society Press: Silver Spring, MD, 2001.
90. Orso A, Do H, Rothermel G, Harrold MJ, Rosenblum DS. Using component metadata to regression test component-based software: Research articles. *Software Testing*, *Verification*, *and Reliability* 2007; **17**(2):61–94.
91. Lin F, Ruth M, Tu S. Applying safe regression test selection techniques to java web services. *Proceedings of the International Conference on Next Generation Web Services Practices* (*NWESP 2006*). IEEE Computer Society: Silver Spring, MD, 2006; 133–142.
92. Ruth M, Tu S. A safe regression test selection technique for web services. *Proceedings of the 2nd International Conference on Internet and Web Applications and Services* (*ICIW 2007*). IEEE Computer Press: Silver Spring, MD, 2007; 47–47.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

93. Ruth M, Tu S. Concurrency issues in automating rts for web services. *Proceedings of the IEEE International Conference on Web Services* (*ICWS 2007*). IEEE Computer Press: Silver Spring, MD, 1142–1143.
94. Tarhini A, Fouchal H, Mansour N. Regression testing web services-based applications. *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications* (*AICCSA 2006*). IEEE Computer Press: Silver Spring, MD, 2006; 163–170.
95. Ruth M, Oh S, Loup A, Horton B, Gallet O, Mata M, Tu S. Towards automatic regression test selection for web services. *Proceedings of the 31st International Computer Software and Applications Conference* (*COMPSAC 2007*). IEEE Computer Press: Silver Spring, MD, 2007; 729–736.
96. Binkley D. Reducing the cost of regression testing by semantics guided test case selection. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1995*). IEEE Computer Society: Silver Spring, MD, 1995; 251–260.
97. Binkley
D. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering* 1997; **23**(8): 498–516.
98. Kung DC, Gao J, Hsia P, Lin J, Toyoshima Y. Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object-oriented Programming* 1995; **8**(2):51–65.
99. White L, Robinson B. Industrial real-time regression testing and analysis using firewalls. *Proceedings of the 20th IEEE International Conference on Software Maintenance* (*ICSM 2004*). IEEE Computer Press: Silver Spring, MD, 2004; 18–27.
100. White L, Jaber K, Robinson B, Rajlich V. Extended firewall for regression testing: An experience report. *Journal of Software Maintenance and Evolution* 2008; **20**(6):419–433.
101. White L, Almezen H, Sastry S. Firewall regression testing of gui sequences and their interactions. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2003*). IEEE Computer Press: Silver Spring, MD, 2003; 398–409.
102. Zheng J, Robinson B, Williams L, Smiley K. A lightweight process for change identification and regression test selection in using cots components. *Proceedings of the 5th International Conference on Commercial-off-the-shelf* (*COTS*)*-based Software Systems* (*ICCBSS 2006*). IEEE Computer Society: Silver Spring, MD, 2006; 137–146. DOI: http://dx.doi.org/10.1109/ICCBSS.2006.1.
103. Zheng J, Robinson B, Williams L, Smiley K. Applying regression test selection for COTS-based applications. *Proceedings of the 28th International Conference on Software Engineering* (*ICSE 2006*). ACM Press: New York, 2006; 512–522.
104. Zheng J, Williams L, Robinson B. Pallino: Automation to support regression test selection for COTS-based applications. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2007*). ACM Press: New York, 2007; 224–233.
105. Zheng J, Williams L, Robinson B, Smiley K. Regression test selection for black-box dynamic link library components. *Proceedings of the 2nd International Workshop on Incorporating COTS Software into Software Systems*: *Tools and Techniques* (*IWICSS 2007*). IEEE Computer Society: Silver Spring, MD, 2007; 9–14.
106. Skoglund M, Runeson P. A case study of the class firewall regression test selection technique on a large scale distributed software system. *Proceedings of the International Symposium on Empirical Software Engineering* (*ISESE 2005*). IEEE Computer Society Press: Silver Spring, MD, 2005; 74–83.
107. Deng D, Sheu PY, Wang T. Model-based testing and maintenance. *Proceedings of the 6th IEEE International Symposium on Multimedia Software Engineering* (*MMSE 2004*). IEEE Computer Society Press: Silver Spring, MD, 2004; 278–285.
108. Pilskalns O, Uyan G, Andrews A. Regression testing uml designs. *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (*ICSM 2006*). IEEE Computer Society Press: Silver Spring, MD, 2006; 254–264.
109. Farooq Qua, Iqbal MZZ, Malik ZI, Nadeem A. An approach for selective state machine based regression testing. *Proceedings of the 3rd International Workshop on Advances in Model-based Testing* (*A-MOST 2007*). ACM: New York, NY, U.S.A., 2007; 44–52.
110. Le Traon Y, Jeron T, Jezequel JM, Morel P. Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability* 2000; **49**(1):12–25.
111. Wu Y, Offutt J. Maintaining evolving component-based software with UML. *Proceedings of the 7th European Conference on Software Maintenance and Reengineering* (*CSMR 2003*). IEEE Computer Society Press: Silver Spring, MD, 2003; 133–142.
112. Muccini H, Dias M, Richardson DJ. Reasoning about software architecture-based regression testing through a case study. *Proceedings of the 29th International Computer Software and Applications Conference* (*COMPSAC 2005*), vol. 2. IEEE Computer Society: Silver Spring, MD, 2005; 189–195.
113. Muccini H, Dias M, Richardson DJ. Software-architecture based regression testing. *The Journal of Systems and Software* 2006; **79**(10):1379–1396.
114. Harrold MJ. Testing evolving software. *The Journal of Systems and Software* 1999; **47**(2–3):173–181.
115. Rothermel G, Untch RH, Chu C, Harrold MJ. Test case prioritization: An empirical study. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1999*). IEEE Computer Press: Silver Spring, MD, 1999; 179–188.

Copyright © 2010 John Wiley & Sons, Ltd.

116. Elbaum SG, Malishevsky AG, Rothermel G. Prioritizing test cases for regression testing. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 2000*). ACM Press: New York, 2000; 102–112.
117. Elbaum S, Gable D, Rothermel G. Understanding and measuring the sources of variation in the prioritization of regression test suites. *Proceedings of the Seventh International Software Metrics Symposium* (*METRICS 2001*). IEEE Computer Press: Silver Spring, MD, 2001; 169–179.
118. Elbaum SG, Malishevsky AG, Rothermel G. Incorporating varying test costs and fault severities into test case prioritization. *Proceedings of the International Conference on Software Engineering* (*ICSE 2001*). ACM Press: New York, 2001; 329–338.
119. Malishevsky A, Rothermel G, Elbaum S. Modeling the cost-benefits tradeoffs for regression testing techniques. *Proceedings of the International Conference on Software Maintenance* (*ICSM 2002*). IEEE Computer Press: Silver Spring, MD, 2002; 230–240.
120. Rothermel G, Elbaum S, Malishevsky A, Kallakuri P, Davia B. The impact of test suite granularity on the cost-effectiveness of regression testing. *Proceedings of the 24th International Conference on Software Engineering* (*ICSE 2002*). ACM Press: New York, 2002; 130–140.
121. Rothermel G, Untch RJ, Chu C. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 2001; **27**(10):929–948.
122. Budd TA. Mutation analysis of program test data. *PhD Thesis*, Yale University, New Haven, CT, U.S.A., 1980.
123. Elbaum S, Malishevsky A, Rothermel G. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering* 2002; **28**(2):159–182.
124. Jones JA, Harrold MJ. Test-suite reduction and prioritization for modified condition/decision coverage. *Proceedings of the International Conference on Software Maintenance* (*ICSM 2001*). IEEE Computer Press: Silver Spring, MD, 2001; 92–101.
125. Chilenski J, Miller S. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal* 1994; **9**(5):193–200.
126. Srivastava A, Thiagarajan J. Effectively prioritizing tests in development environment. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 2002*). ACM Press: New York, 2002; 97–106.
127. Do H, Rothermel G, Kinneer A. Empirical studies of test case prioritization in a junit testing environment. *Proceedings of the 15th International Symposium on Software Reliability Engineering* (*ISSRE 2004*). IEEE Computer Press: Silver Spring, MD, 2004; 113–124.
128. Li Z, Harman M, Hierons RM. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering* 2007; **33**(4):225–237.
129. Bryce RC, Colbourn CJ, Cohen MB. A framework of greedy methods for constructing interaction test suites. *Proceedings of the 27th International Conference on Software Engineering* (*ICSE 2005*). ACM Press: New York, 2005; 146–155.
130. Cohen MB, Dwyer MB, Shi J. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering* 2008; **34**(5):633–650.
131. Bryce RC, Colbourn CJ. Test prioritization for pairwise interaction coverage. *Proceedings of the ACM Workshop on Advances in Model-based Testing* (*A-MOST 2005*). ACM Press: New York, 2005; 1–7.
132. Bryce RC, Colbourn CJ. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Journal of Information and Software Technology* 2006; **48**(10):960–970.
133. Qu X, Cohen MB, Woolf KM. Combinatorial interaction regression testing: A study of test case generation and prioritization. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2007*). IEEE Computer Press: Silver Spring, MD, 2007; 255–264.
134. Qu X, Cohen MB, Rothermel G. Configuration-aware regression testing: An empirical study of sampling and prioritization. *Proceedings of the ACM International Symposium on Software Testing and Analysis* (*ISSTA 2008*). ACM Press: New York, 2008; 75–86.
135. Bryce RC, Memon AM. Test suite prioritization by interaction coverage. *Proceedings of the Workshop on Domain Specific Approaches to Software Test Automation* (*DOSTA 2007*). ACM: New York, 2007; 1–7.
136. Leon D, Podgurski A. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. *Proceedings of the IEEE International Symposium on Software Reliability Engineering* (*ISSRE 2003*). IEEE Computer Press: Silver Spring, MD, 2003; 442–456.
137. Tonella P, Avesani P, Susi A. Using the case-based ranking methodology for test case prioritization. *Proceedings of the 22nd International Conference on Software Maintenance* (*ICSM 2006*). IEEE Computer Society: Silver Spring, MD, 2006; 123–133.
138. Yoo S, Harman M, Tonella P, Susi A. Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA 2009*). ACM Press: New York, 2009; 201–211.
139. Srikanth H, Williams L, Osborne J. System test case prioritization of new and regression test cases. *Proceedings of the International Symposium on Empirical Software Engineering*. IEEE Computer Society Press: Silver Spring, MD, 2005; 64–73.
140. Freund Y, Schapire R. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence* 1999; **14**(5):771–780.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

141. Freund Y, Iyer R, Schapire RE, Singer Y. An efficient boosting algorithm for combining preferences. *Proceedings of the 15th International Conference on Machine Learning* (*ICML 1998*), Shavlik JW (ed.). Morgan Kaufmann Publishers: Los Altos, CA, 1998; 170–178.
142. Saaty T. *The Analytic Hierarchy Process*, *Planning*, *Piority Setting*, *Resource Allocation*. McGraw-Hill: New York, NY, U.S.A., 1980.
143. Karlsson J, Wohlin C, Regnell B. An evaluation of methods for prioritizing software requirements. *Information and Software Technology* 1998; **39**(14–15):939–947.
144. Kim JM, Porter A. A history-based test prioritization technique for regression testing in resource constrained environments. *Proceedings of the 24th International Conference on Software Engineering* (*ICSE 2002*). ACM Press: New York, 2002; 119–129.
145. Mirarab S, Tahvildari L. A prioritization approach for software test cases based on Bayesian networks. *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*. Springer: Berlin, 2007; 276–290.
146. Mirarab S, Tahvildari L. An empirical study on Bayesian network-based approach for test case prioritization. *Proceedings of the International Conference on Software Testing*, *Verification and Validation*. IEEE Computer Society: Silver Spring, MD, 2008; 278–287.
147. Sherriff M, Lake M, Williams L. Prioritization of regression tests using singular value decomposition with empirical change records. *Proceedings of the 18th IEEE International Symposium on Software Reliability* (*ISSRE 2007*). IEEE Computer Society: Washington, DC, U.S.A., 2007; 81–90.
148. Krishnamoorthi R, Sahaaya Arul Mary SA. Factor oriented requirement coverage based system test case prioritization of new and regression test cases. *Information and Software Technology* 2009; **51**(4):799–808.
149. Korel B, Tahat L, Harman M. Test prioritization using system models. *Proceedings of the 21st IEEE International Conference on Software Maintenance* (*ICSM 2005*), 2005; 559–568.
150. Korel B, Koutsogiannakis G, Tahat LH. Model-based test prioritization heuristic methods and their evaluation. *Proceedings of the 3rd International Workshop on Advances in Model-based Testing* (*A-MOST 2007*). ACM Press: New York, 2007; 34–43.
151. Korel B, Koutsogiannakis G, Tahat L. Application of system models in regression test suite prioritization. *Proceedings of the IEEE International Conference on Software Maintenance 2008* (*ICSM 2008*). IEEE Computer Society Press: Silver Spring, MD, 2008; 247–256.
152. Hou SS, Zhang L, Xie T, Mei H, Sun JS. Applying interface-contract mutation in regression testing of component-based software. *Proceedings of the 23rd IEEE International Conference on Software Maintenance* (*ICSM 2007*). IEEE Computer Society Press: Silver Spring, MD, 2007; 174–183.
153. Sampath S, Bryce RC, Viswanath G, Kandimalla V, Koru AG. Prioritizing user-session-based test cases for web applications testing. *Proceedings of the 1st International Conference on Software Testing Verification and Validation* (*ICST 2008*). IEEE Computer Society: Silver Spring, MD, 2008; 141–150.
154. Fraser G, Wotawa F. Test-case prioritization with model-checkers. *SE'07*: *Proceedings of the 25th Conference on IASTED International Multi-conference*. ACTA Press: Anaheim, CA, U.S.A., 2007; 267–272.
155. Fraser G, Wotawa F. Property relevant software testing with model-checkers. *SIGSOFT Software Engineering Notes* 2006; **31**(6):1–10.
156. Rummel M, Kapfhammer GM, Thall A. Towards the prioritization of regression test suites with data flow information. *Proceedings of the 20th Symposium on Applied Computing* (*SAC 2005*). ACM Press: New York, 2005.
157. Jeffrey D, Gupta N. Test case prioritization using relevant slices. *Proceedings of the 30th Annual International Computer Software and Applications Conference* (*COMPSAC 2006*). IEEE Computer Society: Washington, DC, U.S.A., 2006; 411–420.
158. Do H, Mirarab SM, Tahvildari L, Rothermel G. An empirical study of the effect of time constraints on the cost-benefits of regression testing. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press: New York, 2008; 71–82.
159. Hou SS, Zhang L, Xie T, Sun JS. Quota-constrained test-case prioritization for regression testing of service-centric systems. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2008*). IEEE Computer Society Press: Silver Spring, MD, 2008.
160. Zhang L, Hou SS, Guo C, Xie T, Mei H. Time-aware test-case prioritization using Integer Linear Programming. *Proceedings of the International Conference on Software Testing and Analysis* (*ISSTA 2009*). ACM Press: New York, 2009; 212–222.
161. Andrews JH, Briand LC, Labiche Y. *Proceedings of the 27th International Conference on Software Engineering* (*ICSE 2005*). ACM Press: New York, 2005; 402–411.
162. Do H, Rothermel G. A controlled experiment assessing test case prioritization techniques via mutation faults. *Proceedings of the 21st IEEE International Conference on Software Maintenance* (*ICSM 2005*). IEEE Computer Society Press: Silver Spring, MD, 2005; 411–420.
163. Do H, Rothermel G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering* 2006; **32**(9):733–752.
164. Elbaum S, Kallakuri P, Malishevsky A, Rothermel G, Kanduri S. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Software Testing*, *Verification*, *and Reliability* 2003; **13**(2):65–83.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

165. Rothermel G, Elbaum S, Malishevsky A, Kallakuri P, Qiu X. On test suite composition and cost-effective regression testing. *ACM Transactions on Software Engineering and Methodology* 2004; **13**(3):277–331.
166. Kim JM, Porter A, Rothermel G. An empirical study of regression test application frequency. *Proceedings of the 22nd International Conference on Software Engineering* (*ICSE 2000*). ACM Press: New York, 2000; 126–135.
167. Kim JM, Porter A, Rothermel G. An empirical study of regression test application frequency. *Software Testing*, *Verification*, *and Reliability* 2005; **15**(4):257–279.
168. Elbaum S, Rothermel G, Kanduri S, Malishevsky AG. Selecting a cost-effective test case prioritization technique. *Software Quality Control* 2004; **12**(3):185–210.
169. Rosenblum D, Weyuker E. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering* 1997; **23**(3):146–156.
170. Bolsky MI, Korn DG. *The New KornShell Command and Programming Language*. Prentice-Hall PTR: Upper Saddle River, NJ, U.S.A., 1995.
171. Korn D, phong Vo K. SFIO: Safe/Fast String/File IO. *Proceedings of the Summer Usenix Conference 1991*, 1991; 235–256.
172. Leung HKN, White L. A cost model to compare regression test strategies. *Proceedings of the International Conference on Software Maintenance* (*ICSM 1991*). IEEE Computer Press: Silver Spring, MD, 1991; 201–208.
173. Harrold MJ, Rosenblum DS, Rothermel G, Weyuker EJ. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering* 2001; **27**(3):248–263.
174. Ramey C, Fox B. *Bash Reference Manual* (2.2 edn). O'Reilly and Associates: Sebastopol, CA, 1998.
175. Do H, Rothermel G, Kinneer A. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering* 2006; **11**(1):33–70.
176. Smith AM, Kapfhammer GM. An empirical study of incorporating cost into test suite reduction and prioritization. *Proceedings of the 24th Symposium on Applied Computing* (*SAC 2009*). ACM Press: New York, 2009.
177. Do H, Rothermel G. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. *SIGSOFT '06/FSE-14*: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press: New York, 2006; 141–151.
178. Do H, Rothermel G. Using sensitivity analysis to create simplified economic models for regression testing. *Proceedings*
of the ACM SIGSOFT International Symposium on Software Testing and Analysis (*ISSTA 2008*). ACM Press: New York, 2008; 51–61.
179. Do H, Elbaum SG, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 2005; **10**(4):405–435.
180. Memon AM, Soffa ML. Regression testing of GUIs. *ESEC/FSE-11*: *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM: New York, NY, U.S.A., 2003; 118–127.
181. Jia Y, Harman M. Constructing subtle faults using higher order mutation testing. *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation* (*SCAM 2008*). IEEE Computer Society Press: Silver Spring, MD, 2008; 249–258.
182. Harrold M, Orso A. Retesting software during development and maintenance. *Frontiers of Software Maintenance* (*FoSM 2008*). IEEE Computer Society Press: Silver Spring, MD, 2008; 99–108.
183. McMinn P. Search-based software test data generation: A survey. *Software Testing*, *Verification and Reliability* 2004; **14**(2):105–156.
184. Apiwattanapong T, Santelices R, Chittimalli PK, Orso A, Harrold. MJ, Matrix: Maintenance-oriented testing requirements identifier and examiner. *Proceedings of the Testing*: *Academic and Industrial Conference on Practice And Research Techniques*. IEEE Computer Society: Los Alamitos, CA, U.S.A., 2006; 137–146.
185. Santelices RA, Chittimalli PK, Apiwattanapong T, Orso A, Harrold MJ. Test-suite augmentation for evolving software. *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE: New York, 2008; 218–227.
186. Memon AM. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Transactions on Software Engineering Methodology* 2008; **18**(2):1–36.
187. Alshahwan N, Harman M. Automated session data repair for web application regression testing. *Proceedings of the 2008 International Conference on Software Testing*, *Verification*, *and Validation*. IEEE Computer Society: Los Alamitos, CA, U.S.A., 2008; 298–307.
188. Bertolini C, Peres G, d'Amorim M, Mota A. An empirical evaluation of automated black box testing techniques for crashing guis. *Proceedings of the 2nd International Conference on Software Testing Verification and Validation* (*ICST 2009*). IEEE Computer Press: Silver Spring, MD, 2009; 21–30.
189. Fu C, Grechanik M, Xie Q. Inferring types of references to gui objects in test scripts. *Proceedings of the 2nd International Conference on Software Testing Verification and Validation* (*ICST 2009*). IEEE Computer Press: Silver Spring, MD, 2009; 1–10.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

University of Zurich[UZH]

Zurich Open Repository and Archive
University of Zurich
Main Library
Strickhofstrasse 39
CH-8057 Zurich
www.zora.uzh.ch

Year: 2018

Modern code review: a case study at google

Sadowski, Caitlin ; Söderberg, Emma ; Church, Luke ; Sipko, Michal ; Bacchelli, Alberto

DOI: https://doi.org/10.1145/3183519.3183525

Posted at the Zurich Open Repository and Archive, University of Zurich
ZORA URL: https://doi.org/10.5167/uzh-152982
Conference or Workshop Item
Published Version

The following work is licensed under a Creative Commons: Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) License.

Originally published at:
Sadowski, Caitlin; Söderberg, Emma; Church, Luke; Sipko, Michal; Bacchelli, Alberto (2018). Modern code review: a case study at google. In: the 40th International Conference Conference on Software Engineering: Software Engineering in Practice (ICSE SEiP), Gothenburg, Sweden, 27 June 2018 - 3 July 2018, 181-190.
DOI: https://doi.org/10.1145/3183519.3183525

Modern Code Review: A Case Study at Google

Caitlin Sadowski, Emma Söderberg,
Luke Church, Michal Sipko
Google, Inc.
{supertri,emso,lukechurch,sipkom}@google.com

Alberto Bacchelli
University of Zurich
bacchelli@ifi.uzh.ch

ABSTRACT

Employing lightweight, tool-based code review of code changes (aka *modern code review*) has become the norm for a wide variety of open-source and industrial systems. In this paper, we make an exploratory investigation of modern code review at Google. Google introduced code review early on and evolved it over the years; our study sheds light on why Google introduced this practice and analyzes its current status, after the process has been refined through decades of code changes and millions of code reviews. By means of 12 interviews, a survey with 44 respondents, and the analysis of review logs for 9 million reviewed changes, we investigate motivations behind code review at Google, current practices, and developers' satisfaction and challenges.

CCS CONCEPTS

- **Software and its engineering → Software maintenance tools**;

ACM Reference format:
Caitlin Sadowski, Emma Söderberg,
Luke Church, Michal Sipko and Alberto Bacchelli. 2018. Modern Code Review: A Case Study at Google. In *Proceedings of 40th International Conference on Software Engineering: Software Engineering in Practice Track, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE-SEIP '18),* 10 pages.
DOI: 10.1145/3183519.3183525

1 INTRODUCTION

Peer code review, a manual inspection of source code by developers other than the author, is recognized as a valuable tool for improving the quality of software projects [2, 3]. In 1976, Fagan formalized a highly structured process for code reviewing—code inspections [16]. Over the years, researchers provided evidence on the benefits of code inspection, especially for defect finding, but the cumbersome, time-consuming, and synchronous nature of this approach hindered its universal adoption in practice [37]. Nowadays, most organizations adopt more lightweight code review practices to limit the inefficiencies of inspections [33]. Modern code review is (1) informal (in contrast to Fagan-style), (2) tool-based [32], (3) asynchronous, and (4) focused on reviewing code changes.

An open research challenge is understanding which practices represent valuable and effective methods of review in this novel context. Rigby and Bird quantitatively analyzed code review data from software projects spanning varying domains as well as organizations and found five strongly convergent aspects [33], which they conjectured can be prescriptive to other projects. The analysis of Rigby and Bird is based on the value of a *broad* perspective (that analyzes multiple projects from different contexts). For the development of an empirical body of knowledge, championed by Basili [7], it is essential to also consider a *focused and longitudinal* perspective that analyzes a single case. This paper expands on work by Rigby and Bird to focus on the review practices and characteristics established at Google, *i.e.*, a company with a multi-decade history of code review and a high-volume of daily reviews to learn from. This paper can be (1) prescriptive to practitioners performing code review and (2) compelling for researchers who want to understand and support this novel process.

Code review has been a required part of software development at Google since very early on in the company's history; because it was introduced so early on, it has become a core part of Google culture. The process and tooling for code review at Google have been iteratively refined for more than a decade and is applied by more than 25,000 developers making more than 20,000 source code changes each workday, in dozens of offices around the world [30].

We conduct our analysis in the form of an exploratory investigation focusing on three aspects of code review, in line with and expanding on the work by Rigby and Bird [33]: (1) The motivations driving code review, (2) the current practices, and (3) the perception of developers on code review, focusing on challenges encountered with a specific review (*breakdowns* in the review process) and satisfaction. Our research method combines input from multiple data sources: 12 semi-structured interviews with Google developers, an internal survey sent to engineers who recently sent changes to review with 44 responses, and log data from Google's code review tool pertaining to 9 million reviews over two years.

We find that the process at Google is markedly lighter weight than in other contexts, based on a single reviewer, quick iterations, small changes, and a tight integration with the code review tool. Breakdowns still exist, however, due to the complexity of the interactions that occur around code review. Nevertheless, developers consider this process valuable, confirm that it works well at scale, and conduct it for several reasons that also depend on the relationship between author and reviewers. Finally, we find evidence on the use of the code review tool beyond collaborative review and corroboration for the importance of code review as an educational tool.

ICSE-SEIP '18, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s). This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27-June 3, 2018*, http://dx.doi.org/10.1145/3183519.3183525.

2 BACKGROUND AND RELATED WORK

We describe the review processes investigated in literature, then we detail convergent code review practices across these processes [33].

2.1 Code Review Processes and Contexts

Code Inspections. Software inspections are one of the first formalized processes for code review. This highly structured process involves planning, overview, preparation, inspection meeting, reworking, and follow-up [16]. The goal of code inspections is to find defects during a synchronized inspection meeting, with authors and reviewers sitting in the same room to examine code changes. Kollanus and Koskinen compiled the most recent literature survey on code inspection research [25] . They found that the vast majority of studies on code inspections are empirical in nature. There is a consensus about the overall value of code inspection as a defect finding technique and the value of reading techniques to engage the inspectors. Overall, research on code inspection has declined from 2005, in line with the spread of the internet and the growth of asynchronous code review processes.

Asynchronous review via email. Until the late 2000s, most large OSS projects adopted a form of remote, asynchronous reviews, relying on patches sent to communication channels such as mailing lists and issue tracking systems. In this setting, project members evaluate contributed patches and ask for modifications through these channels. When a patch is deemed of high enough quality, core developers commit it to the codebase. Trusted committers may have a commit-then-review process instead of doing pre-commit reviews [33]. Rigby *et al.* were among the first to do extensive work in this setting; they found that this type of review "*has little in common* [with code inspections] *beyond a belief that peers will effectively find software defects*" [34]. Kononenko *et al.* analyzed the same setting and find that review response time and acceptance are related to social factors, such as reviewer load and change author experience [26], which were not present in code inspections.

Tool-based review. To bring structure to the process of reviewing patches, several tools emerged in OSS and industrial settings. These tools support the *logistics* of the review process: (1) The author of a patch submits it to the code review tool, (2) the reviewers can see the diff of the proposed code change and (3) can start a threaded discussion on specific lines with the author and other reviewers, and then (4) the author can propose modifications to address reviewers' comments. This feedback cycle continues until everybody is satisfied or the patch is discarded. Different projects adapted their tools to support their process. Microsoft uses **CodeFlow**, which tracks the state of each person (author or reviewer) and where they are in the process (signed off, waiting, reviewing); CodeFlow does not prevent authors from submitting changes without approval [33] and supports chats in comment threads [4]. Google's Chromium project (along with several other OSS projects) relies on the externally-available **Gerrit** [17]; in Chromium, changes are only merged into the master branch after explicit approval from reviewers and automated verification that the change does not break the build [12]. In Gerrit, unassigned reviewers can also make comments. VMware developed the open-source **ReviewBoard**, which integrates static analysis into the review process; this integration relies on change authors manually requesting analysis and has been shown to improve code review quality [5]. Facebook's code review system, **Phabricator** [29], allows reviewers to "take over" a change and commit it themselves and provides hooks for automatic static analysis or continuous build/test integration.

In the context of tool-based reviews, researchers have investigated the relationship between code change acceptance or response time and features of the changed code and authors [41], as well as the agreement among reviewers [22]. Qualitative investigations have been also conducted to define what constitutes a good code review according to industrial [11] and OSS developers [26].

Pull-based development model. In the GitHub pull request process [18] a developer wanting to make a change forks an existing git repository and then makes changes in their fork. After a pull request has been sent out, it appears in the list of pull requests for the project in question, visible to anyone who can see the project. Gousios *et al.* qualitatively investigated work practices and challenges of pull-request integrators [21] and contributors [20], finding analogies to other tool-based code reviews.

Table 1: Convergent review practices [33].

id	Convergent Practice
CP_1	Contemporary peer review follows a lightweight, flexible process
CP_2	Reviews happen early (before a change is committed), quickly, and frequently
CP_3	Change sizes are small
CP_4	Two reviewers find an optimal number of defects
CP_5	Review has changed from a defect finding activity to a group problem solving activity

2.2 Convergent Practices in Code Review

Rigby and Bird presented the first and most significant work that tries to identify convergent practices across several code review processes and contexts [33]. They considered OSS projects that use email based reviews, OSS projects that use Gerrit, an AMD project that uses a basic code review tool, and Microsoft with CodeFlow. They analyzed the process and the data available from these projects to describe several angles, such as iterative development, reviewer selection, and review discussions. They identified five practices of modern code review to which all the considered projects converged (Table 1). We will refer to these practices using their id, *e.g.*, CP_1. Substantially, they found an agreement in terms of a quick, lightweight process (CP_1,CP_2,CP_3) with few people involved (CP_4) who conduct group problem solving (CP_5).

3 METHODOLOGY

This section describes our research questions and settings; it also outlines our research method and its limitations.

3.1 Research Questions

The overall goal of this research is to investigate modern code review at Google, which is a process that involves thousands of developers and that has been refined over more than a decade. To this aim, we conduct an exploratory investigation that we structure around three main research questions.

RQ_1: What are the motivations for code review at Google? Rigby and Bird found the motivations to be one of the converging traits of modern code review (CP_5). Here we study what motivations and expectations drive code reviews at Google. In particular, we consider both the historical reasons for introducing modern code review (since Google is one of the first companies that used modern code review) and the current expectations.

RQ_2: What is the practice of code review at Google? The other four convergent practices found by Rigby and Bird regard how the process itself is executed, in terms of flow (CP_1), speed and frequency (CP_2), size of the analyzed changes (CP_3), and number of reviewers (CP_4). We analyze these aspects at Google to investigate whether the same findings hold for a company that has a longer history of code review, an explicit culture, and larger volume of reviews compared to those analyzed in previous studies [4, 33].

RQ_3: How do Google developers perceive code review? Finally, in our last research question, we are interested in understanding how Google developers perceive modern code review as implemented in their company. This exploration is needed to better understand practices (since perceptions drive behavior [39]) and to guide future research. We focus on two aspects: *breakdowns* of the review process developers experience during specific reviews and whether developers are *satisfied* with review despite these challenges.

3.2 Research Setting

We briefly describe our research setting for context on our methodology. See Section 5.1 for a comprehensive description of Google's code review process and tooling.

Most software development at Google occurs in a monolithic source repository, accessed via an internal version control system [30]. Since code review is required at Google, every commit to Google's source control system goes first through code review using Critique: an internally developed, centralized, web-based code review tool. The development workflow in Google's monolithic repository, including the code review process, is very uniform. As with other tools described in Section 2, Critique allows reviewers to see a diff of the proposed code change and start a threaded discussion on specific lines with the author and other reviewers. Critique offers extensive logging functionalities; all developer interactions with the tool are captured (including opening the tool, viewing a diff, making comments, and approving changes).

3.3 Research Method

To answer our research questions, we follow a mixed qualitative and quantitative approach [13], which combines data from several sources: semi-structured interviews with employees involved in software development at Google, logs from the code review tool, and a survey to other employees. We use the interviews as a tool to collect data on the diversity (as opposed to frequencies [23]) of the motivations for conducting code review (RQ_1) and to elicit developers' perceptions of code review and its challenges (RQ_3). We use Critique logs to quantify and describe the current review practices (RQ_2). Finally, we use the survey to confirm the diverse motivations for code review that emerged from the interviews (RQ_1) and elicit the developers' satisfaction with the process.

Interviews. We conducted a series of face-to-face semi-structured [13] interviews with selected Google employees, each taking approximately 1 hour.The initial pool of possible participants was selected using snowball sampling, starting with developers known to the paper authors. From this pool, participants were selected to ensure a spread of teams, technical areas, job roles, length of time within the company, and role in the code review process. The interview script (in appendix [1]) included questions about perceived motivations for code review, a recently reviewed/authored change, and best/worst review experiences. Before each interview we reviewed the participant's review history and located a change to discuss in the interview; we selected these changes based on the number of interactions, the number of people involved in the conversation and whether there were comments that seemed surprising. In an observation part of the interview, the participant was asked to think-aloud while reviewing the pending change and to provide some explicit information, such as the entry point for starting the review. The interviews continued until saturation [19] was achieved and interviews were bringing up broadly similar concepts. Overall, we conducted 12 interviews with staff who had been working at Google from 1 month to 10 years (median 5 years), in Software Engineering and Site Reliability Engineering. They included technical leads, managers and individual contributors. Each interview involved three to four people: The participant and 2-3 interviewees (two of which are authors of this paper). Interviews were live-transcribed by one of the interviewers while another one asked questions.

Open Coding on Interview Data. To identify the broad themes emerging from the interview data we performed an open coding pass [27]. Interview transcripts were discussed by two authors to establish common themes, then converted into a coding scheme. An additional author then performed a closed coding of the notes of the discussions to validate the themes. We iterated this process over one of the interviews until we had agreement on the scheme. We also tracked in what context (relationship between reviewer and author) these themes were mentioned. The combination of the design of the questions and the analysis process means that we can discuss stable themes in the results, but cannot meaningfully discuss relative frequencies of occurrence [23].

Analysis of Review Data. We analyzed quantitative data about the code review process by using the logs produced by Critique. We mainly focus on metrics related to convergent practices found by Rigby and Bird [33]. To allow for comparison, we do not consider changes without any reviewers, as we are interested in changes that have gone through an explicit code review process. We consider a "reviewer" to be any user who approves a code change, regardless of whether they were explicitly asked for a review by the change author. We use a name-based heuristic to filter out changes made by automated processes. We focus exclusively on changes that occur in the main codebase at Google. We also exclude changes not yet committed at the time of study and those for which our diff tool reports a delta of zero source lines changed, *e.g.*, a change that only modifies binary files.
On an average workday at Google, about 20,000 changes are committed that meet the filter criteria described above. Our final dataset includes the approximately 9 million changes created by more than 25,000 authors and reviewers from January 2014 until July 2016 that meet these criteria, and about 13 million comments collected from all changes between September 2014 and July 2016.

Survey. We created an online questionnaire that we sent to 98 engineers who recently submitted a code change. The code change had already been reviewed, so we customized the questionnaire to ask respondents about how they perceived the code review for *their specific recent change*; this strategy allowed us to mitigate recall bias [35], yet collect comprehensive data. The survey consisted of three Likert scale questions on the value of the received review, one multiple choice on the effects of the review on their change (based on the expectations that emerged from the interviews) with an optional 'other' response, and one open-ended question eliciting the respondent's opinion on the received review, the code review tool, and/or the process in general. We received 44 valid responses to our survey (45% response rate, which is considered high for software engineering research [31]).

3.4 Threats to Validity and Limitations

We describe threats to validity and limitations of the results of our work, as posed by our research method, and the actions that we took to mitigate them.

Internal validity - Credibility. Concerning the quantitative analysis of review data, we use heuristics to filter out robot-authored changes from our quantitative analysis, but these heuristics may allow some robot authored changes; we mitigated this as we only include robot-authored changes that have a human reviewer. Concerning the qualitative investigation, we used open coding to analyze the interviewees' answers. This coding could have been influenced by the experience and motivations of the authors conducting it, although we tried to mitigate this bias by involving multiple coders. The employees that decided to participate in our interviews and the survey freely decided to do so, thus introducing the risk of *self-selection bias*. For this reason, results may have been different for developers who would not choose to participate; to try to mitigate this issue, we combine information from both interviews and survey. Moreover, we used a snowball sampling method to identify engineers to interview, this is at the risk of *sampling bias*. Although we attempted to mitigate this risk by interviewing developers with a range of job roles and responsibilities, there may be other factors the developers we interviewed share that would not apply across the company. To mitigate *moderator acceptance bias*, the researchers involved in the qualitative data collection were not part of the Critique team. Social desirability bias may have influenced the answers to align more favorably to Google culture; however, at Google people are encouraged to criticize and improve broken workflows when discovered, thus reducing this bias. Finally, we did not interview research scientists or developers that interact with specialist reviewers (such as security reviews), thus our results are biased towards general developers.

Generalizability - Transferability. We designed our study with the stated aim of understanding modern code review within a specific company. For this reason, our results may not generalize to other contexts, rather we are interested in the diversity of the practices and breakdowns that are still occurring after years and millions of reviews of refinement. Given the similarity of the underlying code review mechanism across several companies and OSS projects, it is reasonable to think that should a review process reach the same level of maturity and use comparable tooling, developers would have similar experiences.

4 RESULTS: MOTIVATIONS

In our first research question, we seek to understand motivations and expectations of developers when conducting code review at Google, starting by investigating what led to the introduction of this process in the first place.

4.1 How it All Started

Code review at Google was introduced early on by one of the first employees; the first author of this paper interviewed this employee (referred to as E in the following) to better understand the initial motivation of code review and its evolution. E explained that the main impetus behind the introduction of code review was *to force developers to write code that other developers could understand*; this was deemed important since code must act as a teacher for future developers. E stated that the introduction of code review at Google signaled the transition from a research codebase, which is optimized towards quick prototyping, to a production codebase, where it is critical to think about future engineers reading source code. Code review was also perceived as capable of ensuring that more than one person would be familiar with each piece of code, thus increasing the chances of knowledge staying within the company.

E reiterated on the concept that, although it is great if reviewers find bugs, the foremost reason for introducing code review at Google was to improve code understandability and

maintainability. However, in addition to the initial educational motivation for code review, *E* explained that three additional benefits soon became clear to developers internally: checking the consistency of style and design; ensuring adequate tests; and improving security by making sure no single developer can commit arbitrary code without oversight.

4.2 Current expectations

By coding our interview data, we identified four key themes for what Google developers expect from code reviews: **education**, **maintaining norms**, **gatekeeping**, and **accident prevention**. Education regards either teaching or learning from a code review and is in line with the initial reasons for introducing code review; norms refer to an organization preference for a discretionary choice (*e.g.*, formatting or API usage patterns); gatekeeping concerns the establishment and maintenance of boundaries around source code, design choices or another artifact; and accidents refer to the introduction of bugs, defects or other quality related issues.

These are the main themes *during* the review process, but code review is also used *retrospectively* for **tracking history**. Developers value the review process after it has taken place; code review enables the ability to browse the history of a code change, including what comments occurred and how the change evolved. We also noticed developers using code review history to understand how bugs were introduced. Essentially, code review enables future auditing of changes.

In our survey, we further validated this coding scheme. We asked what authors found valuable about code review comments; they could select one or more of the four themes and/or write in their own. Each of the four themes identified earlier was selected by between 8 and 11 respondents in the context of a particular code review, thus providing additional confidence that the coding scheme described above aligns with developers' perception of the value of the code review.

Although these expectations can map over those found previously at Microsoft [4], the main focus at Google, as explained by our participants, is on education as well as code readability and understandability, in line with the historical impetus for review. For this reason, the focus does not align with that found by Rigby and Bird (*i.e.*, a group problem solving activity) [33].

> ***Finding 1***. *Expectations for code review at Google do not center around problem solving. Reviewing was introduced at Google to ensure code readability and maintainability. Today's developers also perceive this educational aspect, in addition to maintaining norms, tracking history, gatekeeping, and accident prevention. Defect finding is welcomed but not the only focus.*

As stated earlier, when coding interview transcripts we also tracked the review context in which a theme was mentioned. We found that the relative importance of these different themes depends on the relationship between the author and reviewer (Figure 1). For example, maintaining norms came up between an engineer and those with a different seniority (project leads, expert readability reviewers or "new" team members) but less so with their peers or other teams, where instead gatekeeping and accident prevention are primary. The educational benefits are broadly valued and encompass several different relationships.

Figure 1: Relationship diagram describing which themes of review expectations appeared primarily within a particular author/reviewer context.

> ***Finding 2***. *Expectations about a specific code review at Google depend on the work relationship between the author and reviewers.*

5 RESULTS: PRACTICES

In our second research question, we describe the code review process and compare its quantitative aspects with the convergent practices found in previous work [33].

5.1 Describing The Review Process

The code review at Google is linked to two concepts: *ownership* and *readability*. We first introduce them, then we describe the flow of the review process, and we conclude with the distinct characteristics of the internal review tool, Critique, that serve in contrast with other review tools.

Ownership. The Google codebase is arranged in a tree structure, where each directory is explicitly *owned* by a set of people. Although any developer can propose a change to any part of the codebase, an owner of the directory in question (or a parent directory) must review and approve the change before it is committed; even directory owners are expected to have their code reviewed before committing.

Readability. Google defines a concept called *readability*, which was introduced very early on to ensure consistent code style and norms within the codebase. Developers can gain readability certification in a particular language. To apply for readability, a developer sends changes to a set of readability reviewers; once those reviewers are confident the developer understands the code style and best practices for a language, the developer is granted readability for that language. Every change must be either authored or reviewed by someone with a readability certification for the language(s) used.

Code Review Flow. The flow of a review is tightly coupled with Critique and works as follows:

1. *Creating:* Authors start modifying, adding, or deleting some code; once ready, they create a change.
2. *Previewing:* Authors then use Critique to view the diff of the change and view the results of automatic code analyzers (e.g. from Tricorder [36]). When they are ready, authors mail the change out to one or more reviewers.
3. *Commenting:* Reviewers view the diff in the web UI, drafting comments as they go. Program analysis results, if present, are also visible to reviewers. *Unresolved* comments represent action items for the change author to definitely address. *Resolved* comments include optional or informational comments that may not require any action by a change author.
4. *Addressing Feedback:* The author now addresses comments, either by updating the change or by replying to comments. When a change has been updated, the author uploads a new snapshot. The author and reviewers can look at diffs between any pairs of snapshots to see what changed.
5. *Approving:* Once all comments have been addressed, the reviewers approve the change and mark it 'LGTM' (Looks Good To Me). To eventually commit a change, a developer typically must have approval from at least one reviewer. Usually, only one reviewer is required to satisfy the aforementioned requirements of ownership and readability.

We attempt to quantify how "lightweight" reviews are (CP_1). We measure how much back-and-forth there is in a review, by examining how many times a change author mails out a set of comments that resolve previously unresolved comments. We make the assumption that one iteration corresponds to one instance of the author resolving some comments; zero iterations means that the author can commit immediately. We find that over 80% of all changes involve at most one iteration of resolving comments.

Suggesting Reviewers. To identify the best person to review a change, Critique relies on a tool that analyzes the change and suggests possible reviewers. This tool identifies the smallest set of reviewers needed to fulfill the review requirements for all files in a change. Note that often only one reviewer is required since changes are often authored by someone with ownership and/or readability for the files in question. This tool prioritizes reviewers that have recently edited and/or reviewed the included files. New team members are explicitly added as reviewers since they have not yet built up reviewing/editing history. Unassigned reviewers can also make comments on (and can potentially approve) changes. Tool support for finding reviewers is typically only necessary for changes to files beyond those for a particular team. Within a team, developers know who to send changes to. For changes that could be sent to anyone on the team, many teams use a system that assigns reviews sent to the team email address to configured team members in a round-robin manner, taking into account review load and vacations.

Code Analysis Results. Critique shows code analysis results as comments along with human ones (although in different colors). Analyzers (or reviewers) can provide suggested edits, which can be both proposed and applied to the change via Critique. To vet changes before they are committed, development at Google also includes pre-commit hooks: checks where a failure requires an explicit override by the developer to enable a commit. Pre-commit checks include things like basic automated style checking and running automated test suites related to a change. The results of all pre-commit checks are visible in the code review tool. Typically, pre-commit checks are automatically triggered. These checks are configurable such that teams can enforce project-specific invariants and automatically add email lists to changes to raise awareness and transparency. In addition to pre-commit results, Critique displays the result of a variety of automated code analyses through Tricorder [36] that may not block committing a change. Analysis results encompass simple style checks, more complicated compiler-based analysis passes and also project-specific checks. Currently, Tricorder includes 110 analyzers, 5 of which are plugin systems for hundreds of additional checks, in total analyzing more than 30 languages.

> ***Finding 3***. *The Google code review process is aligned with the convergent practice of being lightweight and flexible. In contrast to other studied systems, however, ownership and readability are explicit and play a key role. The review tool includes reviewer recommendation and code analysis results.*

5.2 Quantifying The Review Process

We replicate quantitative analysis that Rigby and Bird conducted that led to the discovery CP_{2-4} so as to compare these practices to the traits that Google has converged to.

Review frequency and speed. Rigby and Bird found that fast-paced, iterative development also applies to modern code review: In their projects, developers work in remarkably consistent short intervals. To find this, they analyzed review frequency and speed.

At Google, in terms of frequency, we find that the median developer authors about 3 changes a week, and 80 percent of authors make fewer than 7 changes a week. Similarly, the median for changes reviewed by developers per week is 4, and 80 percent of reviewers review fewer than 10 changes a week. In terms of speed, we find that developers have to wait for initial feedback on their change a median time of under an hour for small changes and about 5 hours for very large changes. The overall (all code sizes) median latency for the *entire* review process is under 4 hours. This is significantly lower than the median time to approval reported by Rigby and Bird [33], which is 17.5 hours for AMD, 15.7 hours for Chrome OS and 14.7, 19.8, and 18.9 hours for the three Microsoft projects. Another study found the median time to approval at Microsoft to be 24 hours [14].

Review size. Rigby and Bird argued that quick review time could only be enabled by smaller changes to review and subsequently analyzed review sizes. At Google, over 35% of the changes under consideration modify only a single file and

about 90% modify fewer than 10 files. Over 10% of changes modify only a single line of code, and the median number of lines modified is 24. The median change size is significantly lower than reported by Rigby and Bird for companies such as AMD (44 lines), Lucent (263 lines), and Bing, Office and SQL Server at Microsoft (somewhere between those boundaries), but in line for change sizes in open source projects [33].

Number of reviewers and comments. The optimal number of reviewers has been controversial among researchers, even in the deeply investigated code inspections [37]. Rigby and Bird investigated whether the considered projects converged to a similar number of involved reviewers. They found this number to be two, remarkably regardless of whether the reviewers were explicitly invited (such as in Microsoft projects, who invited a median of up to 4 reviewers) or whether the change was openly broadcasted for review [33].

At Google, by contrast, fewer than 25% of changes have more than one reviewer,[1] and over 99% have at most five reviewers with a median reviewer count of 1. Larger changes tend to have more reviewers on average. However, even very large changes on average require fewer than two reviewers.

Rigby and Bird also found a "minimal increase in the number of comments about the change when more [than 2] reviewers were active" [33], and concluded that two reviewers find an optimal number of defects. At Google, the situation is different: A greater number of reviewers results in a greater average number of comments on a change. Moreover, the average number of comments per change grows with the number of lines changed, reaching a peak of 12.5 comments per change for changes of about 1250 lines. Changes larger than this often contain auto-generated code or large deletions, resulting in a lower average number of comments.

> ***Finding 4.*** *Code review at Google has converged to a process with markedly quicker reviews and smaller changes, compared to the other projects previously investigated. Moreover, one reviewer is often deemed as sufficient, compared to two in the other projects.*

6 RESULTS: DEVELOPERS' PERCEPTIONS

Our last research question investigates challenges and satisfaction of developers with code review at Google.

6.1 Code Review Breakdowns at Google

Previous studies investigated challenges in the review process overall [4, 26] and provided compelling evidence, also corroborated by our experience as engineers, that understanding code to review is a major hurdle. To broaden our empirical body of knowledge, we focus here on challenges encountered in specific reviews ("breakdowns"), such as delays or disagreements.

[1] 30% of changes have comments by more than one commenter, meaning that about 5% of changes have additional comments from someone that did not act as an approver for the change.

Five main themes emerged from the analysis of our interview data. The first four themes regard breakdowns concerning aspects of the *process*:

Distance: Interviewees perceive distance in code review from two perspectives: geographical (*i.e.*, the physical distance between the author and reviewers) and organizational (*e.g.*, between different teams or different roles). Both these types of distance are perceived as the cause of delays in the review process or as leading to misunderstandings.

Social interactions: Interviewees perceive the communication within code review as possibily leading to problems from two aspects: *tone* and *power*. Tone refers to the fact that sometimes authors are sensitive to review comments; sentiment analysis on comments has provided evidence that comments with negative tone are less likely to be useful [11]. Power refers to using the code review process to induce another person to change their behavior; for example, dragging out reviews or withholding approvals. Issues with tone or power in reviews can make developers uncomfortable or frustrated with the review process.

Review subject: The interviews referenced disagreements as to whether code review was the most suitable context for reviewing certain aspects, particularly *design reviews*. This resulted in mismatched expectations (*e.g.*, some teams wanted most of the design to be complete before the first review, others wanted to discuss design in the review), which could cause friction among participants and within the process.

Context: Interviewees allowed us to see that misunderstandings can arise based on not knowing what gave rise to the change; for example, if the rationale for a change was an urgent fix to a production problem or a "nice to have" improvement. The resulting mismatch in expectations can result in delays or frustration.

The last theme regarded the *tool* itself:

Customization: Some teams have different requirements for code review, *e.g.*, concerning how many reviewers are required. This is a technical breakdown, since arbitrary customizations are not always supported in Critique, and may cause misunderstandings around these policies. Based on feedback, Critique recently released a new feature that allows change authors to require that all reviewers sign off.

6.2 Satisfaction and Time Invested

To understand the significance of the identified concerns, we used part of the survey to investigate whether code review is overall perceived as valuable.

We find (Table 2) that code review is broadly seen as being valuable and efficient within Google – all respondents agreed with the statement that code review is valuable. This sentiment is echoed by internal satisfaction surveys we conducted on Critique: 97% of developers are satisfied with it.

Within the context of a specific change, the sentiment is more varied. The least satisfied replies were associated with changes that were very small (1 word or 2 lines) or with changes which were necessary to achieve some other goal, *e.g.*, triggering a process from a change in the source code.

	Strongly disagree				Strongly agree
For this change, the review process was a good use of my time	2	4	14	11	13
Overall, I find code review at Google valuable	0	0	0	14	30
	Too little				Too much
For this change, the amount of feedback was	2	2	34	5	0

Table 2: User satisfaction survey results

However, the majority of respondents felt that the amount of feedback for their change was appropriate. 8 respondents described the comments as not being helpful, of these 3 provided more detail, stating that the changes under review were small configuration changes for which code review had less impact. Only 2 respondents said the comments had found a bug.

To contextualize the answers in terms of satisfaction, we also investigate the time developers spend reviewing code. To accurately quantify the reviewer time spent, we tracked developer interactions with Critique (*e.g.*, opened a tab, viewed a diff, commented, approved a change) as well as other tools to estimate how long developers spend reviewing code per week. We group sequences of developer interactions into blocks of time, considering a "review session" to be a sequence of interactions related to the same uncommitted change, by a developer other than the change author, with no more than 10 minutes between each successive interaction. We sum up the total number of hours spent across all review sessions in the five weeks that started in October 2016 and then calculate the average per user per week, filtering out users for whom we do not have data for all 5 weeks. We find that developers spend an average of 3.2 (median 2.6 hours a week) reviewing changes. This is low compared to the 6.4 hours/week of self-reported time for OSS projects [10].

Finding 5. *Despite years of refinement, code review at Google still faces breakdowns. These are mostly linked to the complexity of the interactions that occur around the reviews. Yet, code review is strongly considered a valuable process by developers, who spend around 3 hours a week reviewing.*

7 DISCUSSION

We discuss the themes that have emerged from this investigation, which can inspire practitioners in the setting up of their code review process and researchers in future investigations.

7.1 A Truly Lightweight Process

Modern code review was born as a lighter weight alternative to the cumbersome code inspections [4]; indeed Rigby and Bird confirmed this trait (CP_1) in their investigation across systems [33]. At Google, code review has converged to an even more lightweight process, which developers find both valuable and a good use of their time.

Median review times at Google are much shorter than in other projects [14, 33, 34]. We postulate that these differences are due to the culture of Google on code review (strict reviewing standards and expectations around quick turnaround times for review). Moreover, there is a significant difference with reviewer counts (one at Google vs. two in most other projects); we posit that having one reviewer helps make reviews fast and lightweight.

Both low review times and reviewer counts may result from code review being a required part of the developer workflow; they can also result from small changes. The median change size in OSS projects varies from 11 to 32 lines changed [34], depending on the project. At companies, this change size is typically larger [33], sometimes as high as 263 lines. We find that change size at Google more closely matches OSS: most changes are small. The size distribution of changes is an important factor in the quality of the code review process. Previous studies have found that the number of useful comments decreases [11, 14] and the review latency increases [8, 24] as the size of the change increases. Size also influences developers' perception of the code review process; a survey of Mozilla contributors found that developers feel that size-related factors have the greatest effect on review latency [26]. A correlation between change size and review quality is acknowledged by Google and developers are strongly encouraged to make small, incremental changes (with the exception of large deletions and automated refactoring). These findings and our study support the value of reviewing small changes and the need for research and tools to help developers create such small, self-contained code changes for review [6, 15].

7.2 Software Engineering Research in Practice

Some of the practices that are part of code review at Google are aligned with practices proposed in software engineering research. For example, a study of code ownership at Microsoft found that changes made by minor contributors should be received with more scrutiny to improve code quality [9]; we found that this concept is enforced at Google through the requirement of an approval from an owner. Also, previous research has shown that typically one reviewer for a change will take on the task of checking whether code matches conventions [4]; readability makes this process more explicit. In the following, we focus on the features of Critique that make it a '*next generation code review tool*' [26].

Reviewer recommendation. Researchers found that reviewers with prior knowledge of the code under review give more useful comments [4, 11], thus tools can add support for reviewer selection [11, 26]. We have seen that reviewer recommendation is tool-supported, prioritizing those who recently

edited/reviewed the files under review. This corroborates recent research that frequent reviewers make large contributions to the evolution of modules and should be included along with frequent editors [40]. Detecting the right reviewer does not seem problematic in practice at Google, in fact the model of recommendation implemented is straightforward since it can programmatically identify owners. This is in contrast with other proposed tools that identify reviewers who have reviewed files with similar names [43] or take into account such features as the number of comments included in reviews [42]. At Google, there is also a focus on dealing with reviewers' workload and temporary absences (in line with a study at Microsoft [4]).

Static Analysis integration. A qualitative study of 88 Mozilla developers [26] found that static analysis integration was the most commonly-requested feature for code review. Automated analyses allow reviewers to focus on the understandability and maintainability of changes, instead of getting distracted by trivial comments (*e.g.*, about formatting). Our investigation at Google showed us the practical implications of having static analysis integration in a code review tool. Critique integrates feedback channels for analysis writers [36]: Reviewers have the option to click "Please fix" on an analysis-generated comment as a signal that the author should fix the issue, and either authors or reviewers can click "Not useful" in order to flag an analysis result that is not helpful in the review process. Analyzers with high "Not useful" click rates are fixed or disabled. We found that this feedback loop is critical for maintaining developer trust in the analysis results.

A Review Tool Beyond Collaborative Review. Finally, we found strong evidence that Critique's uses extend beyond reviewing code. Change authors use Critique to examine diffs and browse analysis tool results. In some cases, code review is part of the development process of a change: a reviewer may send out an unfinished change in order to decide how to finish the implementation. Moreover, developers also use Critique to examine the history of submitted changes long after those changes have been approved; this is aligned with what Sutherland and Venolia envisioned as a beneficial use of code review data for development [38]. Future work can investigate these unexpected and potentially impactful non-review uses of code review tools.

7.3 Knowledge spreading

Knowledge transfer is a theme that emerged in the work by Rigby and Bird [33]. In an attempt to measure knowledge transfer due to code review, they built off of prior work that measured expertise in terms of the number of files changed [28], by measuring the number of distinct files changed, reviewed, and the union of those two sets. They find that developers know about more files due to code review.

At Google, knowledge transfer is part of the educational motivation for code review. We attempted to quantify this effect by looking at comments and files edited/reviewed. As developers build experience working at Google, the average number of comments on their changes decreases (Figure 2).

Figure 2: Reviewer comments vs. author's tenure at Google

Figure 3: The number of distinct files seen (edited or reviewed, or both) by a full-time employee over time.

Developers at Google who have started within the past year typically have more than twice as many comments per change. Prior work found that authors considered comments with questions from the reviewer as not useful, and the number of not useful comments decreases with experience [11]. We postulate that this decrease in commenting is a result of reviewers needing to ask fewer questions as they build familiarity with the codebase and corroborates the hypothesis that the educational aspect of code review may pay off over time. Also, we can see that the number of distinct files edited and reviewed by engineers at Google, and the union of those two sets, increase with seniority (Figure 3) and the total number of files seen is clearly larger than the number of files edited. For this graph, we bucket our developers by how long they have been at the company (in 3-month increments) and then compute the number of files they have edited and reviewed. It would be interesting in future work to better understand how reviewing files impacts developer fluency [44].

8 CONCLUSION

Our study found code review is an important aspect of the development workflow at Google. Developers in all roles see it as providing multiple benefits and a context where developers can teach each other about the codebase, maintain

the integrity of their teams' codebases, and build, establish, and evolve norms that ensure readability and consistency of the codebase. Developers reported they were happy with the requirement to review code. The majority of changes are small, have one reviewer and no comments other than the authorization to commit. During the week, 70% of changes are committed less than 24 hours after they are mailed out for an initial review. These characteristics make code review at Google lighter weight than the other projects adopting a similar process. Moreover, we found that Google includes several research ideas in its practice, making the practical implications of current research trends visible.

ACKNOWLEDGMENTS

The authors would like to thank Elizabeth Kemp for her assistance with the qualitative study, Craig Silverstein for context, and past and present members of the Critique team for feedback, including in particular Ben Rohlfs, Ilham Kurnia, and Kate Fitzpatrick. A. Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] 2017. Appendix to this paper. https://goo.gl/dP4ekg. (2017).
[2] A.F. Ackerman, L.S. Buchwald, and F.H. Lewski. 1989. Software inspections: An effective verification process. *IEEE Software* 6, 3 (1989), 31–36.
[3] A.F. Ackerman, P.J. Fowler, and R.G. Ebenau. 1984. Software inspections and the industrial production of software. In *Symposium on Software validation: inspection-testing-verification-alternatives*.
[4] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *ICSE*.
[5] Vipin Balachandran. 2013. Reducing Human Effort and Improving Quality in Peer Code Reviews Using Automatic Static Analysis and Reviewer Recommendation. In *ICSE*.
[6] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In *ICSE*.
[7] V.R. Basili, F. Shull, and F. Lanubile. 1999. Building knowledge through families of experiments. *IEEE Trans. on Software Eng.* 25, 4 (1999), 456–473.
[8] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. 2013. The influence of non-technical factors on code review. In *WCRE*.
[9] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don't touch my code!: examining the effects of ownership on software quality. In *FSE*.
[10] Amiangshu Bosu and Jeffrey C Carver. 2013. Impact of peer code review on peer impression formation: A survey. In *ESEM*.
[11] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of useful code reviews: an empirical study at Microsoft. In *MSR*.
[12] chromium 2016. Chromium developer guidelines. https://www.chromium.org/developers/contributing-code. (August 2016).
[13] J.W. Creswell. 2009. *Research design: Qualitative, quantitative, and mixed methods approaches* (3rd ed.). Sage Publications.
[14] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. 2015. Code Reviews Do Not Find Bugs: How the Current Code Review Best Practice Slows Us Down. In *ICSE (SEIP)*.
[15] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *SANER*.
[16] M.E. Fagan. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15, 3 (1976), 182–211.
[17] gerrit 2016. https://www.gerritcodereview.com/. (August 2016).
[18] githubpull 2016. GitHub pull request process. https://help.github.com/articles/using-pull-requests/. (2016).
[19] Barney G Glaser and Anselm L Strauss. 2009. *The discovery of grounded theory: Strategies for qualitative research*. Transaction Books.
[20] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work Practices and Challenges in Pull-Based Development: The Contributor's Perspective. In *ICSE*.
[21] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. 2015. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *ICSE*.
[22] Toshiki Hirao, Akinori Ihara, Yuki Ueda, Passakorn Phannachitta, and Ken-ichi Matsumoto. 2016. The Impact of a Low Level of Agreement Among Reviewers in a Code Review Process. In *IFIP International Conference on Open Source Systems*.
[23] Harrie Jansen. 2010. The logic of qualitative survey research and its position in the field of social research methods. In *Forum Qualitative Sozialforschung*, Vol. 11.
[24] Yujuan Jiang, Bram Adams, and Daniel M German. 2013. Will my patch make it? and how fast?: Case study on the linux kernel. In *MSR*.
[25] Sami Kollanus and Jussi Koskinen. 2009. Survey of software inspection research. *The Open Software Engineering Journal* 3, 1 (2009), 15–34.
[26] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. 2016. Code review quality: How developers see it. In *ICSE*.
[27] Matthew B Miles and A Michael Huberman. 1994. *Qualitative data analysis: An expanded sourcebook*. Sage.
[28] Audris Mockus and James D Herbsleb. 2002. Expertise browser: a quantitative approach to identifying expertise. In *ICSE*.
[29] phabricator 2016. Meet Phabricator, The Witty Code Review Tool Built Inside Facebook. https://techcrunch.com/2011/08/07/oh-what-noble-scribe-hath-penned-these-words/. (August 2016).
[30] Rachel Potvin and Josh Levenburg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* (2016).
[31] Teade Punter, Marcus Ciolkowski, Bernd Freimut, and Isabel John. 2003. Conducting on-line surveys in software engineering. In *Empirical Software Engineering*.
[32] P. Rigby, B. Cleary, F. Painchaud, M.A. Storey, and D. German. 2012. Open Source Peer Review–Lessons and Recommendations for Closed Source. *IEEE Software* (2012).
[33] Peter C Rigby and Christian Bird. 2013. Convergent software peer review practices. In *FSE*.
[34] Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. 2014. Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory. *IEEE Transactions on Software Engineering* 23, 4, Article 35 (2014).
[35] David L Sackett. 1979. Bias in analytic research. *Journal of chronic diseases* 32, 1-2 (1979), 51–63.
[36] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: building a program analysis ecosystem. In *ICSE*.
[37] F. Shull and C. Seaman. 2008. Inspecting the History of Inspections: An Example of Evidence-Based Technology Diffusion. *IEEE Software* 25, 1 (2008), 88–90.
[38] Andrew Sutherland and Gina Venolia. 2009. Can peer code reviews be exploited for later information needs?. In *ICSE*.
[39] William Isaac Thomas and Dorothy Swaine Thomas. 1928. The methodology of behavior study. *The child in America: Behavior problems and programs* (1928), 553–576.
[40] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *ICSE*.
[41] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2017. Review participation in modern code review. *Empirical Software Engineering* 22, 2 (2017), 768–817.
[42] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *SANER*.
[43] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2016. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering* 42, 6 (2016), 530–543.
[44] Minghui Zhou and Audris Mockus. 2010. Developer Fluency: Achieving True Mastery in Software Projects. In *FSE*.

Expectations, Outcomes, and Challenges Of Modern Code Review

Alberto Bacchelli
REVEAL @ Faculty of Informatics
University of Lugano, Switzerland
alberto.bacchelli@usi.ch

Christian Bird
Microsoft Research
Redmond, Washington, USA
cbird@microsoft.com

***Abstract*—Code review is a common software engineering practice employed both in open source and industrial contexts. Review today is less formal and more "lightweight" than the code inspections performed and studied in the 70s and 80s. We empirically explore the motivations, challenges, and outcomes of tool-based code reviews. We observed, interviewed, and surveyed developers and managers and manually classified hundreds of review comments across diverse teams at Microsoft. Our study reveals that while finding defects remains the main motivation for review, reviews are less about defects than expected and instead provide additional benefits such as knowledge transfer, increased team awareness, and creation of alternative solutions to problems. Moreover, we find that code and change understanding is the key aspect of code reviewing and that developers employ a wide range of mechanisms to meet their understanding needs, most of which are not met by current tools. We provide recommendations for practitioners and researchers.**

I. Introduction

Peer code review, a manual inspection of source code by developers other than the author, is recognized as a valuable tool for reducing software defects and improving the quality of software projects [1] [2]. In 1976, Fagan formalized a highly structured process for code reviewing [3], based on line-by-line group reviews, done in extended meetings—*code inspections*. Over the years, researchers provided evidence on code inspection's benefits, especially in terms of defect finding, but the cumbersome, time-consuming, and synchronous nature of this approach hinders its universal adoption in practice [4].

Nowadays, many organizations are adopting more lightweight code review practices to limit the inefficiencies of inspections. In particular, there is a clear trend toward the usage of tools specifically developed to support code review [5]. In the context of this paper, we define *Modern Code Review*, as review that is (1) informal (in contrast to Fagan-style), (2) tool-based, and that (3) occurs regularly in practice nowadays, for example at companies such as Microsoft, Google [6], Facebook [7], and in other companies and OSS projects [8].

This trend raises questions, such as: Can we apply the lessons learned from previous research on code inspections to modern code reviews? What are the expectations for code review nowadays? What are the actual outcomes of code review? What challenges do people face in code review?

Answers to these questions can provide insight for both practitioners and researchers. Developers and other software project stakeholders can use empirical evidence about expectations and outcomes to make informed decisions about when to use code review and how it should fit into their development process. Researchers can focus their attention on practitioners' challenges to make code review more effective.

We present an in-depth study of practices in teams that use modern code review, revealing what practitioners think, do, and achieve when it comes to modern code review.

Since Microsoft is made up of many different teams working on very diverse products, it gives the opportunity to study teams performing code review *in situ* and understand their expectations, the benefits they derive from code review, the needs they have, and the problems they face.

We set up our study as an explorative investigation. We started without *a priori* hypotheses regarding how and why code review should be performed, with the aim of discovering what developers and managers expect from code review, how reviews are conducted in practice, and what the actual outcomes and challenges are. To that end, we (1) observed 17 industrial developers performing code review with various degrees of experience and seniority across 16 separate product teams with distinct reviewing cultures and policies; (2) interviewed these developers using a semi-structured interviews; (3) manually inspected and classified the content of 570 comments in discussions contained within code reviews; and (4) surveyed 165 managers and 873 programmers.

Our results show that, although the top motivation driving code reviews is still finding defects, the practice and the actual outcomes are less about finding errors than expected: Defect related comments comprise a small proportion and mainly cover small logical low-level issues. At the same time, code review additionally provides a wide spectrum of benefits to software teams, such as knowledge transfer, team awareness, and improved solutions to problems. Moreover, we found that context and change understanding is the key of any review. According to the outcomes they want to achieve, developers employ many mechanisms to fulfill their understanding needs, most of which are not currently met by any code review tool.

This paper makes the following contributions:

- Characterizing the motivations of developers and managers for code review and compare with actual outcomes.
- Relating the outcomes to understanding needs and discuss how developers achieve such needs.

Based on our findings, we provide recommendations for practitioners and implications for researchers as well as outline future avenues for research.

II. Related Work

Previous studies exist that have examined the practices of code inspection and code review. Stein et al. conducted a study focusing specifically on distributed, asynchronous code inspections [17]. The study included evaluation of a tool that allowed for identification and sharing of code faults or defects. Participants at separated locations can then discuss faults via the tool. Laitenburger conducted a survey of code inspection methods, and presented a taxonomy of code inspection techniques [9]. Johnson conducted an investigation into code review practices in open source development and the effect they have on choices made by software project managers [10].

Porter et al. [11] reported on a review of studies on code inspection in 1995 that examined the effects of factors such as team size, type of review, and number of sessions on code inspections. They also assessed costs and benefits across a number of studies. These studies differ from ours in that they were not tool-based and were the majority involved planned meetings to discuss the code.

However, prior research also sheds light on why review today is more often tool-based, informal, and often asynchronous. The current state of code review might be due to the time required for more formal inspections. Votta found that 20% of the interval in a "traditional inspection" is wasted due to scheduling [12]. The ICICLE tool [13], or "Intelligent Code Inspection in a C Language Environment," was developed after researchers at Bellcore observed how much time and work was expended before and during formal code inspections. Many of today's review tools are based on ideas that originated in ICICLE. Other similar tools have been developed in an effort to reduce time for inspection and allow asynchronous work on reviews. Examples include CAIS [14] and Scrutiny [15].

More recently, Rigby has done extensive work examining code review practices in open source software development [5]. For example in a study of practices in the Apache project [16] they data-mined the email archives and found that reviews were typically small and frequent, and that the contributions to a review were often brief and independent from one another.

Sutherland and Venolia conducted a study at Microsoft regarding using code review data for later information needs [17]. They hypothesized that the knowledge exchanged during code reviews could be of great value to engineers later trying to understand or modify the discussed code. They found that "the meat of the code review dialog, no matter what medium, is the articulation of design rationale" and, thus, "code reviews are an enticing opportunity for capturing design rationale."

When studying developer work habits, Latoza *et al.* found that many problems encountered by developers were related to understanding the rationale behind code changes and gathering knowledge from other members of their team [18].

III. Methodology

In this section we define the research questions, describe the research settings, and outline our research method.

A. Research Questions

Our investigation of code review revolves around the following research questions, which we iteratively refined during our initial in-field observations and interviews:

1. What are the motivations and expectations for modern code review? Do they change from managers to developers and testers?
2. What are the actual outcomes of modern code review? Do they match the expectations?
3. What are the main challenges experienced when performing modern code reviews relative to the expectations and outcomes?

B. Research Setting

Our study took place with professional developers, testers, and managers. Microsoft develops software in diverse domains, from high end server enterprise data management solutions such as SQL Server to mobile phone applications and smart phone apps to search engines. Each team has its own development culture and code review policies. Over the past two years, a common tool for code review at Microsoft has achieved wide-spread adoption. As it represents a common and growing solution for code review (over 40,000 developers used it so far), we focused on developers using this tool for code review—*CodeFlow*.

CodeFlow is a collaborative code review tool that allows users to directly annotate source code in its viewer and interact with review participants in a live chat model. The functionality of CodeFlow is similar to other review tools such Google's Mondrian [6], Facebook's Phabricator [7] or open-source Gerrit [8]. Developers who want their code to be reviewed create a package with the changed (new, deleted, and modified) files, select the reviewers, write a message to describe the code review, and submit everything to the CodeFlow service. CodeFlow then notifies the reviewers about the incoming task via email.

Once reviewers open a CodeFlow review, they interact with it via a single desktop window (Figure 1). On the top left **(1)**, they see the list of files changed in the current submission, plus a "description.txt" file, which contains a textual explanation of the change, written by the author. On bottom left, CodeFlow shows the list of reviewers and their status **(2)**. We see that Christian is the review author and Alberto, Tom, and Nachi are the reviewers. Alberto has reviewed and is waiting for the author to act, as the clock icon suggests, while Nachi already signed off on the changes. CodeFlow's main view **(3)** shows the diff-highlighted content of the file currently under review. Both the reviewers and the author can highlight portions of the code and add comments inline **(4)**. These comments can start threads of discussion and are the interaction points for the people involved in the review. Each user viewing the same review in CodeFlow sees events as they happen. Thus, if an author and reviewer are working on the review at the same time, the communication is synchronous and comment threads act similar to instant messaging. The comments are persisted so that if they work at different times, the communication

Figure 1. Screenshot of CodeFlow, the dominant code review tool used by developers at Microsoft.

becomes asynchronous. The bottom right pane **(5)** shows the summary of all the comments in the review.

CodeFlow centralizes and records all the information on code reviews on a central server. This provides an additional data source that we used to analyze real code review comments without incurring the Hawthorne effect [19].

C. Research Method

Our research method followed a mixed approach [20], depicted in Figure 2, collecting data from different sources for triangulation: **(1)** analysis of previous study, **(2)** observations and interviews with developers, **(3)** card sort on interview data, **(4)** card sort on code review comments, **(5)** the creation of an affinity diagram, and **(6)** survey to managers and programmers.

1. Analysis of previous study: Our research started with the analysis of a study commissioned by Microsoft, between April and May 2012 carried out by an external vendor. The study investigated how different product teams were using CodeFlow. It consisted of structured interviews (lasting 30-50 minutes) to 23 people with different roles.

Most of the interview questions revolved around topics that are very specific to tool usage, and were only tangentially related to this work. We found one relevant as a starting point for our study: "*What do you hope to accomplish when you submit a code review*?" We analyzed the transcript of this answer, for each interview, through the process of *coding* [21] (also used in *grounded theory* [22]): breaking up the answers into smaller coherent units (sentences or paragraphs) and adding *codes* to them. We organized codes into *concepts*, which in turn were grouped into more abstract *categories*.

From this analysis, four motivations emerged for code review: finding defects, maintaining team awareness, improving code quality, and assessing the high-level design. We used them to draw an initial guideline for our interviews.

2. Observations and interviews with developers: Next, we conducted a series of one-to-one meetings with developers who use CodeFlow, each taking 40-60 minutes.

We contacted 100 random candidates who signed-off between 50 and 250 code reviews since the CodeFlow release and sampled across different product teams to address our research questions from a *multi-point* perspective. We wrote developers who used CodeFlow in the past and asked them to contact us, giving us 30 minute notice when they received their next review task so that we could observe. The respondents that we interviewed comprised five developers, four senior developers, six testers, one senior tester, and one software architect. Their time in the company ranged from 18 months to almost 10 years, with a median of five years.

Each meeting was comprised of two parts: In the first part, we observed them performing the code review that they had been assigned. To minimize invasiveness we used only one observer and to encourage the participant to narrate their work, we asked the participants to think of us as a newcomer to the team. In this way, most developers thought aloud without need of prompting. With consent, we recorded the audio, assuring the participants of anonymity. Since we, as observers, have backgrounds in software development and practices at Microsoft, we were able to understand most of the work and where and how information was obtained without inquiry.

The second part of the meeting was a *semi-structured interview* [23]. Semi-structured interviews make use of an *interview guide* that contains general groupings of topics and questions rather than a pre-determined exact set and order of questions. They are often used in an exploratory context to "find out what is happening [and] to seek new insights" [24]. The guideline was iteratively refined after each interview, in particular when developers started providing answers very similar to the earlier ones, thus reaching a *saturation* effect.

Observations also reached a saturation point, thus providing insights very similar to the earlier ones. For this, after the first 5-6 observations, we adjusted the meetings to have shorter observations, which we used as a starting point for our meetings and as a "hook" to talk about topics in our guideline.

The audio of each interview was then transcribed and broken up into smaller coherent units for subsequent analysis.

3. Card sort (meetings): To group codes that emerged from interviews and observations into categories, we conducted a *card sort*. Card sorting is a sorting technique that is widely used in information architecture to create mental models and derive taxonomies from input data [25]. In our case it helped to organize the codes into hierarchies to deduce a higher level of abstraction and identify common themes. A card sort involves three phases: In the (1) *preparation phase*, participants of the card sort are selected and the cards are created; in the (2) *execution phase*, cards are sorted into meaningful groups with a descriptive title; and in the (3) *analysis phase*, abstract hierarchies are formed to deduce general categories.

We applied an *open card sort*: There were no predefined groups. Instead, the groups emerged and evolved during the sorting process. In contrast, a closed card sort has predefined groups and is typically applied when themes are known in advance, which was not the case for our study.

The first author of this paper created all of the cards, from the 1,047 coherent units in the interviews. Throughout our

Figure 2. The mixed approach research method applied.

further analysis other researchers (the second author and external people) were involved in developing categories and assigning cards to categories, so as to strengthen the validity of the result. The first author played a special role of ensuring that the context of each question was appropriately considered in the categorization, and creating the initial categories. To ensure the integrity of our categories, the cards were sorted by the first author several times to identify initial themes. Next, all researchers reviewed and agreed on the final set of categories.

4. Card sort (code review comments): The same method was applied to group code review comments into categories: We randomly sampled 200 threads with at least two comments (e.g., Point 4 of Figure 2), from the entire dataset of CodeFlow reviews, which embeds data from dozens of independent software products at Microsoft. We printed one card for each comment (along with the entire discussion thread to give the context), totaling 570 cards, and conducted a card sort, as performed for the interviews, to identify common themes.

5. Affinity Diagram: We used an *affinity diagram* to organize the categories that emerged from the card sort. This tool allows large numbers of ideas to be sorted into groups for review and analysis [26]. We used it to generate an overview of the topics that emerged from the card sort, in order to connect the related concepts and derive the main themes. For generating the affinity diagram, we followed the five canonical steps: we (1) recorded the categories on post-it-notes, (2) spread them onto a wall, (3) sorted the categories based on discussions, until all are sorted and all participants agreed, (4) named each group with a description, and (5) captured and discussed the themes.

6. Surveys: The final step of our study was aimed at validating the concepts that emerged from the previous phases. Towards this goal, we created two surveys to reach a significant number of participants and to challenge our conclusions (The full surveys are available as a technical report [27]). For the design of the surveys, we followed Kitchenham and Pfleeger's guidelines for personal opinion surveys [28]. Both surveys were anonymous to increase response rates [29].

We sent the first survey to a cross section of managers. We considered managers for which at least half of their team performed code reviews regularly (on average, one per week or more) and sampled along two dimensions. The first dimension was whether or not the manager had participated in a code review himself since the beginning of the year and the second dimension was whether the manager managed a single team or multiple teams (a manager of managers). Thus, we had one sample of first level managers who participated in review, another sample of second level managers who participated in reviews, *etc.* The first survey was a short survey comprising 6 questions (all optional), which we sent to 600 managers that had at least ten direct or indirect reporting developers who used CodeFlow in the past. The central focus was the open question asking to enumerate the main motivations for doing code reviews in their team. We received 165 answers (28% response rate), which we analyzed before devising the second survey

The second survey comprised 18 questions, mostly closed with multiple choice answers, and was sent to 2,000 randomly chosen developers who signed off on average at least one code review per week since the beginning of the year. We used the time frame of January to June of 2012 to minimize the amount of organizational churn during the time period and identify employees' activity in their current role and team. We received 873 answers (44% response rate). Both response rates were high, as other online surveys in software engineering have reported response rates ranging from 14% to 20% [30].

IV. Why Do Programmers Do Code Reviews?

Our first research question seeks to understand what motivations and expectations drive code reviews, and whether managers and developers share the same opinions.

Based on the responses that we coded from observations of developers performing code review as well as interviews, there are various motivations for code review. Overall, the interviews revealed that finding defects, even though prominent, is just one of the many motivations driving developers to perform code reviews. Especially when reinforced by a strong team culture around reviews, developers see code reviews as an activity that has multiple beneficial influences not only on the code, but also for the team and the entire development process. In this vein, one senior developer's comment summarized many of the responses: "*[code review] also has several beneficial influences: (1) makes people less protective about their code, (2) gives another person insight into the code, so*

there is (3) better sharing of information across the team, (4) helps support coding conventions on the team, and [...] (5) helps improving the overall process and quality of code."

Through the card sort on both meetings and code review comments, we found several references to motivations for code review and identified six main topics. To complete this list, in the survey for managers, we included an open question on why they perform code reviews in their team. We analyzed the responses to create a comprehensive list of high-level motivations. We included this list in the developers' survey and asked them to rank the top three main reasons that described why they do code reviews.

In the rest of this section, we discuss the motivations that emerged as the most prominent. We order them according to the importance they were given by the 873 developers and testers who responded to the final survey.

A. Finding Defects

One interviewed senior tester explains that he performs code reviews because they "*are a great source of bugs;*" he goes even further stating: "*sometimes code reviews are a cheaper form of bug finding than testing.*" Moreover, the tool seems not to have an impact on this main motivation: "*using CodeFlow or using any other tool makes a little difference to us; it's more about being able to identify flaws in the logic.*"

Almost all the managers included "finding defects" as one of the reasons for doing code reviews; for 44% of the managers, it is the top reason. Managers considered defects to be both low level issues (*e.g.*, "*correct logic is in place*") and high level concerns (*e.g.*, "*catch errors in design*"). Concerning surveyed developers/testers, "finding defects" is the first motivation for code review for 383 of the programmers (44%), second motivation for 204 (23%), and third for 96 (11%).

This is in-line with the reason why code inspections were devised in the first place: reducing software defects [1].

Nevertheless, even though "finding defects" emerged from our data as a strong motivation (the first for almost half of the programmers and managers), interviews and survey results indicate that this only tells part of the story of why practitioners do code reviews and the outcomes they expect.

B. Code Improvement

Code improvements are comments or changes about code in terms of readability, commenting, consistency, dead code removal, *etc.*, but do not involve correctness or defects.

Programmers ranked "code improvement" as an important motivation for code review, close to "finding defects:" This is the primary motivation for 337 programmers (39%), the second for 208 (24%), and the third for 135 (15%). Managers reported code improvements as their primary motivation in 51 cases (31%). One manager wrote how code review in her view is a "*discipline of explaining your code to your peers [that] drives a higher standard of coding. I think the process is even more important than the result.*"

Most interviewed programmers mentioned that at least one of the reviewers involved in each code review takes care of checking whether the code follows the team conventions, for example in terms of code formatting and in terms of function and variable naming. Some programmers use the "code improvement" check as a first step when doing code review: "*the first basic pass on the code is to check whether it is standard across the team.*"

The interviews also gave us a glimpse of the connection between the quality of code reviews and "code improvement" comments. Such comments seem easier to write and sometimes interviewees mentioned them as the way reviewers use to avoid spending time to conduct good code reviews. An observation by a senior developer, in the company for more than nine years, summarizes the opinions we received from many interviewees: "*I've seen quite a few code reviews where someone commented on formatting while missing the fact that there were security issues or data model issues.*"

C. Alternative Solutions

"Alternative solutions" regard changes and comments on improving the submitted code by adopting an idea that leads to a better implementation. This is one of the few motivations in which developers and managers do not agree. While 147 (17%) developers put this as the first motivation, 202 (23%) as the second, and 152 (17%) as the third, only 4 managers (2%) even mentioned it (*e.g.*, "*Generate better ideas, alternative approaches*" and "*Collective wisdom: Someone else on the project may have a better idea to solve a problem*"). The outcome of the interviews was similar to the position of managers: Interviewees vaguely mentioned this motivation, and mostly in terms of generic "*better ways to do things.*"

D. Knowledge Transfer

All the interviewees but one motivated their code reviews also from a learning, or "knowledge transfer," perspective. With the words of a senior developer: "*one of the things that should be happening with code reviews over time is a distribution of knowledge. If you do a code review and did not learn anything about the area and you still do not know anything about the area, then that was not as good code review as it could have been.*" Although we did not include questions related to "knowledge transfer" in our interview guideline, this topic kept emerging spontaneously from each meeting, thus underscoring its value for practitioners.

Sometimes programmers told us that they follow code reviews explicitly for learning purposes. For example, a tester explained: "[I read code reviews because] *from a code review you can learn about the different parts you have to touch to implement a certain feature.*"

According to interviewees, code review is a learning opportunity for both the author of the change and the reviewers: There is a bidirectional knowledge transfer about APIs usage, system design, best practices, team conventions, "*additional code tricks*," *etc.* Moreover code reviews are recognized for educating new developers about code writing.

Managers included "knowledge transfer" as one of the reasons for code review, although never as the top motivation. They mostly wrote about code review as an education means by mentioning among the motivations for code review: "*developer education*," "*education for junior developers who*

Figure 3. Developers' motivations for code review.

are learning the codebase," and "*learning tool to teach more junior team members*."

Programmers answering the survey declared "knowledge transfer" to be their first motivation for code review in 73 cases (8%), their second in 119 (14%), and their third in 141 (16%).

E. Team Awareness and Transparency

During one of our observations, one developer was preparing a code review submission as an author: He wanted other developers to "*double check*" his changes before committing them to the repository. After preparing the code, he specified the developers he wanted to review his code; he required not only two specific people, but he also put a generic email distribution group as an "optional" reviewer. When we inquired about this choice, he explained us: "*I am adding [this alias], so that everybody [in the team] is notified about the change I want to do before I check it in.*" In the subsequent interviews, this concept of using an email list as optional reviewer, or including specific optional reviewers exclusively for awareness emerged again frequently, *e.g.*, "*Code reviews are good FYIs [for your information]*."

Managers often mentioned the concept of team awareness as a motivation for code review, frequently justifying it with the notion of "transparency:" Not only must the team be kept aware of the directions taken by the code, but also nobody should be allowed to "secretly" make changes that might break the code or alter functionalities.

The 873 programmers answering the survey ranked "team awareness and transparency" very close to "knowledge transfer." In fact, the two concepts appeared logically related also in the interviews; for example one tester, while reviewing some code said: "*oh, this guy just implemented this feature, and now let me back and use it somewhere else.*" Showing that he both learned about the new feature and he was now aware of the possibility to use it in his own code. 75 (9%) developers considered team awareness their first motivation for code review, 108 (12%) their second, and 149 (17%) their third.

Although team awareness and transparency emerged from our data as clearly promoted by the code review process, academic research seems to have given little attention to it.

F. Share Code Ownership

The concept of "shared code ownership" is closely related to "team awareness and transparency," but it has a stronger connotation toward active collaboration and overlapping coding activities. Programmers and managers believe that code review is not only an occasion to notify other team members about incoming changes, but also a means to have more than one knowledgeable person about specific parts of the codebase. A manager put the following as her second motivation for code review: "*Broaden knowledge & understanding of how specific features/areas are designed and implemented (e.g., grooming "backup developers" for areas where knowledge is too concentrated on one or two expert developers.*"

Moreover, both developers and managers have the opinion that practicing code review also improves the personal *perception* of team members about shared code ownership. On this note, a senior developer, with more than 30 years in the software industry, explained: "*In the past people did not use to do code reviews and were very reluctant to put themselves in positions where they were having other people critiquing their code. The fact that code reviews are considered as a normal thing helps immensely with making people less protective about their code."* Similarly a manager wrote us explaining that she deems code reviews important because they "*Dilute any "rigid sense of ownership" that might develop over chunks of code.*"

In the programmers' survey, 51 respondents (6%) marked "share code ownership" as their first motivation, 100 (11%) as their second, and (10%) as their third.

G. Summary

In this section, we analyzed the motivations that developers and managers have for doing code review. We abstracted them into a list, which we finally included in the programmers' survey. Figure 3 reports the answers given to this question: The black bar is the number of developers that put that row as their top motivation, the gray bar is the number that put it as the second motivation, *etc.* We have ordered the factors by giving 3 points for a first motivation response, 2 points for a second motivation, *etc.* and then sorting by the sum.

We discussed the five most prominent motivations, which show that "finding defects" is the top motivation, although participants believe that code review brings other benefits. The first two motivations were already popular in research and their effectiveness have been evaluated in the context of code inspections; on the contrary, the other motivations are still unexplored, especially those regarding more "social" benefits on the team, such as shared code ownership.

Although motivations are well defined, we still have to verify whether they actually translate into real outcomes of a modern code review process.

V. The Outcomes of Code Reviews

Our second research question seeks to understand what the actual outcomes of code reviews are, and whether they match the motivations and expectations outlined in the previous section. To that end, we conducted *indirect field research* [31] by analyzing the content of 200 threads (corresponding to 570

Figure 4. Frequency of comments by card sort category.

comments) recorded by CodeFlow. Figure 4 shows the categories of comments found through the card sort.

Code Improvements: The most frequent category, with 165 (29%) comments, is "code improvements." In detail, among "code improvements" comments we find 58 on using better code practices, 55 on removing not necessary or unused code, and 52 on improving code readability.

Defect Finding: Although "defect finding" is the top motivation and expected outcome of code review for many practitioners, the category "defect" is the only the fourth most frequent, out of nine items, with 78 (14%) comments. Among "defect" comments, 65 are on logical issues (*e.g.*, a wrong expression in an *if* clause), 6 on high-level issues, 5 on security, and 3 on wrong exception handling.

Knowledge Transfer: Concerning the other expected outcomes of code reviews, we did not expect to find evidence about them, because of their more "social"—thus harder to quantify—nature. Nevertheless, we found some (12) comments specifically about "knowledge transfer," where the reviewers were directing the code change author to external resources (*e.g.*, internal documentation or websites) for learning how to tackle some issues. This provides additional evidence on the importance of this aspect of reviews.

A. Finding defects: When expectations do not meet reality

Why do we see this significant gap in frequency between "code improvements" and "defects" comments? Possible reasons may be that our sample of 570 comments is too small to represent the population, that the submitted changes might require less need fixing of "real" defects than of small code improvements, or that programmers could consider "code improvements" as actual defects. However, by triangulating these numbers with the interview discussions, the survey answers, and the other categories of comments, another reason seems to justify this situation. First, we start by noting that most of the comments on "defects" regard *uncomplicated* logical errors, *e.g.*, corner cases, common configuration values, or operator precedence. Then, from interview data, we see that: (1) most interviewees explained how, with tool-based code reviews, most of the found defects regard "*logic issues—where the author might not have considered a particular or corner case*"; (2) some interviewees complained that the quality of code reviews is low, because reviewers only look for easy errors: "*[Some reviewers] focus on formatting mistakes because they are easy [...], but it doesn't really help. […] In some ways it's kind of embarrassing if someone asks you to do a code review and all you can find are formatting mistakes when there are real mistakes to be found*"; and (3) other interviewees admitted that if the code is not among their codebase, they look at "*obvious bugs (such as, exception handling).*" Finally, managers mentioned "*catching early obvious bugs*" or "*finding obvious inefficiencies or errors*" as reasons for doing code review.

These points illustrate that the reason for the gap between the number of comments on "code improvements" and on "defects" is not to be found in problems in the sample or in classification misconceptions, but it is rather just additional corroborating evidence that the outcome of code review does not match the main expectation of both programmers and managers—*finding defects*. Review comments about defects are few, comprising one-eighth of the total in our sample, and mostly address "micro" level and superficial concerns; while programmers and managers would expect more insightful remarks on conceptual and design level issues. Why does this happen? The high frequency of understanding comments hints at the answer to our question, addressed in the next section.

VI. What are The Challenges of Code Review?

Our third research question seeks to understand the main challenges faced by the reviewers when performing modern code reviews, also with respect to the expected outcomes. We also seek to uncover the reason behind the mismatch between expectations and actual outcomes on finding defects in code reviews.

A. Code Review is Understanding

Even though we did not ask any specific question concerning understanding, the theme emerged clearly from our interviews. Many interviewees eventually acknowledged that understanding is their main challenge when doing code reviews. For example, a senior developer autonomously explained to us: "*the most difficult thing when doing a code review is understanding the reason of the change*;" a tester, in the same vein: "*the biggest information need in code review: what instigated the change;"* and another senior developer: "*in a successful code review submission the author is sure that his peers understand and approve the change.*" Although the textual description should help reviewers understanding, some developers do not find it useful: "*people can say they are doing one thing, while they are doing many more of them,*" or "*the description is not enough;*" in general, developers seem to confirm that "*not knowing files (or [dealing with] new ones) is a major reason for not understanding a change.*"

From interviews, no other code review challenge emerged as clearly as understanding the submitted change. Even though scheduling and time issues also appeared challenging, we could always trace them back to the first challenge—through the words of a tester: "*understanding the code takes most of the*

Figure 5. Developers' responses in surveys of the amount of code understanding for code review outcomes.

reviewing time." On the same note, in the code review comments we analyzed, the second most frequent category concerns *understanding*. This category includes clarification questions and doubts raised by the reviewers who want to grasp the rationale of the changes done on the code, and the corresponding clarification answers. This is also in line with the evidence delivered by Sutherland & Venolia on the relevance of rationale articulation in reviews [17].

Do *understanding needs* change with the expected outcome of code review? We included a question in the programmers' survey to know how much understanding they needed to achieve each of the motivations listed in Figure 3. The outcome of the question is summarized in Figure 5. The respondents could answer with a four values Likert's scale, by selecting the understanding of the change they felt was required to achieve the specific outcome. The most difficult task from the understanding perspective is "finding defects," immediately followed by "alternative solutions." Both clearly stand out from the other items. The gap in understanding needs between "finding defects" and "code improvement" seems to corroborate our hypothesis that the difference in the number of comments about these two items in review comments is mostly due to understanding issues. Thus, if managers and developers want code review to match their need for "finding defects," context and change understanding must be improved.

B. A Priori Understanding

By observing developers performing code reviews, we noticed that somea started code reviews by thoroughly reading the accompanying textual description, while others went directly to a specific changed file. In the first group, the time required for putting the first review comments and understanding the change rationale was noticeably longer, and some of the comments were asking to clarify the reasons for a change. To better comprehend this situation, we included in our interview guideline a questionn about how the interviewees start code reviews. Participants explained that when they own or are very familiar with the files being changed, they have a better context and it is easier for them to understand the change submitted: "*when doing code review I start with things I am familiar with, so it is easier to see what is going on.*" When they are file owners, they often do not even need to read the description, but they "*go directly to the files they own.*" On the contrary, when they do not own files, or have to review new files, they need more information and try to get it from the description, which is deemed good when it states "*what was changed and why.*"

To better understand this aspect we included two questions in the programmers' survey to know (1) whether it takes longer to review files they are not familiar with, and why; and (2) whether reviewers familiar with the changed files give different feedback, and how.

Most of the respondents (798, 91%) answered positively to the first question, motivating it with the fact that it takes time to familiarize with the code and "*learn enough about the files being modified to understand their purpose, invariants, APIs, etc.,*" because "*big-picture impact analysis requires contextual understanding. When reviewing a small, unfamiliar change, it is often necessary to read through much more code than that being reviewed.*" The comment of a developer anticipates the answer to the second question: "*It takes a lot longer to understand unknown code, but even then understanding isn't very deep. With code I am familiar with I have more to say. I know what to say faster. What I have to say is deeper. And I can be more insistent on it.*" In fact, the answer to the second question is positive in 716 (82%) cases. The main difference with file owner comments is that they are substantially deeper, more detailed and insightful. A respondent explained: "*Comments reflect their deeper understanding -- more likely to find subtle defects, feedback is more conceptual (better ideas, approaches) instead of superficial (naming, mechanical style, etc.)*" another tried to boldly summarize the concept: "*Difference between algorithmic analysis and comments on coding style. The difference is big.*"

In fact, when the context is clear and understanding is very high, as in the case when the reviewer is the owner of changed files, code review authors receive comments that explore "*deeper details*," are "*more directed*" and "*more actionable and pertinent*," and find "*more subtle issues.*"

C. Dealing with Understanding Needs

From the interviews, we found that, in the current situation, reviewers try different paths to understand the context and the changes: They read the change description, try to run the changed code, send emails for understanding high level details about the review, and often (from 20% to 40% of the times) even go to talk in person to have a "*higher communication bandwidth*" for asking clarifications to the author. All code review tools that we see in practice today deliver only basic support for the understanding needs of reviewers – providing features such as diffing capabilities, inline commenting, or syntax highlighting, which are limited when dealing with complex code understanding.

VII. Recommendations and Implications

A. Recommendations for Practitioners

From our work we derive recommendations to developers:

Quality Assurance: There is a mismatch between the expectations and the actual outcomes of code reviews. From our study, review does not result in identifying defects as often as project members would like and even more rarely detects deep, subtle, or "macro" level issues. Relying on code review in this way for quality assurance may be fraught.

Understanding: When reviewers have *a priori* knowledge of the context and the code, they complete reviews more quickly and provide more valuable feedback to the author. Teams should aim to increase the breadth of understanding of developers (if the author of a change is the only expert, she has no potential reviewers) and change authors should include code owners and others with understanding as much as possible when using review to identify defects. Developers indicated that when the author provided context and direction to them in a review, they could respond better and faster.

Beyond Defects: Modern code reviews provide benefits beyond finding defects. Code review can be used to improve code style, find alternative solutions, increase learning, share code ownership, *etc*. This should guide code review policies.

Communication: Despite the growth of tools for supporting code reviews, developers still have need of richer communication than comments annotating the changed code when reviewing. Teams should provide mechanisms for in-person or, at least, synchronous communication.

B. Implications for Researchers

Our work uncovered aspects of code review—beyond our research questions—that deserve further study:

Automate Code Review Tasks: We observed that many code review comments were related to "code improvement" concerns and low-level "micro" defects. Identifying both of these are problems that research has begun to solve. Tools for enforcing team code conventions, checking for typos, and identifying dead code already exist. Even more advanced tasks such as checking boundary conditions or catching common mistakes have been shown to work in practice on real code. For example Google experimented with adding FindBugs to their review process, though little is reported about the results [32]. Automating these tasks frees reviewers to look for deeper, more subtle defects. Code review is fertile ground to have an impact with code analysis tools.

Program Comprehension in Practice: We identified context and change understanding as challenges that developers face when reviewing, with a direct relationship to the quality of review comments. Interestingly, modern IDEs ship with many tools to aid context and understanding, and there is an entire conference (ICPC) devoted to code comprehension, yet all current code review tools we know of show a highlighted diff of the changed files to a reviewer with no additional tool support. The most common motivation that we have seen for code comprehension research is a developer that is working on new code, but we argue that reviewers reviewing code they have not seen before may be more common than a developer working on new code. This is a ripe opportunity for code understanding researchers to have impact on real world scenarios.

Socio-technical Effects: Awareness and learning were cited as motivations for code review, but these outcomes are difficult to observe from traces in reviews. We did not investigate these further, but studies can be designed and carried out to determine if and how awareness and learning increase as a result of being involved in code review.

VIII. Limitations

As a qualitative study, gauging the validity of our findings is a difficult undertaking [33]. While we have endeavored to uncover and report the expectations, outcomes, and challenges of code review, limitations may exist. We describe them with the steps that we took to increase confidence and validity.

To achieve a comprehensive view of code review, we triangulated by collecting and comparing results from multiple sources. For example, we found strong agreement among the results of expectations collected from interviews, surveys of manager, and surveys of developers. By starting with exploratory interviews of a smaller set of subjects (17) followed by open coding to extract themes, we identified core questions that we addressed to a larger audience via survey.

One common notion is that empirical research within one company or one project provides little value for the academic community, and does not contribute to scientific development. Historical evidence shows otherwise. Flyvbjerg provides several examples of individual cases that contributed to discovery in physics, economics, and social science [34]. Beveridge observed for social sciences: *"More discoveries have arisen from intense observation than from statistics applied to large groups"* (as quoted in Kuper and Kuper [35], page 95). This should not be interpreted as a criticism of research that focuses on large samples. For the development of an empirical body of knowledge as championed by Basili [36], both types of research are essential. To understand code review across many contexts, we observed, interviewed, surveyed, and examined code reviews from developers across a diverse group of software teams that work with codebases in various domains, of varying sizes, and with varying processes.

Concerning the representativeness of our results in other contexts, other companies and OSS use tools similar to CodeFlow [8] [7] [6]. However, team dynamics may differ. The need for code understanding may already be met in contexts where projects are smaller or there is shared code ownership and a broad system understanding across the team. We found that higher levels of understanding lead to more informative comments, which identify defects or aid the author in other ways so review in these contexts may uncover more defects. In OSS contexts, project-specific expertise often must be demonstrated prior to being accepted as a "core committer" [37], so learning may not be as important or frequent an outcome for review.

In this work, we have used discussions within CodeFlow to identify and quantify outcomes of code review. However, some motivations that managers and developers described are

not easily observable because they leave little trace. For example, determining how often code review improves team awareness or transfers knowledge is difficult to assess from the discussions in reviews. For these outcomes, we have responses indicating that they occur, but not "hard evidence."

Based on review comments, survey responses, and interviews, we know that in-person discussions occurred frequently. While we are unable to compare frequency of these events to other outcomes as we can with events that are recorded in CodeFlow, we know that they most often occurred to address understanding needs.

IX. Conclusion

In this work, we have investigated modern, tool-based code review, uncovered both a wide range of motivations for review, and determined that the outcomes do not always match those motivations. We identified understanding as a key component and provided recommendations to both practitioners and researchers. It is our hope that the insights we have discovered lead to more effective review in practice and improved tools, based on research, to aid developers perform code reviews.

References

[1] A. Frank Ackerman, Priscilla J. Fowler, and Robert G. Ebenau, "Software inspections and the industrial production of software," in *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, 1984, pp. 13--40.

[2] A.F. Ackerman, L.S. Buchwald, and F.H. Lewski, "Software inspections: an Effective Verification Process," *IEEE Software*, 1989.

[3] M.E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, 1976.

[4] Shull, "Inspecting the History of Inspections," 2008.

[5] P. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. German, "Open Source Peer Review – Lessons and Recommendations for," *IEEE Software*, 2012.

[6] Niall Kennedy. (2006, Dec) How Google does web-based code reviews with Mondrian. [Online]. http://www.niallkennedy.com/blog/2006/11/google-mondrian.html

[7] Alexia Tsotsis. (2011, August) Meet Phabricator, The Witty Code Review Tool Built Inside Facebook. [Online]. http://techcrunch.com/2011/08/07/oh-what-noble-scribe-hath-penned-these-words/

[8] (2012, June) Gerrit (software). Wikipedia Article. [Online]. http://en.wikipedia.org/wiki/Gerrit_(software)

[9] O. Laitenberger, "A Survey of Software Inspection Technologies," in *Handbook on Software Engineering and Knowledge Engineering*., 2002, pp. 517-555.

[10] J.P. Johnson, "Collaboration, Peer Review, and Open Source Software," *Information Economics and Policy*, vol. 18, pp. 477-497, 2006.

[11] A. Porter, H. Siy, and L. Votta, "A review of software inspections," *Advances in Computers*, vol. 42, pp. 39--76, 1996.

[12] L.G. Votta, "Does every inspection need a meeting?," *ACM SIGSOFT Software Engineering Notes*, vol. 18, pp. 107--114, 1993.

[13] L. Brothers, V. Sembugamoorthy, and M. Muller, "ICICLE: groupware for code inspection," in *Conference on Computer-Supported Cooperative Work*, 1990, pp. 169--181.

[14] Vahid Mashayekhi, Chris Feulner, and John Riedl, "CAIS: collaborative asynchronous inspection of software," *SIGSOFT Softw. Eng. Notes*, vol. 19, pp. 21--34, dec 1994.

[15] John Gintell et al., "Scrutiny: A Collaborative Inspection and Review System," in *Proceedings of the 4th European Software Engineering Conference*, 1993, pp. 344--360.

[16] P. Rigby, D. German, and M. Storey, "Open source software peer review practices: a case study of the apache server," in *Proceedings of ICSE*, 2008.

[17] A. Sutherland and G. Venolia, "Can peer code reviews be exploited for later information needs?," in *Proceedings of ICSE*, may 2009.

[18] T. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of ICSE*, 2006.

[19] John G. Adair, "The Hawthorne effect: A reconsideration of the methodological artifact," *Journal of Applied Psychology*, vol. 69, no. 2, pp. 334-345, 1984.

[20] J. Creswell, *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*, 3rd ed.: Sage Publications, Inc., 2009.

[21] B. L. Berg, *Qualitative Research Methods for Social Sciences*. Boston: Pearson, 2004.

[22] S., W. Hall, and P. Kruchten Adolph, "Using Grounded Theory to Study the Experience of Software Development," *Empirical Software Engineering*, vol. 16, no. 4, pp. 487-513, 2011.

[23] T. Lindlof and B. Taylor, *Qualitative Communication Research Methods*.: Sage, 2002.

[24] R. S. Weiss, *Learning From Strangers: The Art and Method of Qualitative Interview Studies*.: The Free Press, 1995.

[25] I. Barker. (2005, May) What is information architecture? KM Column. [Online]. http://www.steptwo.com.au/

[26] Janice E. Shade Stuart J. Janis, *Improving Performance Through Statistical Thinking*.: Mcgraw-Hill, 2000.

[27] A. Bacchelli and C. Bird, "Appendix to Expectations, Outcomes, and Challenges of Modern Code Review," Microsoft Research, Technical Report MSR-TR-2012-83 2012. [Online]. http://research.microsoft.com/apps/pubs/?id=171426

[28] B. A. Kitchenham and S. L. Pfleeger, "Personal Opinion Surveys," in *Guide to Advanced Empirical Software Engineering*, Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, Eds.: Springer, 2007.

[29] P. K. Tyagi, "The Effects of Appeals, Anonymity, and Feedback on Mail Survey Response Patterns from Salespeople," *Journal of The Academy of Marketing Science*, 1989.

[30] T. Punter, M. Ciolkowski, B. G. Freimut, and I. John, "Conducting on-line surveys in software engineering," in *Proc. of International Symposium on Empirical Software Engineering*, 2003.

[31] T. Lethbridge, S. Sim, and J. Singer, "Studying software engineers: Data collection techniques for software field studies," *Empirical Software Engineering*, vol. 10, pp. 311--341, 2005.

[32] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Using FindBugs on production software," in *OOPSLA '07*, 2007.

[33] N. Golafshani, "Understanding reliability and validity in qualitative research," *The qualitative report*, vol. 8, pp. 597--607, 2003.

[34] B. Flyvbjerg, "Five misunderstandings about case-study research," *Qualitative inquiry*, vol. 12, no. 2, pp. 219-245, 2006.

[35] A. Kuper and J. Kuper, *The Social Science Encyclopedia*.: Routledge, 1985.

[36] V. Basili, F. Shull, and F. Lanubile, "Building knowledge through families of experiments.," *IEEE Trans. Software Eng.*, vol. 25, no. 4, pp. 456-173, 1999.

[37] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu, "Open Borders? Immigration in Open Source Projects," in *International Workshop on Mining Software Repositories*, 2007.

4

Requirements engineering

Objectives

The objective of this chapter is to introduce software requirements and to explain the processes involved in discovering and documenting these requirements. When you have read the chapter, you will:

- understand the concepts of user and system requirements and why these requirements should be written in different ways;
- understand the differences between functional and non-functional software requirements;
- understand the main requirements engineering activities of elicitation, analysis, and validation, and the relationships between these activities;
- understand why requirements management is necessary and how it supports other requirements engineering activities.

Contents

4.1 Functional and non-functional requirements
4.2 Requirements engineering processes
4.3 Requirements elicitation
4.4 Requirements specification
4.5 Requirements validation
4.6 Requirements change

The requirements for a system are the descriptions of the services that a system should provide and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analyzing, documenting and checking these services and constraints is called requirements engineering (RE).

The term *requirement* is not used consistently in the software industry. In some cases, a requirement is simply a high-level, abstract statement of a service that a system should provide or a constraint on a system. At the other extreme, it is a detailed, formal definition of a system function. Davis (Davis 1993) explains why these differences exist:

> *If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not predefined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization's needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system*[†].

Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between these different levels of description. I distinguish between them by using the term *user requirements* to mean the high-level abstract requirements and *system requirements* to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

1. User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate. The user requirements may vary from broad statements of the system features required to detailed, precise descriptions of the system functionality.
2. System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

Different kinds of requirement are needed to communicate information about a system to different types of reader. Figure 4.1 illustrates the distinction between user and system requirements. This example from the mental health care patient information system (Mentcare) shows how a user requirement may be expanded into several system requirements. You can see from Figure 4.1 that the user requirement is quite

[†]Davis, A. M. 1993. Software Requirements: Objects, Functions and States. Englewood Cliffs, NJ: Prentice-Hall.

User requirements definition

1. The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g. 10mg, 20mg, etc.) separate reports shall be created for each dose unit.
1.5 Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

Figure 4.1 User and system requirements

general. The system requirements provide more specific information about the services and functions of the system that is to be implemented.

You need to write requirements at different levels of detail because different types of readers use them in different ways. Figure 4.2 shows the types of readers of the user and system requirements. The readers of the user requirements are not usually concerned with how the system will be implemented and may be managers who are not interested in the detailed facilities of the system. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation.

The different types of document readers shown in Figure 4.2 are examples of system stakeholders. As well as users, many other people have some kind of interest in the system. System stakeholders include anyone who is affected by the system in some way and so anyone who has a legitimate interest in it. Stakeholders range from end-users of a system through managers to external stakeholders such as regulators,

Figure 4.2 Readers of different types of requirements specification

Feasibility studies

A feasibility study is a short, focused study that should take place early in the RE process. It should answer three key questions: (1) Does the system contribute to the overall objectives of the organization? (2) Can the system be implemented within schedule and budget using current technology? and (3) Can the system be integrated with other systems that are used?

If the answer to any of these questions is no, you should probably not go ahead with the project.

http://software-engineering-book.com/web/feasibility-study/

who certify the acceptability of the system. For example, system stakeholders for the Mentcare system include:

1. Patients whose information is recorded in the system and relatives of these patients.
2. Doctors who are responsible for assessing and treating patients.
3. Nurses who coordinate the consultations with doctors and administer some treatments.
4. Medical receptionists who manage patients' appointments.
5. IT staff who are responsible for installing and maintaining the system.
6. A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
7. Health care managers who obtain management information from the system.
8. Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

Requirements engineering is usually presented as the first stage of the software engineering process. However, some understanding of the system requirements may have to be developed before a decision is made to go ahead with the procurement or development of a system. This early-stage RE establishes a high-level view of what the system might do and the benefits that it might provide. These may then be considered in a feasibility study, which tries to assess whether or not the system is technically and financially feasible. The results of that study help management decide whether or not to go ahead with the procurement or development of the system.

In this chapter, I present a "traditional" view of requirements rather than requirements in agile processes, which I discussed in Chapter 3. For the majority of large systems, it is still the case that there is a clearly identifiable requirements engineering phase before implementation of the system begins. The outcome is a requirements document, which may be part of the system development contract. Of course, subsequent changes are made to the requirements, and user requirements may be expanded into

more detailed system requirements. Sometimes an agile approach of concurrently eliciting the requirements as the system is developed may be used to add detail and to refine the user requirements.

4.1 Functional and non-functional requirements

Software system requirements are often classified as functional or non-functional requirements:

1. *Functional requirements* These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.
2. *Non-functional requirements* These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole rather than individual system features or services.

In reality, the distinction between different types of requirements is not as clear-cut as these simple definitions suggest. A user requirement concerned with security, such as a statement limiting access to authorized users, may appear to be a non-functional requirement. However, when developed in more detail, this requirement may generate other requirements that are clearly functional, such as the need to include user authentication facilities in the system.

This shows that requirements are not independent and that one requirement often generates or constrains other requirements. The system requirements therefore do not just specify the services or the features of the system that are required; they also specify the necessary functionality to ensure that these services/features are delivered effectively.

4.1.1 Functional requirements

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user requirements, functional requirements should be written in natural language so that system users and managers can understand them. Functional system requirements expand the user requirements and are written for system developers. They should describe the system functions, their inputs and outputs, and exceptions in detail.

Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems. For example, here are examples of functional

Domain requirements

Domain requirements are derived from the application domain of the system rather than from the specific needs of system users. They may be new functional requirements in their own right, constrain existing functional requirements, or set out how particular computations must be carried out.

The problem with domain requirements is that software engineers may not understand the characteristics of the domain in which the system operates. This means that these engineers may not know whether or not a domain requirement has been missed out or conflicts with other requirements.

http://software-engineering-book.com/web/domain-requirements/

requirements for the Mentcare system, used to maintain information about patients receiving treatment for mental health problems:

1. A user shall be able to search the appointments lists for all clinics.
2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

These user requirements define specific functionality that should be included in the system. The requirements show that functional requirements may be written at different levels of detail (contrast requirements 1 and 3).

Functional requirements, as the name suggests, have traditionally focused on what the system should do. However, if an organization decides that an existing off-the-shelf system software product can meet its needs, then there is very little point in developing a detailed functional specification. In such cases, the focus should be on the development of information requirements that specify the information needed for people to do their work. Information requirements specify the information needed and how it is to be delivered and organized. Therefore, an information requirement for the Mentcare system might specify what information is to be included in the list of patients expected for appointments that day.

Imprecision in the requirements specification can lead to disputes between customers and software developers. It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs.

For example, the first Mentcare system requirement in the above list states that a user shall be able to search the appointments lists for all clinics. The rationale for this requirement is that patients with mental health problems are sometimes confused. They may have an appointment at one clinic but actually go to a different clinic. If they have an appointment, they will be recorded as having attended, regardless of the clinic.

A medical staff member specifying a search requirement may expect "search" to mean that, given a patient name, the system looks for that name in all appointments at all clinics. However, this is not explicit in the requirement. System developers may interpret the requirement so that it is easier to implement. Their search function may require the user to choose a clinic and then carry out the search of the patients who attended that clinic. This involves more user input and so takes longer to complete the search.

Ideally, the functional requirements specification of a system should be both complete and consistent. Completeness means that all services and information required by the user should be defined. Consistency means that requirements should not be contradictory.

In practice, it is only possible to achieve requirements consistency and completeness for very small software systems. One reason is that it is easy to make mistakes and omissions when writing specifications for large, complex systems. Another reason is that large systems have many stakeholders, with different backgrounds and expectations. Stakeholders are likely to have different—and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are originally specified, and the inconsistent requirements may only be discovered after deeper analysis or during system development.

4.1.2 Non-functional requirements

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users. These non-functional requirements usually specify or constrain characteristics of the system as a whole. They may relate to emergent system properties such as reliability, response time, and memory use. Alternatively, they may define constraints on the system implementation, such as the capabilities of I/O devices or the data representations used in interfaces with other systems.

Non-functional requirements are often more critical than individual functional requirements. System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly.

While it is often possible to identify which system components implement specific functional requirements (e.g., there may be formatting components that implement reporting requirements), this is often more difficult with non-functional requirements. The implementation of these requirements may be spread throughout the system, for two reasons:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance requirements are met in an embedded system, you may have to organize the system to minimize communications between components.

Figure 4.3 Types of non-functional requirements

2. An individual non-functional requirement, such as a security requirement, may generate several, related functional requirements that define new system services that are required if the non-functional requirement is to be implemented. In addition, it may also generate requirements that constrain existing requirements; for example, it may limit access to information in the system.

Nonfunctional requirements arise through user needs because of budget constraints, organizational policies, the need for interoperability with other software or hardware systems, or external factors such as safety regulations or privacy legislation. Figure 4.3 is a classification of non-functional requirements. You can see from this diagram that the non-functional requirements may come from required characteristics of the software (product requirements), the organization developing the software (organizational requirements), or external sources:

1. *Product requirements* These requirements specify or constrain the runtime behavior of the software. Examples include performance requirements for how fast the system must execute and how much memory it requires; reliability requirements that set out the acceptable failure rate; security requirements; and usability requirements.

2. *Organizational requirements* These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organizations. Examples include operational process requirements that define how the system will be used; development process requirements that specify the

PRODUCT REQUIREMENT
The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 08:30–17:30). Downtime within normal working hours shall not exceed 5 seconds in any one day.

ORGANIZATIONAL REQUIREMENT
Users of the Mentcare system shall identify themselves using their health authority identity card.

EXTERNAL REQUIREMENT
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Figure 4.4 Examples of possible non-functional requirements for the Mentcare system

programming language; the development environment or process standards to be used; and environmental requirements that specify the operating environment of the system.

3. *External requirements* This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a nuclear safety authority; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will be acceptable to its users and the general public.

Figure 4.4 shows examples of product, organizational, and external requirements that could be included in the Mentcare system specification. The product requirement is an availability requirement that defines when the system has to be available and the allowed downtime each day. It says nothing about the functionality of the Mentcare system and clearly identifies a constraint that has to be considered by the system designers.

The organizational requirement specifies how users authenticate themselves to the system. The health authority that operates the system is moving to a standard authentication procedure for all software where, instead of users having a login name, they swipe their identity card through a reader to identify themselves. The external requirement is derived from the need for the system to conform to privacy legislation. Privacy is obviously a very important issue in health care systems, and the requirement specifies that the system should be developed in accordance with a national privacy standard.

A common problem with non-functional requirements is that stakeholders propose requirements as general goals, such as ease of use, the ability of the system to recover from failure, or rapid user response. Goals set out good intentions but cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered. For example, the following system goal is typical of how a manager might express usability requirements:

The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized.

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Megabytes/Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Figure 4.5 Metrics for specifying non-functional requirements

I have rewritten this to show how the goal could be expressed as a "testable" non-functional requirement. It is impossible to objectively verify the system goal, but in the following description you can at least include software instrumentation to count the errors made by users when they are testing the system.

> *Medical staff shall be able to use all the system functions after two hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.*

Whenever possible, you should write non-functional requirements quantitatively so that they can be objectively tested. Figure 4.5 shows metrics that you can use to specify non-functional system properties. You can measure these characteristics when the system is being tested to check whether or not the system has met its non-functional requirements.

In practice, customers for a system often find it difficult to translate their goals into measurable requirements. For some goals, such as maintainability, there are no simple metrics that can be used. In other cases, even when quantitative specification is possible, customers may not be able to relate their needs to these specifications. They don't understand what some number defining the reliability (for example) means in terms of their everyday experience with computer systems. Furthermore, the cost of objectively verifying measurable, non-functional requirements can be very high, and the customers paying for the system may not think these costs are justified.

Non-functional requirements often conflict and interact with other functional or non-functional requirements. For example, the identification requirement in Figure 4.4 requires a card reader to be installed with each computer that connects to the system. However, there may be another requirement that requests mobile access to the system from doctors' or nurses' tablets or smartphones. These are not normally

equipped with card readers so, in these circumstances, some alternative identification method may have to be supported.

It is difficult to separate functional and non-functional requirements in the requirements document. If the non-functional requirements are stated separately from the functional requirements, the relationships between them may be hard to understand. However, you should, ideally, highlight requirements that are clearly related to emergent system properties, such as performance or reliability. You can do this by putting them in a separate section of the requirements document or by distinguishing them, in some way, from other system requirements.

Non-functional requirements such as reliability, safety, and confidentiality requirements are particularly important for critical systems. I cover these dependability requirements in Part 2, which describes ways of specifying reliability, safety, and security requirements.

4.2 Requirements engineering processes

As I discussed in Chapter 2, requirements engineering involves three key activities. These are discovering requirements by interacting with stakeholders (elicitation and analysis); converting these requirements into a standard form (specification); and checking that the requirements actually define the system that the customer wants (validation). I have shown these as sequential processes in Figure 2.4. However, in practice, requirements engineering is an iterative process in which the activities are interleaved.

Figure 4.6 shows this interleaving. The activities are organized as an iterative process around a spiral. The output of the RE process is a system requirements document. The amount of time and effort devoted to each activity in an iteration depends on the stage of the overall process, the type of system being developed, and the budget that is available.

Early in the process, most effort will be spent on understanding high-level business and non-functional requirements, and the user requirements for the system. Later in the process, in the outer rings of the spiral, more effort will be devoted to eliciting and understanding the non-functional requirements and more detailed system requirements.

This spiral model accommodates approaches to development where the requirements are developed to different levels of detail. The number of iterations around the spiral can vary so that the spiral can be exited after some or all of the user requirements have been elicited. Agile development can be used instead of prototyping so that the requirements and the system implementation are developed together.

In virtually all systems, requirements change. The people involved develop a better understanding of what they want the software to do; the organization buying the system changes; and modifications are made to the system's hardware, software, and organizational environment. Changes have to be managed to understand the impact on other requirements and the cost and system implications of making the change. I discuss this process of requirements management in Section 4.6.

Figure 4.6 A spiral view of the requirements engineering process

4.3 Requirements elicitation

The aims of the requirements elicitation process are to understand the work that stakeholders do and how they might use a new system to help support that work. During requirements elicitation, software engineers work with stakeholders to find out about the application domain, work activities, the services and system features that stakeholders want, the required performance of the system, hardware constraints, and so on.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

1. Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.

Figure 4.7 The requirements elicitation and analysis process

2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.

3. Different stakeholders, with diverse requirements, may express their requirements in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.

4. Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.

5. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

A process model of the elicitation and analysis process is shown in Figure 4.7. Each organization will have its own version or instantiation of this general model, depending on local factors such as the expertise of the staff, the type of system being developed, and the standards used.

The process activities are:

1. *Requirements discovery and understanding* This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.

2. *Requirements classification and organization* This activity takes the unstructured collection of requirements, groups related requirements and organizes them into coherent clusters.

3. *Requirements prioritization and negotiation* Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts

Viewpoints

A viewpoint is a way of collecting and organizing a set of requirements from a group of stakeholders who have something in common. Each viewpoint therefore includes a set of system requirements. Viewpoints might come from end-users, managers, or others. They help identify the people who can provide information about their requirements and structure the requirements for analysis.

http://www.software-engineering-book.com/web/viewpoints/

through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.

4. *Requirements documentation* The requirements are documented and input into the next round of the spiral. An early draft of the software requirements documents may be produced at this stage, or the requirements may simply be maintained informally on whiteboards, wikis, or other shared spaces.

Figure 4.7 shows that requirements elicitation and analysis is an iterative process with continual feedback from each activity to other activities. The process cycle starts with requirements discovery and ends with the requirements documentation. The analyst's understanding of the requirements improves with each round of the cycle. The cycle ends when the requirements document has been produced.

To simplify the analysis of requirements, it is helpful to organize and group the stakeholder information. One way of doing so is to consider each stakeholder group to be a viewpoint and to collect all requirements from that group into the viewpoint. You may also include viewpoints to represent domain requirements and constraints from other systems. Alternatively, you can use a model of the system architecture to identify subsystems and to associate requirements with each subsystem.

Inevitably, different stakeholders have different views on the importance and priority of requirements, and sometimes these views are conflicting. If some stakeholders feel that their views have not been properly considered, then they may deliberately attempt to undermine the RE process. Therefore, it is important that you organize regular stakeholder meetings. Stakeholders should have the opportunity to express their concerns and agree on requirements compromises.

At the requirements documentation stage, it is important that you use simple language and diagrams to describe the requirements. This makes it possible for stakeholders to understand and comment on these requirements. To make information sharing easier, it is best to use a shared document (e.g., on Google Docs or Office 365) or a wiki that is accessible to all interested stakeholders.

4.3.1 Requirements elicitation techniques

Requirements elicitation involves meeting with stakeholders of different kinds to discover information about the proposed system. You may supplement this information

with knowledge of existing systems and their usage and information from documents of various kinds. You need to spend time understanding how people work, what they produce, how they use other systems, and how they may need to change to accommodate a new system.

There are two fundamental approaches to requirements elicitation:

1. Interviewing, where you talk to people about what they do.
2. Observation or ethnography, where you watch people doing their job to see what artifacts they use, how they use them, and so on.

You should use a mix of interviewing and observation to collect information and, from that, you derive the requirements, which are then the basis for further discussions.

4.3.1.1 Interviewing

Formal or informal interviews with system stakeholders are part of most requirements engineering processes. In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. Requirements are derived from the answers to these questions. Interviews may be of two types:

1. Closed interviews, where the stakeholder answers a predefined set of questions.
2. Open interviews, in which there is no predefined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develops a better understanding of their needs.

In practice, interviews with stakeholders are normally a mixture of both of these. You may have to obtain the answer to certain questions, but these usually lead to other issues that are discussed in a less structured way. Completely open-ended discussions rarely work well. You usually have to ask some questions to get started and to keep the interview focused on the system to be developed.

Interviews are good for getting an overall understanding of what stakeholders do, how they might interact with the new system, and the difficulties that they face with current systems. People like talking about their work, and so they are usually happy to get involved in interviews. However, unless you have a system prototype to demonstrate, you should not expect stakeholders to suggest specific and detailed requirements. Everyone finds it difficult to visualize what a system might be like. You need to analyze the information collected and to generate the requirements from this.

Eliciting domain knowledge through interviews can be difficult, for two reasons:

1. All application specialists use jargon specific to their area of work. It is impossible for them to discuss domain requirements without using this terminology. They normally use words in a precise and subtle way that requirements engineers may misunderstand.

2. Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning. For example, for a librarian, it goes without saying that all acquisitions are catalogued before they are added to the library. However, this may not be obvious to the interviewer, and so it isn't taken into account in the requirements.

Interviews are not an effective technique for eliciting knowledge about organizational requirements and constraints because there are subtle power relationships between the different people in the organization. Published organizational structures rarely match the reality of decision making in an organization, but interviewees may not wish to reveal the actual rather than the theoretical structure to a stranger. In general, most people are generally reluctant to discuss political and organizational issues that may affect the requirements.

To be an effective interviewer, you should bear two things in mind:

1. You should be open-minded, avoid preconceived ideas about the requirements, and willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, then you should be willing to change your mind about the system.
2. You should prompt the interviewee to get discussions going by using a springboard question or a requirements proposal, or by working together on a prototype system. Saying to people "tell me what you want" is unlikely to result in useful information. They find it much easier to talk in a defined context rather than in general terms.

Information from interviews is used along with other information about the system from documentation describing business processes or existing systems, user observations, and developer experience. Sometimes, apart from the information in the system documents, the interview information may be the only source of information about the system requirements. However, interviewing on its own is liable to miss essential information, and so it should be used in conjunction with other requirements elicitation techniques.

4.3.1.2 Ethnography

Software systems do not exist in isolation. They are used in a social and organizational environment, and software system requirements may be generated or constrained by that environment. One reason why many software systems are delivered but never used is that their requirements do not take proper account of how social and organizational factors affect the practical operation of the system. It is therefore very important that, during the requirements engineering process, you try to understand the social and organizational issues that affect the use of the system.

Ethnography is an observational technique that can be used to understand operational processes and help derive requirements for software to support these processes. An analyst immerses himself or herself in the working environment where

the system will be used. The day-to-day work is observed, and notes are made of the actual tasks in which participants are involved. The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.

People often find it very difficult to articulate details of their work because it is second nature to them. They understand their own work but may not understand its relationship to other work in the organization. Social and organizational factors that affect the work, but that are not obvious to individuals, may only become clear when noticed by an unbiased observer. For example, a workgroup may self-organize so that members know of each other's work and can cover for each other if someone is absent. This may not be mentioned during an interview as the group might not see it as an integral part of their work.

Suchman (Suchman 1983) pioneered the use of ethnography to study office work. She found that actual work practices were far richer, more complex, and more dynamic than the simple models assumed by office automation systems. The difference between the assumed and the actual work was the most important reason why these office systems had no significant effect on productivity. Crabtree (Crabtree 2003) discusses a wide range of studies since then and describes, in general, the use of ethnography in systems design. In my own research, I have investigated methods of integrating ethnography into the software engineering process by linking it with requirements engineering methods (Viller and Sommerville 2000) and documenting patterns of interaction in cooperative systems (Martin and Sommerville 2004).

Ethnography is particularly effective for discovering two types of requirements:

1. Requirements derived from the way in which people actually work, rather than the way in which business process definitions say they ought to work. In practice, people never follow formal processes. For example, air traffic controllers may switch off a conflict alert system that detects aircraft with intersecting flight paths, even though normal control procedures specify that it should be used. The conflict alert system is sensitive and issues audible warnings even when planes are far apart. Controllers may find these distracting and prefer to use other strategies to ensure that planes are not on conflicting flight paths.

2. Requirements derived from cooperation and awareness of other people's activities. For example, air traffic controllers (ATCs) may use an awareness of other controlles' work to predict the number of aircraft that will be entering their control sector. They then modify their control strategies depending on that predicted workload. Therefore, an automated ATC system should allow controllers in a sector to have some visibility of the work in adjacent sectors.

Ethnography can be combined with the development of a system prototype (Figure 4.8). The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer. He or she should then look for the answers to these questions during the next phase of the system study (Sommerville et al. 1993).

Figure 4.8 Ethnography and prototyping for requirements analysis

Ethnography is helpful to understand existing systems, but this understanding does not always help with innovation. Innovation is particularly relevant for new product development. Commentators have suggested that Nokia used ethnography to discover how people used their phones and developed new phone models on that basis; Apple, on the other hand, ignored current use and revolutionized the mobile phone industry with the introduction of the iPhone.

Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques. However, because of its focus on the end-user, this approach is not effective for discovering broader organizational or domain requirements or for suggestion innovations. You therefore have to use ethnography as one of a number of techniques for requirements elicitation.

4.3.2 Stories and scenarios

People find it easier to relate to real-life examples than abstract descriptions. They are not good at telling you the system requirements. However, they may be able to describe how they handle particular situations or imagine things that they might do in a new way of working. Stories and scenarios are ways of capturing this kind of information. You can then use these when interviewing groups of stakeholders to discuss the system with other stakeholders and to develop more specific system requirements.

Stories and scenarios are essentially the same thing. They are a description of how the system can be used for some particular task. They describe what people do, what information they use and produce, and what systems they may use in this process. The difference is in the ways that descriptions are structured and in the level of detail presented. Stories are written as narrative text and present a high-level description of system use; scenarios are usually structured with specific information collected such as inputs and outputs. I find stories to be effective in setting out the "big picture." Parts of stories can then be developed in more detail and represented as scenarios.

Figure 4.9 is an example of a story that I developed to understand the requirements for the iLearn digital learning environment that I introduced in Chapter 1. This story describes a situation in a primary (elementary) school where the teacher is using the environment to support student projects on the fishing industry. You can see this is a very high-level description. Its purpose is to facilitate discussion of how the iLearn system might be used and to act as a starting point for eliciting the requirements for that system.

Photo sharing in the classroom

Jack is a primary school teacher in Ullapool (a village in northern Scotland). He has decided that a class project should be focused on the fishing industry in the area, looking at the history, development, and economic impact of fishing. As part of this project, pupils are asked to gather and share reminiscences from relatives, use newspaper archives, and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCRAN (a history resources site) to access newspaper archives and photographs. However, Jack also needs a photo-sharing site because he wants pupils to take and comment on each other's photos and to upload scans of old photographs that they may have in their families.

Jack sends an email to a primary school teachers' group, which he is a member of, to see if anyone can recommend an appropriate system. Two teachers reply, and both suggest that he use KidsTakePics, a photo-sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account. He uses the iLearn setup service to add KidsTakePics to the services seen by the pupils in his class so that when they log in, they can immediately use the system to upload photos from their mobile devices and class computers.

Figure 4.9 A user story for the iLearn system

The advantage of stories is that everyone can easily relate to them. We found this approach to be particularly useful to get information from a wider community than we could realistically interview. We made the stories available on a wiki and invited teachers and students from across the country to comment on them.

These high-level stories do not go into detail about a system, but they can be developed into more specific scenarios. Scenarios are descriptions of example user interaction sessions. I think that it is best to present scenarios in a structured way rather than as narrative text. User stories used in agile methods such as Extreme Programming, are actually narrative scenarios rather than general stories to help elicit requirements.

A scenario starts with an outline of the interaction. During the elicitation process, details are added to create a complete description of that interaction. At its most general, a scenario may include:

1. A description of what the system and users expect when the scenario starts.
2. A description of the normal flow of events in the scenario.
3. A description of what can go wrong and how resulting problems can be handled.
4. Information about other activities that might be going on at the same time.
5. A description of the system state when the scenario ends.

As an example of a scenario, Figure 4.10 describes what happens when a student uploads photos to the KidsTakePics system, as explained in Figure 4.9. The key difference between this system and other systems is that a teacher moderates the uploaded photos to check that they are suitable for sharing.

You can see this is a much more detailed description than the story in Figure 4.9, and so it can be used to propose requirements for the iLearn system. Like stories, scenarios can be used to facilitate discussions with stakeholders who sometimes may have different ways of achieving the same result.

Uploading photos to KidsTakePics

Initial assumption: A user or a group of users have one or more digital photographs to be uploaded to the picture-sharing site. These photos are saved on either a tablet or a laptop computer. They have successfully logged on to KidsTakePics.

Normal: The user chooses to upload photos and is prompted to select the photos to be uploaded on the computer and to select the project name under which the photos will be stored. Users should also be given the option of inputting keywords that should be associated with each uploaded photo. Uploaded photos are named by creating a conjunction of the user name with the filename of the photo on the local computer.

On completion of the upload, the system automatically sends an email to the project moderator, asking them to check new content, and generates an on-screen message to the user that this checking has been done.

What can go wrong: No moderator is associated with the selected project. An email is automatically generated to the school administrator asking them to nominate a project moderator. Users should be informed of a possible delay in making their photos visible.

Photos with the same name have already been uploaded by the same user. The user should be asked if he or she wishes to re-upload the photos with the same name, rename the photos, or cancel the upload. If users choose to re-upload the photos, the originals are overwritten. If they choose to rename the photos, a new name is automatically generated by adding a number to the existing filename.

Other activities: The moderator may be logged on to the system and may approve photos as they are uploaded.

System state on completion: User is logged on. The selected photos have been uploaded and assigned a status "awaiting moderation." Photos are visible to the moderator and to the user who uploaded them.

Figure 4.10 Scenario for uploading photos in KidsTakePics

4.4 Requirements specification

Requirements specification is the process of writing down the user and system requirements in a requirements document. Ideally, the user and system requirements should be clear, unambiguous, easy to understand, complete, and consistent. In practice, this is almost impossible to achieve. Stakeholders interpret the requirements in different ways, and there are often inherent conflicts and inconsistencies in the requirements.

User requirements are almost always written in natural language supplemented by appropriate diagrams and tables in the requirements document. System requirements may also be written in natural language, but other notations based on forms, graphical, or mathematical system models can also be used. Figure 4.11 summarizes possible notations for writing system requirements.

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge. Ideally, they should specify only the external behavior of the system. The requirements document should not include details of the system architecture or design. Consequently, if you are writing user requirements, you should not use software jargon, structured notations, or formal notations. You should write user requirements in natural language, with simple tables, forms, and intuitive diagrams.

Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system. UML (unified modeling language) use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want, and they are reluctant to accept it as a system contract. (I discuss this approach, in Chapter 10, which covers system dependability.)

Figure 4.11 Notations for writing system requirements

System requirements are expanded versions of the user requirements that software engineers use as the starting point for the system design. They add detail and explain how the system should provide the user requirements. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system.

Ideally, the system requirements should only describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. However, at the level of detail required to completely specify a complex software system, it is neither possible nor desirable to exclude all design information. There are several reasons for this:

1. You may have to design an initial architecture of the system to help structure the requirements specification. The system requirements are organized according to the different subsystems that make up the system. We did this when we were defining the requirements for the iLearn system, where we proposed the architecture shown in Figure 1.8.

2. In most cases, systems must interoperate with existing systems, which constrain the design and impose requirements on the new system.

3. The use of a specific architecture to satisfy non-functional requirements, such as N-version programming to achieve reliability, discussed in Chapter 11, may be necessary. An external regulator who needs to certify that the system is safe may specify that an architectural design that has already been certified should be used.

4.4.1 Natural language specification

Natural language has been used to write requirements for software since the 1950s. It is expressive, intuitive, and universal. It is also potentially vague and ambiguous, and its interpretation depends on the background of the reader. As a result, there

3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow, so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

Figure 4.12 Example requirements for the insulin pump software system

have been many proposals for alternative ways to write requirements. However, none of these proposals has been widely adopted, and natural language will continue to be the most widely used way of specifying system and software requirements.

To minimize misunderstandings when writing natural language requirements, I recommend that you follow these simple guidelines:

1. Invent a standard format and ensure that all requirement definitions adhere to that format. Standardizing the format makes omissions less likely and requirements easier to check. I suggest that, wherever possible, you should write the requirement in one or two sentences of natural language.
2. Use language consistently to distinguish between mandatory and desirable requirements. Mandatory requirements are requirements that the system must support and are usually written using "shall." Desirable requirements are not essential and are written using "should."
3. Use text highlighting (bold, italic, or color) to pick out key parts of the requirement.
4. Do not assume that readers understand technical, software engineering language. It is easy for words such as "architecture" and "module" to be misunderstood. Wherever possible, you should avoid the use of jargon, abbreviations, and acronyms.
5. Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included and who proposed the requirement (the requirement source), so that you know whom to consult if the requirement has to be changed. Requirements rationale is particularly useful when requirements are changed, as it may help decide what changes would be undesirable.

Figure 4.12 illustrates how these guidelines may be used. It includes two requirements for the embedded software for the automated insulin pump, introduced in Chapter 1. Other requirements for this embedded system are defined in the insulin pump requirements document, which can be downloaded from the book's web pages.

4.4.2 Structured specifications

Structured natural language is a way of writing system requirements where requirements are written in a standard way rather than as free-form text. This approach maintains most of the expressiveness and understandability of natural language but

Problems with using natural language for requirements specification

The flexibility of natural language, which is so useful for specification, often causes problems. There is scope for writing unclear requirements, and readers (the designers) may misinterpret requirements because they have a different background to the user. It is easy to amalgamate several requirements into a single sentence, and structuring natural language requirements can be difficult.

http://software-engineering-book.com/web/natural-language/

ensures that some uniformity is imposed on the specification. Structured language notations use templates to specify system requirements. The specification may use programming language constructs to show alternatives and iteration, and may highlight key elements using shading or different fonts.

The Robertsons (Robertson and Robertson 2013), in their book on the VOLERE requirements engineering method, recommend that user requirements be initially written on cards, one requirement per card. They suggest a number of fields on each card, such as the requirements rationale, the dependencies on other requirements, the source of the requirements, and supporting materials. This is similar to the approach used in the example of a structured specification shown in Figure 4.13.

To use a structured approach to specifying system requirements, you define one or more standard templates for requirements and represent these templates as structured forms. The specification may be structured around the objects manipulated by the system, the functions performed by the system, or the events processed by the system. An example of a form-based specification, in this case, one that defines how to calculate the dose of insulin to be delivered when the blood sugar is within a safe band, is shown in Figure 4.13.

When a standard format is used for specifying functional requirements, the following information should be included:

1. A description of the function or entity being specified.
2. A description of its inputs and the origin of these inputs.
3. A description of its outputs and the destination of these outputs.
4. Information about the information needed for the computation or other entities in the system that are required (the "requires" part).
5. A description of the action to be taken.
6. If a functional approach is used, a precondition setting out what must be true before the function is called, and a postcondition specifying what is true after the function is called.
7. A description of the side effects (if any) of the operation.

Using structured specifications removes some of the problems of natural language specification. Variability in the specification is reduced, and requirements are organized

Insulin Pump/Control Software/SRS/3.3.2

Function	Compute insulin dose: Safe sugar level.
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.
Inputs	Current sugar reading (r2), the previous two readings (r0 and r1).
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose—the dose in insulin to be delivered.
Destination	Main control loop.
Action:	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered. (see Figure 4.14)
Requires	Two previous readings so that the rate of change of sugar level can be computed.
Precondition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Postcondition	r0 is replaced by r1 then r1 is replaced by r2.
Side effects	None.

Figure 4.13 The structured specification of a requirement for an insulin pump

more effectively. However, it is still sometimes difficult to write requirements in a clear and unambiguous way, particularly when complex computations (e.g., how to calculate the insulin dose) are to be specified.

To address this problem, you can add extra information to natural language requirements, for example, by using tables or graphical models of the system. These can show how computations proceed, how the system state changes, how users interact with the system, and how sequences of actions are performed.

Tables are particularly useful when there are a number of possible alternative situations and you need to describe the actions to be taken for each of these. The insulin pump bases its computations of the insulin requirement on the rate of change of blood sugar levels. The rates of change are computed using the current and previous readings. Figure 4.14 is a tabular description of how the rate of change of blood sugar is used to calculate the amount of insulin to be delivered.

Figure 4.14 The tabular specification of computation in an insulin pump

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 − r1)<(r1 − r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing r2 > r1 & ((r2 − r1) ≥ (r1 − r0))	CompDose = round ((r2 − r1)/4) If rounded result = 0 then CompDose = MinimumDose

Figure 4.15 Use cases for the Mentcare system

4.4.3 Use cases

Use cases are a way of describing interactions between users and a system using a graphical model and structured text. They were first introduced in the Objectory method (Jacobsen et al. 1993) and have now become a fundamental feature of the Unified Modeling Language (UML). In their simplest form, a use case identifies the actors involved in an interaction and names the type of interaction. You then add additional information describing the interaction with the system. The additional information may be a textual description or one or more graphical models such as the UML sequence or state charts (see Chapter 5).

Use cases are documented using a high-level use case diagram. The set of use cases represents all of the possible interactions that will be described in the system requirements. Actors in the process, who may be human or other systems, are represented as stick figures. Each class of interaction is represented as a named ellipse. Lines link the actors with the interaction. Optionally, arrowheads may be added to lines to show how the interaction is initiated. This is illustrated in Figure 4.15, which shows some of the use cases for the Mentcare system.

Use cases identify the individual interactions between the system and its users or other systems. Each use case should be documented with a textual description. These can then be linked to other models in the UML that will develop the scenario in more detail. For example, a brief description of the Setup Consultation use case from Figure 4.15 might be:

> *Setup consultation allows two or more doctors, working in different offices, to view the same patient record at the same time. One doctor initiates the consultation by choosing the people involved from a dropdown menu of doctors who are online. The patient record is then displayed on their screens, but only the initiating doctor can edit the record. In addition, a text chat window is created*

to help coordinate actions. It is assumed that a phone call for voice communication can be separately arranged.

The UML is a standard for object-oriented modeling, so use cases and use case-based elicitation are used in the requirements engineering process. However, my experience with use cases is that they are too fine-grained to be useful in discussing requirements. Stakeholders don't understand the term *use case*; they don't find the graphical model to be useful, and they are often not interested in a detailed description of each and every system interaction. Consequently, I find use cases to be more helpful in systems design than in requirements engineering. I discuss use cases further in Chapter 5, which shows how they are used alongside other system models to document a system design.

Some people think that each use case is a single, low-level interaction scenario. Others, such as Stevens and Pooley (Stevens and Pooley 2006), suggest that each use case includes a set of related, low-level scenarios. Each of these scenarios is a single thread through the use case. Therefore, there would be a scenario for the normal interaction plus scenarios for each possible exception. In practice, you can use them in either way.

4.4.4 The software requirements document

The software requirements document (sometimes called the software requirements specification or SRS) is an official statement of what the system developers should implement. It may include both the user requirements for a system and a detailed specification of the system requirements. Sometimes the user and system requirements are integrated into a single description. In other cases, the user requirements are described in an introductory chapter in the system requirements specification.

Requirements documents are essential when systems are outsourced for development, when different teams develop different parts of the system, and when a detailed analysis of the requirements is mandatory. In other circumstances, such as software product or business system development, a detailed requirements document may not be needed.

Agile methods argue that requirements change so rapidly that a requirements document is out of date as soon as it is written, so the effort is largely wasted. Rather than a formal document, agile approaches often collect user requirements incrementally and write these on cards or whiteboards as short user stories. The user then prioritizes these stories for implementation in the next increment of the system.

For business systems where requirements are unstable, I think that this approach is a good one. However, I think that it is still useful to write a short supporting document that defines the business and dependability requirements for the system; it is easy to forget the requirements that apply to the system as a whole when focusing on the functional requirements for the next system release.

The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software. Figure 4.16 shows possible users of the document and how they use it.

Figure 4.16 Users of a requirements document

The diversity of possible users means that the requirements document has to be a compromise. It has to describe the requirements for customers, define the requirements in precise detail for developers and testers, as well as include information about future system evolution. Information on anticipated changes helps system designers to avoid restrictive design decisions and maintenance engineers to adapt the system to new requirements.

The level of detail that you should include in a requirements document depends on the type of system that is being developed and the development process used. Critical systems need detailed requirements because safety and security have to be analyzed in detail to find possible requirements errors. When the system is to be developed by a separate company (e.g., through outsourcing), the system specifications need to be detailed and precise. If an in-house, iterative development process is used, the requirements document can be less detailed. Details can be added to the requirements and ambiguities resolved during development of the system.

Figure 4.17 shows one possible organization for a requirements document that is based on an IEEE standard for requirements documents (IEEE 1998). This standard is a generic one that can be adapted to specific uses. In this case, the standard has been extended to include information about predicted system evolution. This information helps the maintainers of the system and allows designers to include support for future system features.

Chapter	Description
Preface	This defines the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This describes the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This defines the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter presents a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This describes the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This chapter includes graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This describes the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These provide detailed, specific information that is related to the application being developed—for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Figure 4.17 The structure of a requirements document

Naturally, the information included in a requirements document depends on the type of software being developed and the approach to development that is to be used. A requirements document with a structure like that shown in Figure 4.17 might be produced for a complex engineering system that includes hardware and software developed by different companies. The requirements document is likely to be long and detailed. It is therefore important that a comprehensive table of contents and document index be included so that readers can easily find the information they need.

By contrast, the requirements document for an in-house software product will leave out many of detailed chapters suggested above. The focus will be on defining the user requirements and high-level, nonfunctional system requirements. The system designers and programmers use their judgment to decide how to meet the outline user requirements for the system.

Requirements document standards

A number of large organizations, such as the U.S. Department of Defense and the IEEE, have defined standards for requirements documents. These are usually very generic but are nevertheless useful as a basis for developing more detailed organizational standards. The U.S. Institute of Electrical and Electronic Engineers (IEEE) is one of the best-known standards providers, and they have developed a standard for the structure of requirements documents. This standard is most appropriate for systems such as military command and control systems that have a long lifetime and are usually developed by a group of organizations.

http://software-engineering-book.com/web/requirements-standard/

4.5 Requirements validation

Requirements validation is the process of checking that requirements define the system that the customer really wants. It overlaps with elicitation and analysis, as it is concerned with finding problems with the requirements. Requirements validation is critically important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.

The cost of fixing a requirements problem by making a system change is usually much greater than repairing design or coding errors. A change to the requirements usually means that the system design and implementation must also be changed. Furthermore, the system must then be retested.

During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

1. *Validity checks* These check that the requirements reflect the real needs of system users. Because of changing circumstances, the user requirements may have changed since they were originally elicited.

2. *Consistency checks* Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.

3. *Completeness checks* The requirements document should include requirements that define all functions and the constraints intended by the system user.

4. *Realism checks* By using knowledge of existing technologies, the requirements should be checked to ensure that they can be implemented within the proposed budget for the system. These checks should also take account of the budget and schedule for the system development.

5. *Verifiability* To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

Requirements reviews

A requirements review is a process in which a group of people from the system customer and the system developer read the requirements document in detail and check for errors, anomalies, and inconsistencies. Once these have been detected and recorded, it is then up to the customer and the developer to negotiate how the identified problems should be solved.

http://software-engineering-book.com/web/requirements-reviews/

A number of requirements validation techniques can be used individually or in conjunction with one another:

1. *Requirements reviews* The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
2. *Prototyping* This involves developing an executable model of a system and using this with end-users and customers to see if it meets their needs and expectations. Stakeholders experiment with the system and feed back requirements changes to the development team.
3. *Test-case generation* Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of test-driven development.

You should not underestimate the problems involved in requirements validation. Ultimately, it is difficult to show that a set of requirements does in fact meet a user's needs. Users need to picture the system in operation and imagine how that system would fit into their work. It is hard even for skilled computer professionals to perform this type of abstract analysis and harder still for system users.

As a result, you rarely find all requirements problems during the requirements validation process. Inevitably, further requirements changes will be needed to correct omissions and misunderstandings after agreement has been reached on the requirements document.

4.6 Requirements change

The requirements for large software systems are always changing. One reason for the frequent changes is that these systems are often developed to address "wicked" problems—problems that cannot be completely defined (Rittel and Webber 1973). Because the problem cannot be fully defined, the software requirements are bound to

Figure 4.18
Requirements evolution

be incomplete. During the software development process, the stakeholders' understanding of the problem is constantly changing (Figure 4.18). The system requirements must then evolve to reflect this changed problem understanding.

Once a system has been installed and is regularly used, new requirements inevitably emerge. This is partly a consequence of errors and omissions in the original requirements that have to be corrected. However, most changes to system requirements arise because of changes to the business environment of the system:

1. The business and technical environment of the system always changes after installation. New hardware may be introduced and existing hardware updated. It may be necessary to interface the system with other systems. Business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that require system compliance.

2. The people who pay for a system and the users of that system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements, and, after delivery, new features may have to be added for user support if the system is to meet its goals.

3. Large systems usually have a diverse stakeholder community, with stakeholders having different requirements. Their priorities may be conflicting or contradictory. The final system requirements are inevitably a compromise, and some stakeholders have to be given priority. With experience, it is often discovered that the balance of support given to different stakeholders has to be changed and the requirements re-prioritized.

As requirements are evolving, you need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You therefore need a formal process for making change proposals and linking these to system requirements. This process of "requirements management" should start as soon as a draft version of the requirements document is available.

Agile development processes have been designed to cope with requirements that change during the development process. In these processes, when a user proposes a requirements change, this change does not go through a formal change management

Enduring and volatile requirements

Some requirements are more susceptible to change than others. Enduring requirements are the requirements that are associated with the core, slow-to-change activities of an organization. Enduring requirements are associated with fundamental work activities. Volatile requirements are more likely to change. They are usually associated with supporting activities that reflect how the organization does its work rather than the work itself.

http://software-engineering-book.com/web/changing-requirements/

process. Rather, the user has to prioritize that change and, if it is high priority, decide what system features that were planned for the next iteration should be dropped for the change to be implemented.

The problem with this approach is that users are not necessarily the best people to decide on whether or not a requirements change is cost-effective. In systems with multiple stakeholders, changes will benefit some stakeholders and not others. It is often better for an independent authority, who can balance the needs of all stakeholders, to decide on the changes that should be accepted.

4.6.1 Requirements management planning

Requirements management planning is concerned with establishing how a set of evolving requirements will be managed. During the planning stage, you have to decide on a number of issues:

1. *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and used in traceability assessments.
2. *A change management process* This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
3. *Traceability policies* These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.
4. *Tool support* Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to shared spreadsheets and simple database systems.

Requirements management needs automated support, and the software tools for this should be chosen during the planning phase. You need tool support for:

1. *Requirements storage* The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.

Figure 4.19 Requirements change management

2. *Change management* The process of change management (Figure 4.19) is simplified if active tool support is available. Tools can keep track of suggested changes and responses to these suggestions.

3. *Traceability management* As discussed above, tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

For small systems, you do not need to use specialized requirements management tools. Requirements management can be supported using shared web documents, spreadsheets, and databases. However, for larger systems, more specialized tool support, using systems such as DOORS (IBM 2013), makes it much easier to keep track of a large number of changing requirements.

4.6.2 Requirements change management

Requirements change management (Figure 4.19) should be applied to all proposed changes to a system's requirements after the requirements document has been approved. Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation. The advantage of using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way.

There are three principal stages to a change management process:

1. *Problem analysis and change specification* The process starts with an identified requirements problem or, sometimes, with a specific change proposal. During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.

2. *Change analysis and costing* The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made as to whether or not to proceed with the requirements change.

Requirements traceability

You need to keep track of the relationships between requirements, their sources, and the system design so that you can analyze the reasons for proposed changes and the impact that these changes are likely to have on other parts of the system. You need to be able to trace how a change ripples its way through the system. Why?

http://software-engineering-book.com/web/traceability/

3. *Change implementation* The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

If a new requirement has to be urgently implemented, there is always a temptation to change the system and then retrospectively modify the requirements document. This almost inevitably leads to the requirements specification and the system implementation getting out of step. Once system changes have been made, it is easy to forget to include these changes in the requirements document. In some circumstances, emergency changes to a system have to be made. In those cases, it is important that you update the requirements document as soon as possible in order to include the revised requirements.

KEY POINTS

- Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- Non-functional requirements often constrain the system being developed and the development process being used. These might be product requirements, organizational requirements, or external requirements. They often relate to the emergent properties of the system and therefore apply to the system as a whole.
- The requirements engineering process includes requirements elicitation, requirements specification, requirements validation, and requirements management.
- Requirements elicitation is an iterative process that can be represented as a spiral of activities—requirements discovery, requirements classification and organization, requirements negotiation, and requirements documentation.

- Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.
- The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.
- Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism, and verifiability.
- Business, organizational, and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.

FURTHER READING

"Integrated Requirements Engineering: A Tutorial." This is a tutorial paper that discusses requirements engineering activities and how these can be adapted to fit with modern software engineering practice. (I. Sommerville, *IEEE Software,* 22(1), January–February 2005) http://dx.doi.org/10.1109/MS.2005.13.

"Research Directions in Requirements Engineering." This is a good survey of requirements engineering research that highlights future research challenges in the area to address issues such as scale and agility. (B. H. C. Cheng and J. M. Atlee, *Proc. Conf. on Future of Software Engineering,* IEEE Computer Society, 2007) http://dx.doi.org/10.1109/FOSE.2007.17.

Mastering the Requirements Process, 3rd ed. A well-written, easy-to-read book that is based on a particular method (VOLERE) but that also includes lots of good general advice about requirements engineering. (S. Robertson and J. Robertson, 2013, Addison-Wesley).

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/requirements-and-design/

Requirements document for the insulin pump:

http://software-engineering-book.com/case-studies/insulin-pump/

Mentcare system requirements information:

http://software-engineering-book.com/case-studies/mentcare-system/

EXERCISES

4.1. Identify and briefly describe four types of requirements that may be defined for a computer-based system.

4.2. Discover ambiguities or omissions in the following statement of the requirements for part of a drone system intended for search and recovery:

The drone, a quad chopper, will be very useful in search and recovery operations, especially in remote areas or in extreme weather conditions. It will click high-resolution images. It will fly according to a path preset by a ground operator, but will be able to avoid obstacles on its own, returning to its original path whenever possible. The drone will also be able to identify various objects and match them to the target it is looking for.

4.3. Rewrite the above description using the structured approach described in this chapter. Resolve the identified ambiguities in a sensible way.

4.4. Write a set of non-functional requirements for the drone system, setting out its expected safety and response time.

4.5. Using the technique suggested here, where natural language descriptions are presented in a standard format, write plausible user requirements for the following functions:

An unattended petrol (gas) pump system that includes a credit card reader. The customer swipes the card through the reader, then specifies the amount of fuel required. The fuel is delivered and the customer's account debited.

The cash-dispensing function in a bank ATM.

In an Internet banking system, a facility that allows customers to transfer funds from one account held with the bank to another account with the same bank.

4.6. Suggest how an engineer responsible for drawing up a system requirements specification might keep track of the relationships between functional and non-functional requirements.

4.7. Using your knowledge of how an ATM is used, develop a set of use cases that could serve as a basis for understanding the requirements for an ATM system.

4.8. To minimize mistakes during a requirements review, an organization decides to allocate two scribes to document the review session. Explain how this can be done.

4.9. When emergency changes have to be made to systems, the system software may have to be modified before changes to the requirements have been approved. Suggest a model of a process for making these modifications that will ensure that the requirements document and the system implementation do not become inconsistent.

4.10. You have taken a job with a software user who has contracted your previous employer to develop a system for them. You discover that your company's interpretation of the requirements is different from the interpretation taken by your previous employer. Discuss what you

should do in such a situation. You know that the costs to your current employer will increase if the ambiguities are not resolved. However, you also have a responsibility of confidentiality to your previous employer.

REFERENCES

Crabtree, A. 2003. *Designing Collaborative Systems: A Practical Guide to Ethnography*. London: Springer-Verlag.

Davis, A. M. 1993. *Software Requirements: Objects, Functions and States*. Englewood Cliffs, NJ: Prentice-Hall.

IBM. 2013. "Rational Doors Next Generation: Requirements Engineering for Complex Systems." https://jazz.net/products/rational-doors-next-generation/

IEEE. 1998. "IEEE Recommended Practice for Software Requirements Specifications." In *IEEE Software Engineering Standards Collection*. Los Alamitos, CA: IEEE Computer Society Press.

Jacobsen, I., M. Christerson, P. Jonsson, and G. Overgaard. 1993. *Object-Oriented Software Engineering*. Wokingham, UK: Addison-Wesley.

Martin, D., and I. Sommerville. 2004. "Patterns of Cooperative Interaction: Linking Ethnomethodology and Design." *ACM Transactions on Computer-Human Interaction* 11 (1) (March 1): 59–89. doi:10.1145/972648.972651.

Rittel, H., and M. Webber. 1973. "Dilemmas in a General Theory of Planning." *Policy Sciences* 4: 155–169. doi:10.1007/BF01405730.

Robertson, S., and J. Robertson. 2013. *Mastering the Requirements Process, 3rd ed.* Boston: Addison-Wesley.

Sommerville, I., T. Rodden, P. Sawyer, R. Bentley, and M. Twidale. 1993. "Integrating Ethnography into the Requirements Engineering Process." In *RE'93*, 165–173. San Diego, CA: IEEE Computer Society Press. doi:10.1109/ISRE.1993.324821.

Stevens, P., and R. Pooley. 2006. *Using UML: Software Engineering with Objects and Components, 2nd ed.* Harlow, UK: Addison-Wesley.

Suchman, L. 1983. "Office Procedures as Practical Action: Models of Work and System Design." *ACM Transactions on Office Information Systems* 1 (3): 320–328. doi:10.1145/357442.357445.

Viller, S., and I. Sommerville. 2000. "Ethnographically Informed Analysis for Software Engineers." *Int. J. of Human-Computer Studies* 53 (1): 169–196. doi:10.1006/ijhc.2000.0370.

5

System modeling

Objectives

The aim of this chapter is to introduce system models that may be developed as part of requirements engineering and system design processes. When you have read the chapter, you will:

- understand how graphical models can be used to represent software systems and why several types of model are needed to fully represent a system;
- understand the fundamental system modeling perspectives of context, interaction, structure, and behavior;
- understand the principal diagram types in the Unified Modeling Language (UML) and how these diagrams may be used in system modeling;
- have been introduced to model-driven engineering, where an executable system is automatically generated from structural and behavioral models.

Contents

5.1 Context models
5.2 Interaction models
5.3 Structural models
5.4 Behavioral models
5.5 Model-driven engineering

IEEE Std 830-1998
(Revision of
IEEE Std 830-1993)

IEEE Recommended Practice for Software Requirements Specifications

Sponsor

**Software Engineering Standards Committee
of the
IEEE Computer Society**

Approved 25 June 1998

IEEE-SA Standards Board

Abstract: The content and qualities of a good software requirements specification (SRS) are described and several sample SRS outlines are presented. This recommended practice is aimed at specifying requirements of software to be developed but also can be applied to assist in the selection of in-house and commercial software products. Guidelines for compliance with IEEE/EIA 12207.1-1997 are also provided.
Keywords: contract, customer, prototyping, software requirements specification, supplier, system requirements specifications

---

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

Copyright © 1998 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 1998. Printed in the United States of America.

ISBN 0-7381-0332-2

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; (978) 750-8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Introduction

(This introduction is not a part of IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications.)

This recommended practice describes recommended approaches for the specification of software requirements. It is based on a model in which the result of the software requirements specification process is an unambiguous and complete specification document. It should help

a) Software customers to accurately describe what they wish to obtain;
b) Software suppliers to understand exactly what the customer wants;
c) Individuals to accomplish the following goals:
    1) Develop a standard software requirements specification (SRS) outline for their own organizations;
    2) Define the format and content of their specific software requirements specifications;
    3) Develop additional local supporting items such as an SRS quality checklist, or an SRS writer's handbook.

To the customers, suppliers, and other individuals, a good SRS should provide several specific benefits, such as the following:

— *Establish the basis for agreement between the customers and the suppliers on what the software product is to do*. The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs.

— *Reduce the development effort*. The preparation of the SRS forces the various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.

— *Provide a basis for estimating costs and schedules*. The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates.

— *Provide a baseline for validation and verification*. Organizations can develop their validation and verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured.

— *Facilitate transfer*. The SRS makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.

— *Serve as a basis for enhancement*. Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued production evaluation.

The readers of this document are referred to Annex B for guidelines for using this recommended practice to meet the requirements of IEEE/EIA 12207.1-1997, IEEE/EIA Guide—Industry Implementation of ISO/IEC 12207: 1995, Standard for Information Technology—Software life cycle processes—Life cycle data.

Copyright © 1998 IEEE. All rights reserved.

Participants

This recommended practice was prepared by the Life Cycle Data Harmonization Working Group of the Software Engineering Standards Committee of the IEEE Computer Society. At the time this recommended practice was approved, the working group consisted of the following members:

Leonard L. Tripp, *Chair*

Edward Byrne
Paul R. Croll
Perry DeWeese
Robin Fralick
Marilyn Ginsberg-Finner
John Harauz
Mark Henley
Dennis Lawrence
David Maibor
Ray Milovanovic
James Moore
Timothy Niesen
Dennis Rilling
Terry Rout
Richard Schmidt
Norman F. Schneidewind
David Schultz
Basil Sherlund
Peter Voldner
Ronald Wade

The following persons were on the balloting committee:

Syed Ali
Theodore K. Atchinson
Mikhail Auguston
Robert E. Barry
Leo Beltracchi
H. Ronald Berlack
Richard E. Biehl
Michael A. Blackledge
Sandro Bologna
Juris Borzovs
Kathleen L. Briggs
M. Scott Buck
Michael Caldwell
James E. Cardow
Enrico A. Carrara
Lawrence Catchpole
Keith Chan
Antonio M. Cicu
Theo Clarke
Sylvain Clermont
Rosemary Coleman
Virgil Lee Cooper
W. W. Geoff Cozens
Paul R. Croll
Gregory T. Daich
Geoffrey Darnton
Taz Daughtrey
Bostjan K. Derganc
Perry R. DeWeese
James Do
Evelyn S. Dow
Carl Einar Dragstedt
Sherman Eagles
Christof Ebert
Leo Egan
Richard E. Fairley
John W. Fendrich
Jay Forster
Kirby Fortenberry
Eva Freund
Richard C. Fries
Roger U. Fujii
Adel N. Ghannam
Marilyn Ginsberg-Finner
John Garth Glynn
Julio Gonzalez-Sanz
L. M. Gunther
David A. Gustafson
Jon D. Hagar
John Harauz
Robert T. Harley
Herbert Hecht
William Hefley
Manfred Hein
Mark Heinrich
Mark Henley
Debra Herrmann
John W. Horch
Jerry Huller
Peter L. Hung
George Jackelen
Frank V. Jorgensen
William S. Junk
George X. Kambic
Richard Karcich
Ron S. Kenett
Judith S. Kerner
Robert J. Kierzyk
Dwayne L. Knirk
Shaye Koenig
Thomas M. Kurihara
John B. Lane
J. Dennis Lawrence
Fang Ching Lim
William M. Lively
James J. Longbucco
Dieter Look
John Lord
Stan Magee
David Maibor
Harold Mains
Robert A. Martin
Tomoo Matsubara
Mike McAndrew
Patrick D. McCray
Christopher McMacken
Jerome W. Mersky
Bret Michael
Alan Miller
Celia H. Modell
James W. Moore
Pavol Navrat
Myrna L. Olson
Indradeb P. Pal
Alex Polack
Peter T. Poon
Lawrence S. Przybylski
Kenneth R. Ptack
Annette D. Reilly
Dennis Rilling
Andrew P. Sage
Helmut Sandmayr
Stephen R. Schach
Hans Schaefer
Norman Schneidewind
David J. Schultz
Lisa A. Selmon
Robert W. Shillato
David M. Siefert
Carl A. Singer
James M. Sivak
Richard S. Sky
Nancy M. Smith
Melford E. Smyre
Harry M. Sneed
Alfred R. Sorkowitz
Donald W. Sova
Luca Spotorno
Julia Stesney
Fred J. Strauss
Christine Brown Strysik
Toru Takeshita
Richard H. Thayer
Booker Thomas
Patricia Trellue
Theodore J. Urbanowicz
Glenn D. Venables
Udo Voges
David D. Walden
Dolores Wallace
William M. Walsh
John W. Walz
Camille SWhite-Partain
Scott A. Whitmire
P. A. Wolfgang
Paul R. Work
Natalie C. Yopconka
Janusz Zalewski
Geraldine Zimmerman
Peter F. Zoll

Copyright © 1998 IEEE. All rights reserved.

When the IEEE-SA Standards Board approved this recommended practice on 25 June 1998, it had the following membership:

Richard J. Holleman, *Chair* **Donald N. Heirman,** *Vice Chair*
Judith Gorman, *Secretary*

Satish K. Aggarwal
Clyde R. Camp
James T. Carlo
Gary R. Engmann
Harold E. Epstein
Jay Forster*
Thomas F. Garrity
Ruben D. Garzon
James H. Gurney
Jim D. Isaak
Lowell G. Johnson
Robert Kennelly
E. G. "Al" Kiener
Joseph L. Koepfinger*
Stephen R. Lambert
Jim Logothetis
Donald C. Loughry
L. Bruce McClung
Louis-François Pau
Ronald C. Petersen
Gerald H. Peterson
John B. Posey
Gary S. Robinson
Hans E. Weinrich
Donald W. Zipse

*Member Emeritus

Valerie E. Zelenty
IEEE Standards Project Editor

Copyright © 1998 IEEE. All rights reserved.

Contents

1. Overview 1

1.1 Scope 1

2. References 2

3. Definitions 2

4. Considerations for producing a good SRS 3

4.1 Nature of the SRS 3
4.2 Environment of the SRS 3
4.3 Characteristics of a good SRS 4
4.4 Joint preparation of the SRS 8
4.5 SRS evolution 8
4.6 Prototyping 9
4.7 Embedding design in the SRS 9
4.8 Embedding project requirements in the SRS 10

5. The parts of an SRS 10

5.1 Introduction (Section 1 of the SRS) 11
5.2 Overall description (Section 2 of the SRS) 12
5.3 Specific requirements (Section 3 of the SRS) 15
5.4 Supporting information 19

Annex A (informative) SRS templates 21

Annex B (informative) Guidelines for compliance with IEEE/EIA 12207.1-1997 27

Copyright © 1998 IEEE. All rights reserved.

IEEE Recommended Practice for Software Requirements Specifications

1. Overview

This recommended practice describes recommended approaches for the specification of software requirements. It is divided into five clauses. Clause 1 explains the scope of this recommended practice. Clause 2 lists the references made to other standards. Clause 3 provides definitions of specific terms used. Clause 4 provides background information for writing a good SRS. Clause 5 discusses each of the essential parts of an SRS. This recommended practice also has two annexes, one which provides alternate format templates, and one which provides guidelines for compliance with IEEE/EIA 12207.1-1997.

1.1 Scope

This is a recommended practice for writing software requirements specifications. It describes the content and qualities of a good software requirements specification (SRS) and presents several sample SRS outlines.

This recommended practice is aimed at specifying requirements of software to be developed but also can be applied to assist in the selection of in-house and commercial software products. However, application to already-developed software could be counterproductive.

When software is embedded in some larger system, such as medical equipment, then issues beyond those identified in this recommended practice may have to be addressed.

This recommended practice describes the process of creating a product and the content of the product. The product is an SRS. This recommended practice can be used to create such an SRS directly or can be used as a model for a more specific standard.

This recommended practice does not identify any specific method, nomenclature, or tool for preparing an SRS.

Copyright © 1998 IEEE. All rights reserved.

2. References

This recommended practice shall be used in conjunction with the following publications.

ASTM E1340-96, Standard Guide for Rapid Prototyping of Computerized Systems.[1]

IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.[2]

IEEE Std 730-1998, IEEE Standard for Software Quality Assurance Plans.

IEEE Std 730.1-1995, IEEE Guide for Software Quality Assurance Planning.

IEEE Std 828-1998, IEEE Standard for Software Configuration Management Plans.[3]

IEEE Std 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software.

IEEE Std 982.2-1988, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software.

IEEE Std 1002-1987 (Reaff 1992), IEEE Standard Taxonomy for Software Engineering Standards.

IEEE Std 1012-1998, IEEE Standard for Software Verification and Validation.

IEEE Std 1012a-1998, IEEE Standard for Software Verification and Validation: Content Map to IEEE/EIA 12207.1-1997.[4]

IEEE Std 1016-1998, IEEE Recommended Practice for Software Design Descriptions.[5]

IEEE Std 1028-1997, IEEE Standard for Software Reviews.

IEEE Std 1042-1987 (Reaff 1993), IEEE Guide to Software Configuration Management.

IEEE P1058/D2.1, Draft Standard for Software Project Management Plans, dated 5 August 1998.[6]

IEEE Std 1058a-1998, IEEE Standard for Software Project Management Plans: Content Map to IEEE/EIA 12207.1-1997.[7]

IEEE Std 1074-1997, IEEE Standard for Developing Software Life Cycle Processes.

IEEE Std 1233, 1998 Edition, IEEE Guide for Developing System Requirements Specifications.[8]

---

[1]ASTM publications are available from the American Society for Testing and Materials, 100 Barr Harbor Drive, West Conshohocken, PA 19428-2959, USA.

[2]IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

[3]As this standard goes to press, IEEE Std 828-1998; IEEE Std 1012a-1998; IEEE Std 1016-1998; and IEEE Std 1233, 1998 Edition are approved but not yet published. The draft standards are, however, available from the IEEE. Anticipated publication date is Fall 1998. Contact the IEEE Standards Department at 1 (732) 562-3800 for status information.

[4]See Footnote 3.

[5]See Footnote 3.

[6]Upon approval of IEEE P1058 by the IEEE-SA Standards Board, this standard will be integrated with IEEE Std 1058a-1998 and published as IEEE Std 1058, 1998 Edition. Approval is expected 8 December 1998.

[7]As this standard goes to press, IEEE Std 1058a-1998 is approved but not yet published. The draft standard is, however, available from the IEEE. Anticipated publication date is December 1998. Contact the IEEE Standards Department at 1 (732) 562-3800 for status information. See Footnote 6.

[8]See Footnote 3.

 Copyright © 1998 IEEE. All rights reserved.

3. Definitions

In general the definitions of terms used in this recommended practice conform to the definitions provided in IEEE Std 610.12-1990. The definitions below are key terms as they are used in this recommended practice.

3.1 contract: A legally binding document agreed upon by the customer and supplier. This includes the technical and organizational requirements, cost, and schedule for a product. A contract may also contain informal but useful information such as the commitments or expectations of the parties involved.

3.2 customer: The person, or persons, who pay for the product and usually (but not necessarily) decide the requirements. In the context of this recommended practice the customer and the supplier may be members of the same organization.

3.3 supplier: The person, or persons, who produce a product for a customer. In the context of this recommended practice, the customer and the supplier may be members of the same organization.

3.4 user: The person, or persons, who operate or interact directly with the product. The user(s) and the customer(s) are often not the same person(s).

4. Considerations for producing a good SRS

This clause provides background information that should be considered when writing an SRS. This includes the following:

a) Nature of the SRS;
b) Environment of the SRS;
c) Characteristics of a good SRS;
d) Joint preparation of the SRS;
e) SRS evolution;
f) Prototyping;
g) Embedding design in the SRS;
h) Embedding project requirements in the SRS.

4.1 Nature of the SRS

The SRS is a specification for a particular software product, program, or set of programs that performs certain functions in a specific environment. The SRS may be written by one or more representatives of the supplier, one or more representatives of the customer, or by both. Subclause 4.4 recommends both.

The basic issues that the SRS writer(s) shall address are the following:

a) *Functionality.* What is the software supposed to do?
b) *External interfaces*. How does the software interact with people, the system's hardware, other hardware, and other software?
c) *Performance*. What is the speed, availability, response time, recovery time of various software functions, etc.?
d) *Attributes*. What are the portability, correctness, maintainability, security, etc. considerations?
e) *Design constraints imposed on an implementation*. Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

The SRS writer(s) should avoid placing either design or project requirements in the SRS.

For recommended contents of an SRS see Clause 5.

Copyright © 1998 IEEE. All rights reserved.

4.2 Environment of the SRS

It is important to consider the part that the SRS plays in the total project plan, which is defined in IEEE Std 610.12-1990. The software may contain essentially all the functionality of the project or it may be part of a larger system. In the latter case typically there will be an SRS that will state the interfaces between the system and its software portion, and will place external performance and functionality requirements upon the software portion. Of course the SRS should then agree with and expand upon these system requirements.

IEEE Std 1074-1997 describes the steps in the software life cycle and the applicable inputs for each step. Other standards, such as those listed in Clause 2, relate to other parts of the software life cycle and so may complement software requirements.

Since the SRS has a specific role to play in the software development process, the SRS writer(s) should be careful not to go beyond the bounds of that role. This means the SRS

a) Should correctly define all of the software requirements. A software requirement may exist because of the nature of the task to be solved or because of a special characteristic of the project.
b) Should not describe any design or implementation details. These should be described in the design stage of the project.
c) Should not impose additional constraints on the software. These are properly specified in other documents such as a software quality assurance plan.

Therefore, a properly written SRS limits the range of valid designs, but does not specify any particular design.

4.3 Characteristics of a good SRS

An SRS should be

a) Correct;
b) Unambiguous;
c) Complete;
d) Consistent;
e) Ranked for importance and/or stability;
f) Verifiable;
g) Modifiable;
h) Traceable.

4.3.1 Correct

An SRS is correct if, and only if, every requirement stated therein is one that the software shall meet.

There is no tool or procedure that ensures correctness. The SRS should be compared with any applicable superior specification, such as a system requirements specification, with other project documentation, and with other applicable standards, to ensure that it agrees. Alternatively the customer or user can determine if the SRS correctly reflects the actual needs. Traceability makes this procedure easier and less prone to error (see 4.3.8).

4.3.2 Unambiguous

An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. As a minimum, this requires that each characteristic of the final product be described using a single unique term.

 Copyright © 1998 IEEE. All rights reserved.

In cases where a term used in a particular context could have multiple meanings, the term should be included in a glossary where its meaning is made more specific.

An SRS is an important part of the requirements process of the software life cycle and is used in design, implementation, project monitoring, verification and validation, and in training as described in IEEE Std 1074-1997. The SRS should be unambiguous both to those who create it and to those who use it. However, these groups often do not have the same background and therefore do not tend to describe software requirements the same way. Representations that improve the requirements specification for the developer may be counterproductive in that they diminish understanding to the user and vice versa.

Subclauses 4.3.2.1 through 4.3.2.3 recommend how to avoid ambiguity.

4.3.2.1 Natural language pitfalls

Requirements are often written in natural language (e.g., English). Natural language is inherently ambiguous. A natural language SRS should be reviewed by an independent party to identify ambiguous use of language so that it can be corrected.

4.3.2.2 Requirements specification languages

One way to avoid the ambiguity inherent in natural language is to write the SRS in a particular requirements specification language. Its language processors automatically detect many lexical, syntactic, and semantic errors.

One disadvantage in the use of such languages is the length of time required to learn them. Also, many nontechnical users find them unintelligible. Moreover, these languages tend to be better at expressing certain types of requirements and addressing certain types of systems. Thus, they may influence the requirements in subtle ways.

4.3.2.3 Representation tools

In general, requirements methods and languages and the tools that support them fall into three general categories—object, process, and behavioral. Object-oriented approaches organize the requirements in terms of real-world objects, their attributes, and the services performed by those objects. Process-based approaches organize the requirements into hierarchies of functions that communicate via data flows. Behavioral approaches describe external behavior of the system in terms of some abstract notion (such as predicate calculus), mathematical functions, or state machines.

The degree to which such tools and methods may be useful in preparing an SRS depends upon the size and complexity of the program. No attempt is made here to describe or endorse any particular tool.

When using any of these approaches it is best to retain the natural language descriptions. That way, customers unfamiliar with the notations can still understand the SRS.

4.3.3 Complete

An SRS is complete if, and only if, it includes the following elements:

a) All significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces. In particular any external requirements imposed by a system specification should be acknowledged and treated.

Copyright © 1998 IEEE. All rights reserved.

b) Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to both valid and invalid input values.

c) Full labels and references to all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.

4.3.3.1 Use of TBDs

Any SRS that uses the phrase "to be determined" (TBD) is not a complete SRS. The TBD is, however, occasionally necessary and should be accompanied by

a) A description of the conditions causing the TBD (e.g., why an answer is not known) so that the situation can be resolved;

b) A description of what must be done to eliminate the TBD, who is responsible for its elimination, and by when it must be eliminated.

4.3.4 Consistent

Consistency refers to internal consistency. If an SRS does not agree with some higher-level document, such as a system requirements specification, then it is not correct (see 4.3.1).

4.3.4.1 Internal consistency

An SRS is internally consistent if, and only if, no subset of individual requirements described in it conflict. The three types of likely conflicts in an SRS are as follows:

a) The specified characteristics of real-world objects may conflict. For example,
   1) The format of an output report may be described in one requirement as tabular but in another as textual.
   2) One requirement may state that all lights shall be green while another may state that all lights shall be blue.

b) There may be logical or temporal conflict between two specified actions. For example,
   1) One requirement may specify that the program will add two inputs and another may specify that the program will multiply them.
   2) One requirement may state that "A" must always follow "B," while another may require that "A and B" occur simultaneously.

c) Two or more requirements may describe the same real-world object but use different terms for that object. For example, a program's request for a user input may be called a "prompt" in one requirement and a "cue" in another. The use of standard terminology and definitions promotes consistency.

4.3.5 Ranked for importance and/or stability

An SRS is ranked for importance and/or stability if each requirement in it has an identifier to indicate either the importance or stability of that particular requirement.

Typically, all of the requirements that relate to a software product are not equally important. Some requirements may be essential, especially for life-critical applications, while others may be desirable.

 Copyright © 1998 IEEE. All rights reserved.

Each requirement in the SRS should be identified to make these differences clear and explicit. Identifying the requirements in the following manner helps:

a) Have customers give more careful consideration to each requirement, which often clarifies any hidden assumptions they may have.
b) Have developers make correct design decisions and devote appropriate levels of effort to the different parts of the software product.

4.3.5.1 Degree of stability

One method of identifying requirements uses the dimension of stability. Stability can be expressed in terms of the number of expected changes to any requirement based on experience or knowledge of forthcoming events that affect the organization, functions, and people supported by the software system.

4.3.5.2 Degree of necessity

Another way to rank requirements is to distinguish classes of requirements as essential, conditional, and optional.

a) *Essential*. Implies that the software will not be acceptable unless these requirements are provided in an agreed manner.
b) *Conditional*. Implies that these are requirements that would enhance the software product, but would not make it unacceptable if they are absent.
c) *Optional*. Implies a class of functions that may or may not be worthwhile. This gives the supplier the opportunity to propose something that exceeds the SRS.

4.3.6 Verifiable

An SRS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement. In general any ambiguous requirement is not verifiable.

Nonverifiable requirements include statements such as “works well,” “good human interface,” and “shall usually happen.” These requirements cannot be verified because it is impossible to define the terms “good,” “well,” or “usually.” The statement that “the program shall never enter an infinite loop” is nonverifiable because the testing of this quality is theoretically impossible.

An example of a verifiable statement is

> *Output of the program shall be produced within 20 s of event × 60% of the time; and shall be produced within 30 s of event × 100% of the time.*

This statement can be verified because it uses concrete terms and measurable quantities.

If a method cannot be devised to determine whether the software meets a particular requirement, then that requirement should be removed or revised.

Copyright © 1998 IEEE. All rights reserved.

4.3.7 Modifiable

An SRS is modifiable if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style. Modifiability generally requires an SRS to

a) Have a coherent and easy-to-use organization with a table of contents, an index, and explicit cross-referencing;
b) Not be redundant (i.e., the same requirement should not appear in more than one place in the SRS);
c) Express each requirement separately, rather than intermixed with other requirements.

Redundancy itself is not an error, but it can easily lead to errors. Redundancy can occasionally help to make an SRS more readable, but a problem can arise when the redundant document is updated. For instance, a requirement may be altered in only one of the places where it appears. The SRS then becomes inconsistent. Whenever redundancy is necessary, the SRS should include explicit cross-references to make it modifiable.

4.3.8 Traceable

An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. The following two types of traceability are recommended:

a) *Backward traceability (i.e., to previous stages of development)*. This depends upon each requirement explicitly referencing its source in earlier documents.
b) *Forward traceability (i.e., to all documents spawned by the SRS)*. This depends upon each requirement in the SRS having a unique name or reference number.

The forward traceability of the SRS is especially important when the software product enters the operation and maintenance phase. As code and design documents are modified, it is essential to be able to ascertain the complete set of requirements that may be affected by those modifications.

4.4 Joint preparation of the SRS

The software development process should begin with supplier and customer agreement on what the completed software must do. This agreement, in the form of an SRS, should be jointly prepared. This is important because usually neither the customer nor the supplier is qualified to write a good SRS alone.

a) Customers usually do not understand the software design and development process well enough to write a usable SRS.
b) Suppliers usually do not understand the customer's problem and field of endeavor well enough to specify requirements for a satisfactory system.

Therefore, the customer and the supplier should work together to produce a well-written and completely understood SRS.

A special situation exists when a system and its software are both being defined concurrently. Then the functionality, interfaces, performance, and other attributes and constraints of the software are not predefined, but rather are jointly defined and subject to negotiation and change. This makes it more difficult, but no less important, to meet the characteristics stated in 4.3. In particular, an SRS that does not comply with the requirements of its parent system specification is incorrect.

 Copyright © 1998 IEEE. All rights reserved.

This recommended practice does not specifically discuss style, language usage, or techniques of good writing. It is quite important, however, that an SRS be well written. General technical writing books can be used for guidance.

4.5 SRS evolution

The SRS may need to evolve as the development of the software product progresses. It may be impossible to specify some details at the time the project is initiated (e.g., it may be impossible to define all of the screen formats for an interactive program during the requirements phase). Additional changes may ensue as deficiencies, shortcomings, and inaccuracies are discovered in the SRS.

Two major considerations in this process are the following:

a) Requirements should be specified as completely and thoroughly as is known at the time, even if evolutionary revisions can be foreseen as inevitable. The fact that they are incomplete should be noted.
b) A formal change process should be initiated to identify, control, track, and report projected changes. Approved changes in requirements should be incorporated in the SRS in such a way as to
    1) Provide an accurate and complete audit trail of changes;
    2) Permit the review of current and superseded portions of the SRS.

4.6 Prototyping

Prototyping is used frequently during the requirements portion of a project. Many tools exist that allow a prototype, exhibiting some characteristics of a system, to be created very quickly and easily. See also ASTM E1340-96.

Prototypes are useful for the following reasons:

a) The customer may be more likely to view the prototype and react to it than to read the SRS and react to it. Thus, the prototype provides quick feedback.
b) The prototype displays unanticipated aspects of the systems behavior. Thus, it produces not only answers but also new questions. This helps reach closure on the SRS.
c) An SRS based on a prototype tends to undergo less change during development, thus shortening development time.

A prototype should be used as a way to elicit software requirements. Some characteristics such as screen or report formats can be extracted directly from the prototype. Other requirements can be inferred by running experiments with the prototype.

4.7 Embedding design in the SRS

A requirement specifies an externally visible function or attribute of a system. A design describes a particular subcomponent of a system and/or its interfaces with other subcomponents. The SRS writer(s) should clearly distinguish between identifying required design constraints and projecting a specific design. Note that every requirement in the SRS limits design alternatives. This does not mean, though, that every requirement is design.

Copyright © 1998 IEEE. All rights reserved.

The SRS should specify what functions are to be performed on what data to produce what results at what location for whom. The SRS should focus on the services to be performed. The SRS should not normally specify design items such as the following:

a) Partitioning the software into modules;
b) Allocating functions to the modules;
c) Describing the flow of information or control between modules;
d) Choosing data structures.

4.7.1 Necessary design requirements

In special cases some requirements may severely restrict the design. For example, security or safety requirements may reflect directly into design such as the need to

a) Keep certain functions in separate modules;
b) Permit only limited communication between some areas of the program;
c) Check data integrity for critical variables.

Examples of valid design constraints are physical requirements, performance requirements, software development standards, and software quality assurance standards.

Therefore, the requirements should be stated from a purely external viewpoint. When using models to illustrate the requirements, remember that the model only indicates the external behavior, and does not specify a design.

4.8 Embedding project requirements in the SRS

The SRS should address the software product, not the process of producing the software product.

Project requirements represent an understanding between the customer and the supplier about contractual matters pertaining to production of software and thus should not be included in the SRS. These normally include items such as

a) Cost;
b) Delivery schedules;
c) Reporting procedures;
d) Software development methods;
e) Quality assurance;
f) Validation and verification criteria;
g) Acceptance procedures.

Project requirements are specified in other documents, typically in a software development plan, a software quality assurance plan, or a statement of work.

5. The parts of an SRS

This clause discusses each of the essential parts of the SRS. These parts are arranged in Figure 1 in an outline that can serve as an example for writing an SRS.

While an SRS does not have to follow this outline or use the names given here for its parts, a good SRS should include all the information discussed here.

 Copyright © 1998 IEEE. All rights reserved.

Table of Contents

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, acronyms, and abbreviations
   - 1.4 References
   - 1.5 Overview
2. Overall description
   - 2.1 Product perspective
   - 2.2 Product functions
   - 2.3 User characteristics
   - 2.4 Constraints
   - 2.5 Assumptions and dependencies
3. Specific requirements (See 5.3.1 through 5.3.8 for explanations of possible specific requirements. See also Annex A for several different ways of organizing this section of the SRS.)

Appendixes

Index

Figure 1—Prototype SRS outline

5.1 Introduction (Section 1 of the SRS)

The introduction of the SRS should provide an overview of the entire SRS. It should contain the following subsections:

a) Purpose;
b) Scope;
c) Definitions, acronyms, and abbreviations;
d) References;
e) Overview.

5.1.1 Purpose (1.1 of the SRS)

This subsection should

a) Delineate the purpose of the SRS;
b) Specify the intended audience for the SRS.

5.1.2 Scope (1.2 of the SRS)

This subsection should

a) Identify the software product(s) to be produced by name (e.g., Host DBMS, Report Generator, etc.);
b) Explain what the software product(s) will, and, if necessary, will not do;
c) Describe the application of the software being specified, including relevant benefits, objectives, and goals;
d) Be consistent with similar statements in higher-level specifications (e.g., the system requirements specification), if they exist.

Copyright © 1998 IEEE. All rights reserved.

5.1.3 Definitions, acronyms, and abbreviations (1.3 of the SRS)

This subsection should provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the SRS. This information may be provided by reference to one or more appendixes in the SRS or by reference to other documents.

5.1.4 References (1.4 of the SRS)

This subsection should

a) Provide a complete list of all documents referenced elsewhere in the SRS;
b) Identify each document by title, report number (if applicable), date, and publishing organization;
c) Specify the sources from which the references can be obtained.

This information may be provided by reference to an appendix or to another document.

5.1.5 Overview (1.5 of the SRS)

This subsection should

a) Describe what the rest of the SRS contains;
b) Explain how the SRS is organized.

5.2 Overall description (Section 2 of the SRS)

This section of the SRS should describe the general factors that affect the product and its requirements. This section does not state specific requirements. Instead, it provides a background for those requirements, which are defined in detail in Section 3 of the SRS, and makes them easier to understand.

This section usually consists of six subsections, as follows:

a) Product perspective;
b) Product functions;
c) User characteristics;
d) Constraints;
e) Assumptions and dependencies;
f) Apportioning of requirements.

5.2.1 Product perspective (2.1 of the SRS)

This subsection of the SRS should put the product into perspective with other related products. If the product is independent and totally self-contained, it should be so stated here. If the SRS defines a product that is a component of a larger system, as frequently occurs, then this subsection should relate the requirements of that larger system to functionality of the software and should identify interfaces between that system and the software.

A block diagram showing the major components of the larger system, interconnections, and external interfaces can be helpful.

 Copyright © 1998 IEEE. All rights reserved.

This subsection should also describe how the software operates inside various constraints. For example, these constraints could include

a) System interfaces;
b) User interfaces;
c) Hardware interfaces;
d) Software interfaces;
e) Communications interfaces;
f) Memory;
g) Operations;
h) Site adaptation requirements.

5.2.1.1 System interfaces

This should list each system interface and identify the functionality of the software to accomplish the system requirement and the interface description to match the system.

5.2.1.2 User interfaces

This should specify the following:

a) *The logical characteristics of each interface between the software product and its users*. This includes those configuration characteristics (e.g., required screen formats, page or window layouts, content of any reports or menus, or availability of programmable function keys) necessary to accomplish the software requirements.

b) *All the aspects of optimizing the interface with the person who must use the system*. This may simply comprise a list of do's and don'ts on how the system will appear to the user. One example may be a requirement for the option of long or short error messages. Like all others, these requirements should be verifiable, e.g., "a clerk typist grade 4 can do function *X* in *Z* min after 1 h of training" rather than "a typist can do function *X*." (This may also be specified in the Software System Attributes under a section titled Ease of Use.)

5.2.1.3 Hardware interfaces

This should specify the logical characteristics of each interface between the software product and the hardware components of the system. This includes configuration characteristics (number of ports, instruction sets, etc.). It also covers such matters as what devices are to be supported, how they are to be supported, and protocols. For example, terminal support may specify full-screen support as opposed to line-by-line support.

5.2.1.4 Software interfaces

This should specify the use of other required software products (e.g., a data management system, an operating system, or a mathematical package), and interfaces with other application systems (e.g., the linkage between an accounts receivable system and a general ledger system). For each required software product, the following should be provided:

— Name;
— Mnemonic;
— Specification number;
— Version number;
— Source.

Copyright © 1998 IEEE. All rights reserved.

For each interface, the following should be provided:

— Discussion of the purpose of the interfacing software as related to this software product.
— Definition of the interface in terms of message content and format. It is not necessary to detail any well-documented interface, but a reference to the document defining the interface is required.

5.2.1.5 Communications interfaces

This should specify the various interfaces to communications such as local network protocols, etc.

5.2.1.6 Memory constraints

This should specify any applicable characteristics and limits on primary and secondary memory.

5.2.1.7 Operations

This should specify the normal and special operations required by the user such as

a) The various modes of operations in the user organization (e.g., user-initiated operations);
b) Periods of interactive operations and periods of unattended operations;
c) Data processing support functions;
d) Backup and recovery operations.

NOTE—This is sometimes specified as part of the User Interfaces section.

5.2.1.8 Site adaptation requirements

This should

a) Define the requirements for any data or initialization sequences that are specific to a given site, mission, or operational mode (e.g., grid values, safety limits, etc.);
b) Specify the site or mission-related features that should be modified to adapt the software to a particular installation.

5.2.2 Product functions (2.2 of the SRS)

This subsection of the SRS should provide a summary of the major functions that the software will perform. For example, an SRS for an accounting program may use this part to address customer account maintenance, customer statement, and invoice preparation without mentioning the vast amount of detail that each of those functions requires.

Sometimes the function summary that is necessary for this part can be taken directly from the section of the higher-level specification (if one exists) that allocates particular functions to the software product. Note that for the sake of clarity

a) The functions should be organized in a way that makes the list of functions understandable to the customer or to anyone else reading the document for the first time.
b) Textual or graphical methods can be used to show the different functions and their relationships. Such a diagram is not intended to show a design of a product, but simply shows the logical relationships among variables.

 Copyright © 1998 IEEE. All rights reserved.

5.2.3 User characteristics (2.3 of the SRS)

This subsection of the SRS should describe those general characteristics of the intended users of the product including educational level, experience, and technical expertise. It should not be used to state specific requirements, but rather should provide the reasons why certain specific requirements are later specified in Section 3 of the SRS.

5.2.4 Constraints (2.4 of the SRS)

This subsection of the SRS should provide a general description of any other items that will limit the developer's options. These include

a) Regulatory policies;
b) Hardware limitations (e.g., signal timing requirements);
c) Interfaces to other applications;
d) Parallel operation;
e) Audit functions;
f) Control functions;
g) Higher-order language requirements;
h) Signal handshake protocols (e.g., XON-XOFF, ACK-NACK);
i) Reliability requirements;
j) Criticality of the application;
k) Safety and security considerations.

5.2.5 Assumptions and dependencies (2.5 of the SRS)

This subsection of the SRS should list each of the factors that affect the requirements stated in the SRS. These factors are not design constraints on the software but are, rather, any changes to them that can affect the requirements in the SRS. For example, an assumption may be that a specific operating system will be available on the hardware designated for the software product. If, in fact, the operating system is not available, the SRS would then have to change accordingly.

5.2.6 Apportioning of requirements (2.6 of the SRS)

This subsection of the SRS should identify requirements that may be delayed until future versions of the system.

5.3 Specific requirements (Section 3 of the SRS)

This section of the SRS should contain all of the software requirements to a level of detail sufficient to enable designers to design a system to satisfy those requirements, and testers to test that the system satisfies those requirements. Throughout this section, every stated requirement should be externally perceivable by users, operators, or other external systems. These requirements should include at a minimum a description of every input (stimulus) into the system, every output (response) from the system, and all functions performed by the system in response to an input or in support of an output. As this is often the largest and most important part of the SRS, the following principles apply:

a) Specific requirements should be stated in conformance with all the characteristics described in 4.3.
b) Specific requirements should be cross-referenced to earlier documents that relate.
c) All requirements should be uniquely identifiable.
d) Careful attention should be given to organizing the requirements to maximize readability.

Copyright © 1998 IEEE. All rights reserved.

Before examining specific ways of organizing the requirements it is helpful to understand the various items that comprise requirements as described in 5.3.1 through 5.3.7.

5.3.1 External interfaces

This should be a detailed description of all inputs into and outputs from the software system. It should complement the interface descriptions in 5.2 and should not repeat information there.

It should include both content and format as follows:

a) Name of item;
b) Description of purpose;
c) Source of input or destination of output;
d) Valid range, accuracy, and/or tolerance;
e) Units of measure;
f) Timing;
g) Relationships to other inputs/outputs;
h) Screen formats/organization;
i) Window formats/organization;
j) Data formats;
k) Command formats;
l) End messages.

5.3.2 Functions

Functional requirements should define the fundamental actions that must take place in the software in accepting and processing the inputs and in processing and generating the outputs. These are generally listed as “shall” statements starting with “The system shall…”

These include

a) Validity checks on the inputs
b) Exact sequence of operations
c) Responses to abnormal situations, including
    1) Overflow
    2) Communication facilities
    3) Error handling and recovery
d) Effect of parameters
e) Relationship of outputs to inputs, including
    1) Input/output sequences
    2) Formulas for input to output conversion

It may be appropriate to partition the functional requirements into subfunctions or subprocesses. This does not imply that the software design will also be partitioned that way.

5.3.3 Performance requirements

This subsection should specify both the static and the dynamic numerical requirements placed on the software or on human interaction with the software as a whole. Static numerical requirements may include the following:

a) The number of terminals to be supported;
b) The number of simultaneous users to be supported;
c) Amount and type of information to be handled.

 Copyright © 1998 IEEE. All rights reserved.

Static numerical requirements are sometimes identified under a separate section entitled Capacity.

Dynamic numerical requirements may include, for example, the numbers of transactions and tasks and the amount of data to be processed within certain time periods for both normal and peak workload conditions.

All of these requirements should be stated in measurable terms.

For example,

95% of the transactions shall be processed in less than 1 s.

rather than,

An operator shall not have to wait for the transaction to complete.

NOTE—Numerical limits applied to one specific function are normally specified as part of the processing subparagraph description of that function.

5.3.4 Logical database requirements

This should specify the logical requirements for any information that is to be placed into a database. This may include the following:

a) Types of information used by various functions;
b) Frequency of use;
c) Accessing capabilities;
d) Data entities and their relationships;
e) Integrity constraints;
f) Data retention requirements.

5.3.5 Design constraints

This should specify design constraints that can be imposed by other standards, hardware limitations, etc.

5.3.5.1 Standards compliance

This subsection should specify the requirements derived from existing standards or regulations. They may include the following:

a) Report format;
b) Data naming;
c) Accounting procedures;
d) Audit tracing.

For example, this could specify the requirement for software to trace processing activity. Such traces are needed for some applications to meet minimum regulatory or financial standards. An audit trace requirement may, for example, state that all changes to a payroll database must be recorded in a trace file with before and after values.

5.3.6 Software system attributes

There are a number of attributes of software that can serve as requirements. It is important that required attributes be specified so that their achievement can be objectively verified. Subclauses 5.3.6.1 through 5.3.6.5 provide a partial list of examples.

Copyright © 1998 IEEE. All rights reserved.

5.3.6.1 Reliability

This should specify the factors required to establish the required reliability of the software system at time of delivery.

5.3.6.2 Availability

This should specify the factors required to guarantee a defined availability level for the entire system such as checkpoint, recovery, and restart.

5.3.6.3 Security

This should specify the factors that protect the software from accidental or malicious access, use, modification, destruction, or disclosure. Specific requirements in this area could include the need to

a) Utilize certain cryptographical techniques;
b) Keep specific log or history data sets;
c) Assign certain functions to different modules;
d) Restrict communications between some areas of the program;
e) Check data integrity for critical variables.

5.3.6.4 Maintainability

This should specify attributes of software that relate to the ease of maintenance of the software itself. There may be some requirement for certain modularity, interfaces, complexity, etc. Requirements should not be placed here just because they are thought to be good design practices.

5.3.6.5 Portability

This should specify attributes of software that relate to the ease of porting the software to other host machines and/or operating systems. This may include the following:

a) Percentage of components with host-dependent code;
b) Percentage of code that is host dependent;
c) Use of a proven portable language;
d) Use of a particular compiler or language subset;
e) Use of a particular operating system.

5.3.7 Organizing the specific requirements

For anything but trivial systems the detailed requirements tend to be extensive. For this reason, it is recommended that careful consideration be given to organizing these in a manner optimal for understanding. There is no one optimal organization for all systems. Different classes of systems lend themselves to different organizations of requirements in Section 3 of the SRS. Some of these organizations are described in 5.3.7.1 through 5.3.7.7.

5.3.7.1 System mode

Some systems behave quite differently depending on the mode of operation. For example, a control system may have different sets of functions depending on its mode: training, normal, or emergency. When organizing this section by mode, the outline in A.1 or A.2 should be used. The choice depends on whether interfaces and performance are dependent on mode.

 Copyright © 1998 IEEE. All rights reserved.

5.3.7.2 User class

Some systems provide different sets of functions to different classes of users. For example, an elevator control system presents different capabilities to passengers, maintenance workers, and fire fighters. When organizing this section by user class, the outline in A.3 should be used.

5.3.7.3 Objects

Objects are real-world entities that have a counterpart within the system. For example, in a patient monitoring system, objects include patients, sensors, nurses, rooms, physicians, medicines, etc. Associated with each object is a set of attributes (of that object) and functions (performed by that object). These functions are also called services, methods, or processes. When organizing this section by object, the outline in A.4 should be used. Note that sets of objects may share attributes and services. These are grouped together as classes.

5.3.7.4 Feature

A feature is an externally desired service by the system that may require a sequence of inputs to effect the desired result. For example, in a telephone system, features include local call, call forwarding, and conference call. Each feature is generally described in a sequence of stimulus-response pairs. When organizing this section by feature, the outline in A.5 should be used.

5.3.7.5 Stimulus

Some systems can be best organized by describing their functions in terms of stimuli. For example, the functions of an automatic aircraft landing system may be organized into sections for loss of power, wind shear, sudden change in roll, vertical velocity excessive, etc. When organizing this section by stimulus, the outline in A.6 should be used.

5.3.7.6 Response

Some systems can be best organized by describing all the functions in support of the generation of a response. For example, the functions of a personnel system may be organized into sections corresponding to all functions associated with generating paychecks, all functions associated with generating a current list of employees, etc. The outline in A.6 (with all occurrences of stimulus replaced with response) should be used.

5.3.7.7 Functional hierarchy

When none of the above organizational schemes prove helpful, the overall functionality can be organized into a hierarchy of functions organized by either common inputs, common outputs, or common internal data access. Data flow diagrams and data dictionaries can be used to show the relationships between and among the functions and data. When organizing this section by functional hierarchy, the outline in A.7 should be used.

5.3.8 Additional comments

Whenever a new SRS is contemplated, more than one of the organizational techniques given in 5.3.7.7 may be appropriate. In such cases, organize the specific requirements for multiple hierarchies tailored to the specific needs of the system under specification. For example, see A.8 for an organization combining user class and feature. Any additional requirements may be put in a separate section at the end of the SRS.

There are many notations, methods, and automated support tools available to aid in the documentation of requirements. For the most part, their usefulness is a function of organization. For example, when organizing by mode, finite state machines or state charts may prove helpful; when organizing by object, object-oriented

Copyright © 1998 IEEE. All rights reserved.

analysis may prove helpful; when organizing by feature, stimulus-response sequences may prove helpful; and when organizing by functional hierarchy, data flow diagrams and data dictionaries may prove helpful.

In any of the outlines given in A.1 through A.8, those sections called "Functional Requirement *i*" may be described in native language (e.g., English), in pseudocode, in a system definition language, or in four subsections titled: Introduction, Inputs, Processing, and Outputs.

5.4 Supporting information

The supporting information makes the SRS easier to use. It includes the following:

a) Table of contents;
b) Index;
c) Appendixes.

5.4.1 Table of contents and index

The table of contents and index are quite important and should follow general compositional practices.

5.4.2 Appendixes

The appendixes are not always considered part of the actual SRS and are not always necessary. They may include

a) Sample input/output formats, descriptions of cost analysis studies, or results of user surveys;
b) Supporting or background information that can help the readers of the SRS;
c) A description of the problems to be solved by the software;
d) Special packaging instructions for the code and the media to meet security, export, initial loading, or other requirements.

When appendixes are included, the SRS should explicitly state whether or not the appendixes are to be considered part of the requirements.

 Copyright © 1998 IEEE. All rights reserved.

Annex A

(informative)

SRS templates

A.1 Template of SRS Section 3 organized by mode: Version 1

3. Specific requirements
3.1 External interface requirements
  3.1.1 User interfaces
  3.1.2 Hardware interfaces
  3.1.3 Software interfaces
  3.1.4 Communications interfaces
3.2 Functional requirements
  3.2.1 Mode 1
    3.2.1.1 Functional requirement 1.1
    .
    .
    .
    3.2.1.*n* Functional requirement 1.*n*
  3.2.2 Mode 2
  .
  .
  .
  3.2.*m* Mode *m*
    3.2.*m*.1 Functional requirement *m*.1
    .
    .
    .
    3.2.*m*.*n* Functional requirement *m*.*n*
3.3 Performance requirements
3.4 Design constraints
3.5 Software system attributes
3.6 Other requirements

A.2 Template of SRS Section 3 organized by mode: Version 2

3. Specific requirements
3.1. Functional requirements
  3.1.1 Mode 1
    3.1.1.1 External interfaces
      3.1.1.1.1 User interfaces
      3.1.1.1.2 Hardware interfaces
      3.1.1.1.3 Software interfaces
      3.1.1.1.4 Communications interfaces
    3.1.1.2 Functional requirements
      3.1.1.2.1 Functional requirement 1
      .
      .
      .

Copyright © 1998 IEEE. All rights reserved.

3.1.1.2.*n* Functional requirement *n*
3.1.1.3 Performance
3.1.2 Mode 2
.
.
.
3.1.*m* Mode *m*
3.2 Design constraints
3.3 Software system attributes
3.4 Other requirements

A.3 Template of SRS Section 3 organized by user class

3. Specific requirements
3.1 External interface requirements
3.1.1 User interfaces
3.1.2 Hardware interfaces
3.1.3 Software interfaces
3.1.4 Communications interfaces
3.2 Functional requirements
3.2.1 User class 1
3.2.1.1 Functional requirement 1.1
.
.
.
3.2.1.*n* Functional requirement 1.*n*
3.2.2 User class 2
.
.
.
3.2.*m* User class *m*
3.2.*m*.1 Functional requirement *m*.1
.
.
.
3.2.*m*.*n* Functional requirement *m*.*n*
3.3 Performance requirements
3.4 Design constraints
3.5 Software system attributes
3.6 Other requirements

A.4 Template of SRS Section 3 organized by object

3. Specific requirements
3.1 External interface requirements
3.1.1 User interfaces
3.1.2 Hardware interfaces
3.1.3 Software interfaces
3.1.4 Communications interfaces
3.2 Classes/Objects
3.2.1 Class/Object 1

 Copyright © 1998 IEEE. All rights reserved.

3.2.1.1 Attributes (direct or inherited)
3.2.1.1.1 Attribute 1
.
.
.
3.2.1.1.*n* Attribute *n*
3.2.1.2 Functions (services, methods, direct or inherited)
3.2.1.2.1 Functional requirement 1.1
.
.
.
3.2.1.2.*m* Functional requirement 1.*m*
3.2.1.3 Messages (communications received or sent)
3.2.2 Class/Object 2
.
.
.
3.2.*p* Class/Object *p*
3.3 Performance requirements
3.4 Design constraints
3.5 Software system attributes
3.6 Other requirements

A.5 Template of SRS Section 3 organized by feature

3. Specific requirements
3.1 External interface requirements
3.1.1 User interfaces
3.1.2 Hardware interfaces
3.1.3 Software interfaces
3.1.4 Communications interfaces
3.2 System features
3.2.1 System Feature 1
3.2.1.1 Introduction/Purpose of feature
3.2.1.2 Stimulus/Response sequence
3.2.1.3 Associated functional requirements
3.2.1.3.1 Functional requirement 1
.
.
.
3.2.1.3.*n* Functional requirement *n*
3.2.2 System feature 2
.
.
.
3.2.*m* System feature *m*
.
.
.
3.3 Performance requirements
3.4 Design constraints
3.5 Software system attributes
3.6 Other requirements

Copyright © 1998 IEEE. All rights reserved.

A.6 Template of SRS Section 3 organized by stimulus

3. Specific requirements
3.1 External interface requirements
3.1.1 User interfaces
3.1.2 Hardware interfaces
3.1.3 Software interfaces
3.1.4 Communications interfaces
3.2 Functional requirements
3.2.1 Stimulus 1
3.2.1.1 Functional requirement 1.1
.
.
.
3.2.1.*n* Functional requirement 1.*n*
3.2.2 Stimulus 2
.
.
.
3.2.*m* Stimulus m
3.2.*m*.1 Functional requirement *m*.1
.
.
.
3.2.*m*.*n* Functional requirement *m*.*n*
3.3 Performance requirements
3.4 Design constraints
3.5 Software system attributes
3.6 Other requirements

A.7 Template of SRS Section 3 organized by functional hierarchy

3. Specific requirements
3.1 External interface requirements
3.1.1 User interfaces
3.1.2 Hardware interfaces
3.1.3 Software interfaces
3.1.4 Communications interfaces
3.2 Functional requirements
3.2.1 Information flows
3.2.1.1 Data flow diagram 1
3.2.1.1.1 Data entities
3.2.1.1.2 Pertinent processes
3.2.1.1.3 Topology
3.2.1.2 Data flow diagram 2
3.2.1.2.1 Data entities
3.2.1.2.2 Pertinent processes
3.2.1.2.3 Topology
.
.
.
3.2.1.*n* Data flow diagram *n*

 Copyright © 1998 IEEE. All rights reserved.

3.2.1.*n*.1 Data entities
3.2.1.*n*.2 Pertinent processes
3.2.1.*n*.3 Topology

3.2.2 Process descriptions

3.2.2.1 Process 1
3.2.2.1.1 Input data entities
3.2.2.1.2 Algorithm or formula of process
3.2.2.1.3 Affected data entities

3.2.2.2 Process 2
3.2.2.2.1 Input data entities
3.2.2.2.2 Algorithm or formula of process
3.2.2.2.3 Affected data entities

.
.
.

3.2.2.*m* Process *m*
3.2.2.*m*.1 Input data entities
3.2.2.*m*.2 Algorithm or formula of process
3.2.2.*m*.3 Affected data entities

3.2.3 Data construct specifications

3.2.3.1 Construct 1
3.2.3.1.1 Record type
3.2.3.1.2 Constituent fields

3.2.3.2 Construct 2
3.2.3.2.1 Record type
3.2.3.2.2 Constituent fields

.
.
.

3.2.3.*p* Construct *p*
3.2.3.*p*.1 Record type
3.2.3.*p*.2 Constituent fields

3.2.4 Data dictionary

3.2.4.1 Data element 1
3.2.4.1.1 Name
3.2.4.1.2 Representation
3.2.4.1.3 Units/Format
3.2.4.1.4 Precision/Accuracy
3.2.4.1.5 Range

3.2.4.2 Data element 2
3.2.4.2.1 Name
3.2.4.2.2 Representation
3.2.4.2.3 Units/Format
3.2.4.2.4 Precision/Accuracy
3.2.4.2.5 Range

.
.
.

3.2.4.*q* Data element *q*
3.2.4.*q*.1 Name
3.2.4.*q*.2 Representation
3.2.4.*q*.3 Units/Format
3.2.4.*q*.4 Precision/Accuracy
3.2.4.*q*.5 Range

Copyright © 1998 IEEE. All rights reserved.

3.3 Performance requirements
3.4 Design constraints
3.5 Software system attributes
3.6 Other requirements

A.8 Template of SRS Section 3 showing multiple organizations

3. Specific requirements
3.1 External interface requirements
  3.1.1 User interfaces
  3.1.2 Hardware interfaces
  3.1.3 Software interfaces
  3.1.4 Communications interfaces
3.2 Functional requirements
  3.2.1 User class 1
    3.2.1.1 Feature 1.1
      3.2.1.1.1 Introduction/Purpose of feature
      3.2.1.1.2 Stimulus/Response sequence
      3.2.1.1.3 Associated functional requirements
    3.2.1.2 Feature 1.2
      3.2.1.2.1 Introduction/Purpose of feature
      3.2.1.2.2 Stimulus/Response sequence
      3.2.1.2.3 Associated functional requirements
    .
    .
    .
    3.2.1.*m* Feature 1.*m*
      3.2.1.*m*.1 Introduction/Purpose of feature
      3.2.1.*m*.2 Stimulus/Response sequence
      3.2.1.*m*.3 Associated functional requirements
  3.2.2 User class 2
  .
  .
  .
  3.2.*n* User class *n*
  .
  .
  .
3.3 Performance requirements
3.4 Design constraints
3.5 Software system attributes
3.6 Other requirements

 Copyright © 1998 IEEE. All rights reserved.

Annex B

(informative)

Guidelines for compliance with IEEE/EIA 12207.1-1997

B.1 Overview

The Software Engineering Standards Committee (SESC) of the IEEE Computer Society has endorsed the policy of adopting international standards. In 1995, the international standard, ISO/IEC 12207, Information technology—Software life cycle processes, was completed. The standard establishes a common framework for software life cycle processes, with well-defined terminology, that can be referenced by the software industry.

In 1995 the SESC evaluated ISO/IEC 12207 and decided that the standard should be adopted and serve as the basis for life cycle processes within the IEEE Software Engineering Collection. The IEEE adaptation of ISO/IEC 12207 is IEEE/EIA 12207.0-1996. It contains ISO/IEC 12207 and the following additions: improved compliance approach, life cycle process objectives, life cycle data objectives, and errata.

The implementation of ISO/IEC 12207 within the IEEE also includes the following:

— IEEE/EIA 12207.1-1997, IEEE/EIA Guide for Information Technology—Software life cycle processes—Life cycle data;

— IEEE/EIA 12207.2-1997, IEEE/EIA Guide for Information Technology—Software life cycle processes—Implementation considerations; and

— Additions to 11 SESC standards (i.e., IEEE Stds 730, 828, 829, 830, 1012, 1016, 1058, 1062, 1219, 1233, 1362) to define the correlation between the data produced by existing SESC standards and the data produced by the application of IEEE/EIA 12207.1-1997.

NOTE—Although IEEE/EIA 12207.1-1997 is a guide, it also contains provisions for application as a standard with specific compliance requirements. This annex treats 12207.1-1997 as a standard.

B.1.1 Scope and purpose

Both IEEE Std 830-1998 and IEEE/EIA 12207.1-1997 place requirements on a Software Requirements Description Document. The purpose of this annex is to explain the relationship between the two sets of requirements so that users producing documents intended to comply with both standards may do so.

B.2 Correlation

This clause explains the relationship between IEEE Std 830-1998 and IEEE/EIA 12207.0-1996 and IEEE/EIA 12207.1-1997 in the following areas: terminology, process, and life cycle data.

B.2.1 Terminology correlation

Both this recommended practice and IEEE/EIA 12207.0-1996 have similar semantics for the key terms of software, requirements, specification, supplier, developer, and maintainer. This recommended practice uses

Copyright © 1998 IEEE. All rights reserved.

the term "customer" where IEEE/EIA 12207.0-1996 uses "acquirer," and this recommended practice uses "user" where IEEE/EIA 12207.0-1996 uses "operator."

B.2.2 Process correlation

IEEE/EIA 12207.0-1996 uses a process-oriented approach for describing the definition of a set of requirements for software. This recommended practice uses a product-oriented approach, where the product is a Software Requirements Description (SRD). There are natural process steps, namely the steps to create each portion of the SRD. These may be correlated with the process requirements of IEEE/EIA 12207.0-1996. The difference is that this recommended practice is focused on the development of software requirements whereas IEEE/EIA 12207.0-1996 provides an overall life cycle view and mentions Software Requirements Analysis as part of its Development Process. This recommended practice provides a greater level of detail on what is involved in the preparation of an SRD.

B.2.3 Life cycle data correlation

IEEE/EIA 12207.0-1996 takes the viewpoint that the software requirements are derived from the system requirements. Therefore, it uses the term, "description" rather that "specification" to describe the software requirements. In a system in which software is a component, each requiring its own specification, there would be a System Requirements Specification (SRS) and one or more SRDs. If the term Software Requirements Specification had been used, there would be a confusion between an SRS referring to the system or software requirements. In the case where there is a stand-alone software system, IEEE/EIA 12207.1-1997 states "If the software is a stand-alone system, then this document should be a specification."

B.3 Content mapping

This clause provides details bearing on a claim that an SRS complying with this recommended practice would also achieve "document compliance" with the SRD described in IEEE/EIA 12207.1-1997. The requirements for document compliance are summarized in a single row of Table 1 of IEEE/EIA 12207.1-1997. That row is reproduced in Table B.1 of this recommended practice.

Table B.1—Summary of requirements for an SRD excerpted from Table 1 of IEEE/EIA 12207.1-1997

Information item	IEEE/EIA 12207.0-1996 Clause	Kind	IEEE/EIA 12207.1-1997 Clause	References
Software Requirements Description	5.1.1.4, 5.3.4.1, 5.3.4.2	Description (See note for 6.22.1 of IEEE/EIA 12207.1-1997.)	6.22	IEEE Std 830-1998; EIA/IEEE J-STD-016, F.2.3, F.2.4; MIL-STD 961D. Also see ISO/IEC 5806, 5807, 6593, 8631, 8790, and 11411 for guidance on use of notations.

The requirements for document compliance are discussed in the following subclauses:

— B.3.1 discusses compliance with the information requirements noted in column 2 of Table B.1 as prescribed by 5.1.1.4, 5.3.4.1, and 5.3.4.2 of IEEE/EIA 12207.0-1996.

 Copyright © 1998 IEEE. All rights reserved.

— B.3.2 discusses compliance with the generic content guideline (the "kind" of document) noted in column 3 of Table B.1 as a "description". The generic content guidelines for a "description" appear in 5.1 of IEEE/EIA 12207.1-1997.

— B.3.3 discusses compliance with the specific requirements for a Software Requirements Description noted in column 4 of Table B.1 as prescribed by 6.22 of IEEE/EIA 12207.1-1997.

— B.3.4 discusses compliance with the life cycle data objectives of Annex H of IEEE/EIA 12207.0-1996 as described in 4.2 of IEEE/EIA 12207.1-1997.

B.3.1 Compliance with information requirements of IEEE/EIA 12207.0-1996

The information requirements for an SRD are those prescribed by 5.1.1.4, 5.3.4.1, and 5.3.4.2 of IEEE/EIA 12207.0-1996. The requirements are substantively identical to those considered in B.3.3 of this recommended practice.

B.3.2 Compliance with generic content guidelines of IEEE/EIA 12207.1-1997

According to IEEE/EIA 12207.1-1997, the generic content guideline for an SRD is generally a description, as prescribed by 5.1 of IEEE/EIA 12207.1-1997. A complying description shall achieve the purpose stated in 5.1.1 and include the information listed in 5.1.2 of IEEE/EIA 12207.1-1997.

The purpose of a description is:

> IEEE/EIA 12207.1-1997, subclause 5.1.1: Purpose: Describe a planned or actual function, design, performance, or process.

An SRD complying with this recommended practice would achieve the stated purpose.

Any description or specification complying with IEEE/EIA 12207.1-1997 shall satisfy the generic content requirements provided in 5.1.2 of that standard. Table B.2 of this recommended practice lists the generic content items and, where appropriate, references the clause of this recommended practice that requires the same information.

Table B.2—Coverage of generic description requirements by IEEE Std 830-1998

IEEE/EIA 12207.1-1997 generic content	Corresponding clauses of IEEE Std 830-1998	Additions to requirements of IEEE Std 830-1998
a) Date of issue and status	—	Date of issue and status shall be provided.
b) Scope	5.1.1 Scope	—
c) Issuing organization	—	Issuing organization shall be identified.
d) References	5.1.4 References	—
e) Context	5.1.2 Scope	—
f) Notation for description	4.3 Characteristics of a good SRS	—
g) Body	5. The parts of an SRS	—
h) Summary	5.1.1. Overview	—
i) Glossary	5.1.3 Definitions	—
j) Change history	—	Change history for the SRD shall be provided or referenced.

Copyright © 1998 IEEE. All rights reserved.

B.3.3 Compliance with specific content requirements of IEEE/EIA 12207.1-1997

The specific content requirements for an SRD in IEEE/EIA 12207.1-1997 are prescribed by 6.22 of IEEE/EIA 12207.1-1997. A compliant SRD shall achieve the purpose stated in 6.22.1 of IEEE/EIA 12207.1-1997.

The purpose of the SRD is:

> IEEE/EIA 12207.1-1997, subclause 6.22.1: Purpose: Specify the requirements for a software item and the methods to be used to ensure that each requirement has been met. Used as the basis for design and qualification testing of a software item.

An SRS complying with this recommended practice and meeting the additional requirements of Table B.3 of this recommended practice would achieve the stated purpose.

An SRD compliant with IEEE/EIA 12207.1-1997 shall satisfy the specific content requirements provided in 6.22.3 and 6.22.4 of that standard. Table B.3 of this recommended practice lists the specific content items and, where appropriate, references the clause of this recommended practice that requires the same information.

An SRD specified according the requirements stated or referenced in Table B.3 of this recommended practice shall be evaluated considering the criteria provided in 5.3.4.2 of IEEE/EIA 12207.0-1996.

Table B.3—Coverage of specific SRD requirements by IEEE Std 830-1998

IEEE/EIA 12207.1-1997 specific content	Corresponding clauses of IEEE Std 830-1998	Additions to requirements of IEEE Std 830-1998
a) Generic description information	See Table B.2	—
b) System identification and overview	5.1.1 Scope	—
c) Functionality of the software item including: – Performance requirements – Physical characteristics – Environmental conditions	5.3.2 Functions 5.3.3 Performance requirements	Physical characteristics and environmental conditions should be provided.
d) Requirements for interfaces external to software item	5.3.1 External interfaces	—
e) Qualification requirements	—	The requirements to be used for qualification testing should be provided (or referenced).
f) Safety specifications	5.2.4 Constraints	—
g) Security and privacy specifications	5.3.6.3 Security	—
h) Human-factors engineering requirements	5.2.3 User characteristics 5.2.1.2 User interfaces	—
i) Data definition and database requirements	5.3.4 Logical data base requirements	—
j) Installation and acceptance requirements at operation site	5.2.1.8 Site adaptation requirements	Installation and acceptance requirements at operation site
k) Installation and acceptance requirements at maintenance site	—	Installation and acceptance requirements at maintenance site
l) User documentation requirements	—	User documentation requirements
m) User operation and execution requirements	5.2.1.7 Operations	User execution requirements

 Copyright © 1998 IEEE. All rights reserved.

Table B.3—Coverage of specific SRD requirements by IEEE Std 830-1998 (continued)

IEEE/EIA 12207.1-1997 specific content	Corresponding clauses of IEEE Std 830-1998	Additions to requirements of IEEE Std 830-1998
n) User maintenance requirements	5.3.6.4 Maintainability	—
o) Software quality characteristics	5.3.6 Software system attributes	—
p) Design and implementation constraints	5.2.4 Constraints	—
q) Computer resource requirements	5.3.3 Performance requirements	Computer resource requirements
r) Packaging requirements	—	Packaging requirements
s) Precedence and criticality of requirements	5.2.6 Apportioning of requirements	—
t) Requirements traceability	4.3.8 Traceable	—
u) Rationale	5.2.5 Assumptions and dependencies	—
Items a) through f) below are from 6.22.4 a) Support the life cycle data objectives of Annex H of IEEE/EIA 12207.0-1996	—	Support the life cycle data objectives of Annex H of IEEE/EIA 12207.0-1996
b) Describe any function using well-defined notation	4.3 Characteristics of a good SRS	—
c) Define no requirements that are in conflict	4.3 Characteristics of a good SRS	—
d) User standard terminology and definitions	5.1.3 Definition	—
e) Define each unique requirement one to prevent inconsistency	4.3 Characteristics of a good SRS	—
f) Uniquely identify each requirement	4.3 Characteristics of a good SRS	—

B.3.4 Compliance with life cycle data objectives

In addition to the content requirements, life cycle data shall be managed in accordance with the objectives provided in Annex H of IEEE/EIA 12207.0-1996.

B.4 Conclusion

The analysis suggests that any SRS complying with this recommended practice and the additions shown in Table B.2 and Table B.3 also complies with the requirements of an SRD in IEEE/EIA 12207.1-1997. In addition, to comply with IEEE/EIA 12207.1-1997, an SRS shall support the life cycle data objectives of Annex H of IEEE/EIA 12207.0-1996.

Copyright © 1998 IEEE. All rights reserved.

On Non-Functional Requirements

Martin Glinz
Department of Informatics, University of Zurich, Switzerland
glinz@ifi.uzh.ch

Abstract

Although the term 'non-functional requirement' has been in use for more than 20 years, there is still no consensus in the requirements engineering community what non-functional requirements are and how we should elicit, document, and validate them. On the other hand, there is a unanimous consensus that non-functional requirements are important and can be critical for the success of a project.

This paper surveys the existing definitions of the term, highlights and discusses the problems with the current definitions, and contributes concepts for overcoming these problems.

1. Introduction

If you want to trigger a hot debate among a group of requirements engineering people, just let them talk about non-functional requirements. Although this term has been in use for more than two decades, there is still no consensus about the nature of non-functional requirements and how to document them in requirements specifications.

This paper is an attempt to work out and discuss the problems that we have with the notion of non-functional requirements and to contribute concepts for overcoming these problems. The focus is on system (or product) requirements; the role of non-functional requirements in the software process is not discussed [16].

The paper is organized as follows. Section 2 surveys typical definitions for the terms 'functional requirement' and 'non-functional requirement'. The problems with these definitions are discussed in Section 3. Section 4 presents concepts about how these problems can be overcome or at least alleviated. The paper ends with a discussion of these concepts.

2. Defining the term

In every current requirements classification (for example [7], [11], [12]), we find a distinction between requirements concerning the functionality of a system and other requirements.

There is a rather broad consensus about how to define the term *'functional requirements'*. The existing definitions follow two threads that coincide to a large extent. In the first thread, the emphasis is on functions: a functional requirement specifies "a function that a system (...) must be able to perform" [6], "what the product must do" [18], "what the system should do" [20]. The second thread emphasizes behavior: functional requirements "describe the behavioral aspects of a system" [1]; behavioral requirements are "those requirements that specify the inputs (stimuli) to the system, the outputs (responses) from the system, and behavioral relationships between them; also called functional or operational requirements." [3].

Wiegers as well as Jacobson, Rumbaugh and Booch try a synthesis: "A statement of a piece of required functionality or a behavior that a system will exhibit under specific conditions." [21]; "A requirement that specifies an action that a system must be able to perform, without considering physical constraints; a requirement that specifies input/output behavior of a system." [10].

There is only one semantic difference that may arise between the different definitions: timing requirements may be viewed as behavioral, while they are not functional. However, most publications in RE consider timing requirements to be performance requirements which in turn are classified as non-functional requirements.

On the other hand, there is no such consensus for *non-functional requirements*. Table 1 gives an overview of selected definitions from the literature or the web, which – in my opinion – are representative of the definitions that exist.

3. Where is the problem?

The problems that we currently have with the notion of non-functional requirements can be divided into *definition problems*, *classification problems* and *representation problems*.

1090-705X/07 $25.00 © 2007 IEEE
DOI 10.1109/RE.2007.45

IEEE computer society

Authorized licensed use limited to: University of Tartu IEL Trial. Downloaded on March 30,2010 at 04:00:21 EDT from IEEE Xplore. Restrictions apply.

Table 1. Definitions of the term *'non-functional requirement(s)'* (listed in alphabetical order of sources)

Source	Definition
Antón [1]	Describe the nonbehavioral aspects of a system, capturing the properties and constraints under which a system must operate.
Davis [3]	The required overall attributes of the system, including portability, reliability, efficiency, human engineering, testability, understandability, and modifiability.
IEEE 610.12 [6]	Term is not defined. The standard distinguishes design requirements, implementation requirements, interface requirements, performance requirements, and physical requirements.
IEEE 830-1998 [7]	Term is not defined. The standard defines the categories functionality, external interfaces, performance, attributes (portability, security, etc.), and design constraints. Project requirements (such as schedule, cost, or development requirements) are explicitly excluded.
Jacobson, Booch and Rumbaugh [10]	A requirement that specifies system properties, such as environmental and implementation constraints, performance, platform dependencies, maintainability, extensibility, and reliability. A requirement that specifies physical constraints on a functional requirement.
Kotonya and Sommerville [11]	Requirements which are not specifically concerned with the functionality of a system. They place restrictions on the product being developed and the development process, and they specify external constraints that the product must meet.
Mylopoulos, Chung and Nixon [16]	"... global requirements on its development or operational cost, performance, reliability, maintainability, portability, robustness, and the like. (...) There is not a formal definition or a complete list of nonfunctional requirements."
Ncube [17]	The behavioral properties that the specified functions must have, such as performance, usability.
Robertson and Robertson [18]	A property, or quality, that the product must have, such as an appearance, or a speed or accuracy property.
SCREEN Glossary [19]	A requirement on a service that does not have a bearing on its functionality, but describes attributes, constraints, performance considerations, design, quality of service, environmental considerations, failure and recovery.
Wiegers [21]	A description of a property or characteristic that a software system must exhibit or a constraint that it must respect, other than an observable system behavior.
Wikipedia: Non-Functional Requirements [22]	Requirements which specify criteria that can be used to judge the operation of a system, rather than specific behaviors.
Wikipedia: Requirements Analysis [23]	Requirements which impose constraints on the design or implementation (such as performance requirements, quality standards, or design constraints).

3.1. Definition problems

When analyzing the definitions in Table 1, we find not only terminological, but also major conceptual discrepancies. Basically, all definitions build on the following terms: *property* or *characteristic, attribute*, *quality*, *constraint*, and *performance*. However, there is no consensus about the concepts that these terms denote. There are also cases where the meaning is not clear, because terms are used without a definition or a clarifying example.

Property and *characteristic* seem to be used in their general meaning, i.e. they denote something that the system must have, which typically includes specific qualities such as usability or reliability, but excludes any functional quality. There is no consensus whether constraints also are properties: Jacobson, Booch and Rumbaugh [10] include them, others, e.g. Wiegers [21] and Antón [1] exclude them.

Attribute is a term that is used both with a broad and a narrow meaning. In IEEE 830-1998 [7], attributes are a collection of specific qualities, excluding performance and constraints. On the other hand, in the definition by Davis [3], every non-functional requirement is an attribute of the system.

Every requirement (including all functional ones) can be regarded as a *quality*, because, according to ISO 9000:2000 [8], quality is the "degree to which a set of inherent characteristics fulfils requirements". Similarly, every requirement can be regarded as a *constraint*, because it constrains the space of potential solutions to those that meet this requirement.

Hence, in all definitions that mention the term *quality*, its meaning is restricted to a set of specific qualities other than functionality: usability, reliability, security, etc.

Correspondingly, it is clear that *constraint* in the context of non-functional requirements must have a restricted meaning. However, there is no consensus among the existing definitions what precisely this restriction should be. For example, IEEE 830-1998 [7] or Jacobson, Booch and Rumbaugh [10] restrict the meaning of constraint to design constraints and physi-

Authorized licensed use limited to: University of Tartu IEL Trial. Downloaded on March 30,2010 at 04:00:21 EDT from IEEE Xplore. Restrictions apply.

cal constraints. Others, for example the definition in the Wikipedia article on requirements analysis [23], do not treat constraints as a sub-category of non-functional requirements, but consider every non-functional requirement to be a constraint. Davis [3] does not mention constraints at all when discussing non-functional requirements. Robertson and Robertson [18] have a rather specific view of constraints: they consider operational, cultural, and legal constraints to be non-functional properties, whereas design constraints are regarded as a concept that is different from both functional and non-functional requirements.

Performance is treated as a quality or attribute in many definitions. Others, e.g. IEEE 830-1998 [7] and Wikipedia [23] consider it as a separate category.

Another discrepancy exists in the *scope* of non-functional requirements. Some definitions emphasize that non-functional requirements have, by definition, a *global scope*: "global requirements" (Mylopoulos, Chung and Nixon [16]), "overall attributes" (Davis [3]). Accordingly, proponents of this global view separate functional and non-functional requirements completely. For example, in the Volere template [18] they are documented in two separate top-level sections. On the other hand, Jacobson, Booch and Rumbaugh [10] emphasize that there are both local non-functional requirements (e.g. most performance requirements) and global requirements such as security or reliability.

Most definitions refer to *system requirements* (also called *product requirements*) only and exclude *project* and *process requirements* either explicitly [7] or implicitly. However, in the definition by Kotonya and Sommerville [11], project requirements are considered to be non-functional requirements.

Finally, the variety and divergence of concepts used in the definitions of non-functional requirements also lead to definitions containing elements that are obviously misconceived. For example, the SCREEN glossary [19] lists failure and recovery as non-functional properties, while behavior in the case of failure and recovery from failures are clearly functional issues. Ncube [17] even defines non-functional requirements as "the behavioral properties...". Although this is more likely a severe typo than a misconception, the fact that this error went unnoticed in a PhD thesis illustrates the fuzziness of the current notions of non-functional requirements and the need for a clear and concise definition. The worst example is the definition in the Wikipedia article on non-functional requirements [22], which is so vague that it could mean almost anything.

3.2. Classification problems

When analyzing the definitions given in Table 1, we also find rather divergent concepts for sub-classifying non-functional requirements. Davis [3] regards them as qualities and uses Boehm's quality tree [2] as a sub-classification for non-functional requirements. The IEEE standard 830-1998 on Software Requirements Specifications [7] sub-classifies non-functional requirements into external interface requirements, performance requirements, attributes and design constraints, where the attributes are a set of qualities such as reliability, availability, security, etc. The IEEE Standard Glossary of Software Engineering Terminology [6] distinguishes functional requirements on the one hand and design requirements, implementation requirements, interface requirements, performance requirements, and physical requirements on the other. Sommerville [20] uses a sub-classification into product requirements, organizational requirements and external requirements.

More classification problems arise due to mixing three concepts that should better be separated. These are the concepts of *kind* (should a given requirement be regarded as a function, a quality, a constraint, etc.), *representation* (see below), and *satisfaction* (hard vs. soft requirements). For an in-depth discussion of this problem, see [5].

3.3. Representation problems

As long as we regard any requirement that describes a function or behavior as a functional requirement, the notion of non-functional requirements is *representation-dependent*. Consider the following example: A particular security requirement could be expressed as "The system shall prevent any unauthorized access to the customer data", which, according to all definitions given in Table 1 is a non-functional requirement. If we represent this requirement in a more concrete form, for example as "The probability for successful access to the customer data by an unauthorized person shall be smaller than 10^{-5}", this is still a non-functional requirement. However, if we refine the original requirement to "The database shall grant access to the customer data only to those users that have been authorized by their user name and password", we have a functional requirement, albeit it is still a security requirement. In a nutshell, the kind of a requirement depends on the way we represent it.

A second representational problem is the lack of consensus *where to document* non-functional requirements. As discussed above, some authors recommend the documentation of functional and non-functional requirements in separate chapters of the software requirements specification. The Volere template [18] is a prominent example for this documentation style. In IEEE 830-1998 [7], seven of the eight proposed SRS templates also separate the functional requirements

Authorized licensed use limited to: University of Tartu IEL Trial. Downloaded on March 30,2010 at 04:00:21 EDT from IEEE Xplore. Restrictions apply.

completely from the non-functional ones. On the other hand, when documenting requirements in a use-case-oriented style according to the Rational Unified Process [10], non-functional requirements are attached to use case as far as possible. Only the remaining (global) non-functional requirements are documented separately as so-called supplementary requirements.

However, there are also cases where a non-functional requirement affects neither a single functional requirement nor the system as a whole. Instead, it pertains to a specific set of functional requirements. For example, some subset of the set of all use cases may need secure communication, while the other use cases do not. Such a case cannot be documented adequately with classic requirements specification templates and must be handled by setting explicit traceability links. As we will see in the next chapter, aspect-orientation provides a solution for this problem.

4. Elements of a solution

4.1. A faceted classification

In [5], I have proposed that the classification problems, and – in a radical sense – also the definition problems be overcome by introducing a faceted classification for requirements (Fig. 1) where the terms 'functional requirement' and 'non-functional requirement' no longer appear.

Separating the concepts of *kind*, *representation*, *satisfaction*, and *role* has several advantages, for example: (i) It is the representation of a requirement (not its kind!) that determines the way in which we can verify that the system satisfies a requirement (Table 2). (ii) Decoupling the kind and satisfaction facets reflects the fact that functional requirements are not always hard and non-functional requirements not always soft. (iii) The role facet allows a clear distinction of (prescriptive) system requirements and (normative or assumptive) domain requirements.

Figure 1. A faceted classification of requirements (from [5])

However, when using this classification it turned out that the idea of getting rid of the terms 'functional requirement' and 'non-functional requirement' is too radical. When practicing requirements engineering, *there is a need* to distinguish functional concerns from other, "non-functional" concerns and there is also a need for a sub-classification of the "non-functional" concerns in a clear and comprehensible way. In the next section, such a definition is presented.

Table 2. Representation determines verification [5]

Representation	Type of verification
Operational	Review, test or formal verification
Quantitative	Measurement (at least on an ordinal scale)
Qualitative	No direct verification. May be done by subjective stakeholder judgment of deployed system, by prototypes or indirectly by goal refinement or derived metrics
Declarative	Review

4.2. A definition based on concerns

We define a taxonomy of terms that is based on the concept of *concerns*, which makes it independent of the chosen representation. We assume a requirements engineering context, where *system* is the entity whose requirements have to be specified.

Furthermore, the taxonomy concentrates on *system requirements*. As project and process requirements are conceptually different from system requirements, they should be distinguished at the root level and not in a sub-category such as non-functional requirements.

DEFINITION. A *concern* is a matter of interest in a system. A concern is a *functional* or *behavioral* concern if its matter of interest is primarily the expected behavior of a system or system component in terms of its reaction to given input stimuli and the functions and data required for processing the stimuli and producing the reaction. A concern is a *performance* concern if its matter of interest is timing, speed, volume or throughput. A concern is a *quality concern* if its matter of interest is a quality of the kind enumerated in ISO/IEC 9126 [9].

DEFINITION. The set of all requirements of a system is partitioned into *functional requirements*, *performance requirements*, *specific quality requirements*, and *constraints*.

A *functional requirement* is a requirement that pertains to a functional concern.

A *performance requirement* is a requirement that pertains to a performance concern.

A *specific quality requirement* is a requirement that pertains to a quality concern other than the quality of meeting the functional requirements.

A *constraint* is a requirement that constrains the solution space beyond what is necessary for meeting

Authorized licensed use limited to: University of Tartu IEL Trial. Downloaded on March 30,2010 at 04:00:21 EDT from IEEE Xplore. Restrictions apply.

the given functional, performance, and specific quality requirements.

An *attribute* is a performance requirement or a specific quality requirement.

The taxonomy defined above is visualized in Figure 2. In my opinion, this structure is sufficient and useful both for theoretical and practical work. For persons who do not want to dispose of the term 'non-functional requirement', we can define this term additionally as follows.

DEFINITION. A *non-functional requirement* is an attribute of or a constraint on a system.

Figure 2. A concern-based taxonomy of requirements

Performance is a category of its own in our taxonomy (as well as in the IEEE standards [6] [7]) because performance requirements are typically treated separately in practice. This is probably due to the fact that measuring performance is not difficult a priori, while measuring other attributes is: there is a broad consensus to measure performance in terms of time, volume, and volume per unit of time. For any other attribute, there are no such generally agreed measures, which means that the task of eliciting specific qualities always implies finding an agreement among the stakeholders how to measure these qualities.

Interfaces, which are a separate category in the IEEE standards [6] [7], no longer appear in the terminology defined above. An interface requirement is classified according to the concern it pertains to as functional, performance, specific quality, or constraint.

Due to the systematic construction of our taxonomy, the task of *classifying* a given requirement becomes easier and less ambiguous. The often heard rule 'What the system does → functional requirement; How the system behaves → non-functional requirement' [4] is too coarse and leads to mis-classifications when attributes and constraints are represented functionally or when they are very important[1].

Table 3 gives the classification rules for the taxonomy laid out in Figure 2.

Table 3. Classification rules

No[1]	**Question** Was this requirement stated because we need to specify...	**Result**
1	... some of the system's behavior, data, input, or reaction to input stimuli – regardless of the way how this is done?	Functional
2	... restrictions about timing, processing or reaction speed, data volume, or throughput?	Performance
3	... a specific quality that the system or a component shall have?	Specific quality
4	... any other restriction about what the system shall do, how it shall do it, or any prescribed solution or solution element?	Constraint

[1] Questions must be applied in this order

4.3. Aspect-oriented representation

As soon as we structure the functional requirements specification systematically (by using a text template or by modeling requirements), the question about structuring the non-functional requirements comes up. Structuring attributes and constraints into sub-kinds according to a documentation template is helpful here, but clearly this is not enough; in particular when we have attributes and constraints that are neither completely local nor fully global, but pertain to some specific parts of the system.

An aspect-oriented representation of requirements, in particular of attributes and constraints, helps overcome this problem. A multi-dimensional separation of concerns [15] allows every concern to be modeled separately. Thus, all concerns are treated equally, which looks clean and elegant from an academic viewpoint. However, in practice, there is almost always a dominant concern, which is typically a functional one. This concern is crosscut by other concerns, both functional and non-functional ones.

In my research group, we have developed an aspect-oriented-extension for a hierarchical modeling lan-

[1] For example, in particle physics, the detectors of contemporary accelerators produce enormous amounts of data in real time. When asked a 'what shall the system do' question about the data processing software for such a detector, one of the first answers a physicist typically would give would be that the system must be able to cope with the data volume. However, this is a performance requirement.

Authorized licensed use limited to: University of Tartu IEL Trial. Downloaded on March 30,2010 at 04:00:21 EDT from IEEE Xplore. Restrictions apply.

guage [13], [14] where we identify a dominant functional concern and decompose the system model hierarchically according to the structure of this concern. All other concerns are modeled as aspects of this primary model. Aspect composition is supported by formal model weaving semantics.

Thus we can document attributes and constraints as separate entities, but, at the same time, attach them systematically to those elements in the primary model hierarchy where they apply. Global attributes and constraints are attached to the root of the decomposition hierarchy, while an attribute or constraint that restricts only some parts of the model is attached exactly to these parts by modeling join relationship from the aspect to the affected parts of the primary model.

5. Discussion

The analysis of the definitions given in Table 1 reveals the deficiencies of the current terminology. The *new definitions* proposed in this paper

- are more systematically constructed than any existing definitions that I am aware of,
- are representation-independent; i.e. the kind of a requirement is always the same, regardless of its representation,
- make it easier to classify a given requirement: with the classification criteria given in Table 3, the classification is much less ambiguous than with traditional definitions,
- better support the evolution of a requirements specification, because the classification of a requirement remains invariant under refinement and change as long as the concern to which the requirements pertains remains the same.

With an aspect-oriented documentation of attributes and constraints, the *documentation* and *traceability* problems of non-functional requirements are alleviated.

As this is mainly theoretical work, the *validation* of the theoretical soundness and usefulness of these ideas will be the extent to which other researchers find these ideas useful and adopt or build upon them.

The systematic exploration of the *practical usefulness* is a topic for further investigation.

References

[1] A. Antón (1997). *Goal Identification and Refinement in the Specification of Information Systems*. PhD Thesis, Georgia Institute of Technology.

[2] B. Boehm et al. (1976). Quantitative Evaluation of Software Quality. *Proc. 2nd IEEE International Conference on Software Engineering*. 592-605.

[3] A. Davis (1993). *Software Requirements: Objects, Functions and States*. Prentice Hall.

[4] X. Franch (1998). Systematic Formulation of Non-Functional Characteristics of Software. *Proc. 3rd Int'l Conf. Requirements Engineering (ICRE'98)*. 174-181.

[5] M. Glinz (2005). Rethinking the Notion of Non-Functional Requirements. *Proc. Third World Congress for Software Quality (3WCSQ 2005)*, Munich, Germany, Vol. II. 55-64.

[6] IEEE (1990). *Standard Glossary of Software Engineering Terminology*. IEEE Standard 610.12-1990.

[7] IEEE (1998). *IEEE Recommended Practice for Software Requirements Specifications*. IEEE Std. 830-1998.

[8] ISO 9000 (2000). *Quality Management Systems – Fundamentals and Vocabulary*. International Organization for Standardization.

[9] ISO/IEC 9126-1 (2001). *Software Engineering – Product Quality – Part 1: Quality Model*. International Organization for Standardization.

[10] I. Jacobson, G. Booch, and J. Rumbaugh (1999). *The Unified Software Development Process*. Reading, Mass.: Addison Wesley.

[11] G. Kotonya, I. Sommerville (1998). *Requirements Engineering: Processes and Techniques*. John Wiley & Sons.

[12] A. van Lamsweerde (2001). Goal-Oriented Requirements Engineering: A Guided Tour. *Proc. 5th International Symposium on Requirements Engineering (RE'01)*, Toronto. 249-261.

[13] S. Meier, T. Reinhard, C. Seybold, and M. Glinz (2006). Aspect-Oriented Modeling with Integrated Object Models. *Proc. Modellierung 2006*, Innsbruck, Austria. 129-144.

[14] S. Meier, T. Reinhard, R. Stoiber, M. Glinz (2007). Modeling and Evolving Crosscutting Concerns in ADORA. *Proc. Early Aspects at ICSE: Workshop in Aspect-Oriented Requirements Engineering and Architecture Design*.

[15] A. Moreira, A. Rashid, and J. Araújo (2005). Multi-Dimensional Separation of Concerns in Requirements Engineering. *Proc. 13th IEEE International Requirements Engineering Conference (RE'05)*. 285-296.

[16] J. Mylopoulos, L. Chung, B. Nixon (1992). Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. *IEEE Transactions on Software Engineering* **18**, 6 (June 1992). 483-497.

[17] C. Ncube (2000). *A Requirements Engineering Method for COTS-Based Systems Development*. PhD Thesis, City University London.

[18] S. Robertson and J. Robertson (1999). *Mastering the Requirements Process*. ACM Press.

[19] SCREEN (1999). *Glossary of EU SCREEN Project*. http://cordis.europa.eu/infowin/acts/rus/projects/screen/glossary/glossary.htm (visited 2007-07-05)

[20] I. Sommerville (2004). *Software Engineering*, Seventh Edition. Pearson Education.

[21] K. Wiegers (2003). *Software Requirements*, 2nd edition. Microsoft Press.

[22] Wikipedia: *Non-Functional Requirements* http://en.wikipedia.org/wiki/Non-functional_requirements (visited 2007-07-05)

[23] Wikipedia: *Requirements Analysis* http://en.wikipedia.org/wiki/Requirements_analysis (visited 2007-07-05)

Authorized licensed use limited to: University of Tartu IEL Trial. Downloaded on March 30,2010 at 04:00:21 EDT from IEEE Xplore. Restrictions apply.

Using statement-level templates to improve the quality of requirements

An Integrate white paper

October 2012

integrate
systems engineering

© 2012 integrate systems engineering ltd

Using statement-level templates to improve the quality of requirements

Abstract

Much has been written on the subject of what constitutes an acceptable requirement statement, and tools are available to check the quality of a requirement in terms of grammar and vocabulary after it has been written. Using a statement-level template (or "boilerplate") is a means of creating an acceptable requirement as it is written, rather than after the event.

There have been a number of papers published on using templates and structured language for improving requirements quality, and for relating textual requirements to formal specifications of various kinds. This paper focuses on the use of statement-level templates to aid in the expression of textual requirements in general, including stakeholder requirements at levels of abstraction above the specification. The aim of the approach is to provide grammatical structures that encapsulate the accepted rules for writing singular, unambiguous, verifiable requirements.

The paper surveys a number of tools for supporting statement-level templates, and describes some experiences of applying the use of templates to projects.

Introduction

It is received wisdom in the discipline of requirements management that textual requirements should possess a number of characteristics, such as singularity, non-ambiguity and verifiability. A synthesis of such properties can be found in [1] and in more detail in [12]. A number of tools are available, such as LEXIOR [2], and RQA (Requirements Quality Analyzer) [3], that analyze textual requirements, and identify where they fail to possess such characteristics.

This paper reviews an approach to assuring that such characteristics are present in the requirements as they are written, rather than assessing them after the event. The approach encourages the author to select a pre-determined textual template [18] specific to the type of requirement to be expressed, and to fill out place-holders to instantiate it.

There is a body of published work in the area of requirements templates (or, more precisely, patterns) associated with formal specifications, which work is briefly reviewed below. The main motivation of such work is to make the specification of a system more approachable through the use of structured text that has a precise relationship with one or more underlying formal models.

The approached described in this paper in slightly different. We do not attempt to tie the templates to any underlying formal model, and argue that there are still significant benefits to be gained, as enumerated below. What's more, the approach can address a very broad range of kinds of requirement at all levels of abstraction, including high-level stakeholder goals, aspirations and requirements.

Early work on the use of this kind of requirement template dates from 1998. Although not formally published, the basic approach can be found in [18] and at [4]. Each template is an English language sentence with place-holders for specific words that define the particular requirement. No constraints are placed on the kinds of words that can fill the place-holders; only encouragement to use consistent vocabulary.

More recent work [15], [19] has applied the approach in a very specific domain and attempted to quantify the benefits. Later development makes use of semantic parsing with underlying domain-specific ontologies to guide the author in the selection of appropriate terms, units and values.

The benefits of using this approach to author requirements include:

- *An aid in articulation*. Being able to draw upon a palette of classified templates helps in finding effective ways of expressing particular kinds of requirement or constraint.
- *Uniformity of grammar*. By always considering the reuse of existing templates, the same kinds of requirement will always be expressed in the same kinds of way, leading to a consistent use of language in expressing requirements.

- *Uniformity of vocabulary*. Each placeholder in a template can be controlled through application of ontology. For instance, if the place-holder is expected to contain a "mode of operation", the ontology can articulate the allowable vocabulary.
- *Ensuring essential characteristics*. The use of a template aids in assuring that the requirement statement is unambiguous, quantified, testable and other essential characteristics.
- *Easier identification of repeated and conflicting requirements*. When requirements are expressed in uniform way, automatic detection of potentially repeated or similar, requirements becomes possible. Likewise, candidates for contradictory requirements can be identified.
- *One-stop control over expression*. With effective tool support, global changes in the way requirements are expressed can be effected by just changing the template in one place. E.g. changing "will" to "shall".
- *Protection of classified information*. Often the only part of a classified requirement is the quantity associated with it – e.g. the speed of the vessel, the rate of fire. Using templates, the classification can be focused on just the restricted attributes, allowing the rest of the requirement to be viewed for non-restricted use.

Terminology

We wish here to make a distinction between "templates", "boilerplates" and "patterns". The term "template" is most often used in the field of requirements management to refer to the heading structure of a requirements document. The use of such templates helps ensure that the authors consider the complete range of concerns when collating a set of requirement statements.
Because of this use of the word "template", early work [18] in the subject of this paper coined the term "boilerplate" to refer to a grammatical structure for an individual textual requirement. In this paper, we use the term "statement-level template" to refer to this concept.
"Patterns" usually refer to something broader than an individual statement. Design patterns [21], for instance, are bundles of artefacts that frequent occur together when defining a particular kind of design. One could imagine requirements patterns as sets of related requirements that together provide a complete specification of a particular feature. This paper is not about such patterns.

Related work

Existing work on the concept of requirements templates falls into three categories:

- The use of controlled English grammar to express properties that have an underlying formal semantics.
- The use of templates in tabular rather than textual form.
- The use of statement-level templates.

Approaches related to formal models

Examples of the use of statement-level templates and patterns associated with formal notations include:

- *(1999) Finite state automata and Petri Nets*. In [6], specification patterns are related to a formal semantics based on finite state automata. This work is later extended by [7] to cover real-time aspects. Here the patterns are given projections into three different formalisms.
- *(2001) Safety. Finite state automata* in the safety domain are considered by [8].
- (2008) *Probabilistic verification techniques*. The work of [5] applies the concept of specification patterns to make it easier for authors to formulate probabilistic properties of systems, improving the acceptability of formal verification techniques.
- *(2009) Requirements Documents generation*. The use of formally declared requirements patterns, as in [13] allows semi-automatic techniques to create software requirements documents.

The frequently cited objective of using such templates or patterns is to allow the specification author to specify system properties in familiar language controlled in such a way that there is an underlying formal semantics. The advantages of this are clear: automated formal verification and model-checking techniques can be brought to bear on the specification despite the requirements being expressed in natural language. The most relevant drawback has to do with the difficulty to use a strongly formal approach to write requirements.

Approaches that use tabular templates

Examples of approaches that use tabular templates or patterns without formal semantics are:

- *Planguage*. In [9], tabular templates with keywords are used to specify requirements rather than relying on grammatical sentences.

One of the key motivations for this is to ensure that an appropriate set of performance requirements and constraints are considered for each functional requirement, thus ensuring completeness. The consistent use of keywords also ensures uniform and unambiguous language.

- *Volere*. In [10], a form of tabular template is proposed called a "snow card", which ensures that certain information, such as rationale and requirement type, is associated with each requirement statement.

These approaches are not primarily textual-requirement-centric, attempting to provide semi-formal representation schemas.

Approaches based on statement-level templates

In this paper, we make no attempt to associate the statement-level templates with a formal semantics. We simply control the language used to reduce the risk of ambiguity or inconsistency, and to maintain certain important characteristics of requirements statements we focus on the expression of singular textual requirements. Nor do we address the completeness of sets or sub sets of related requirements.
Other work that is most similar to this approach includes:

- *(2002) Requirements Boilerplates*. The ideas presented in Chapter 7 of [18] are the most direct predecessor of this work.
- *(2009) Easy Approach to Requirements Syntax (EARS) and Adv-EARS*. In [15] and [16], a mapping of the grammatical constructs of English has been use to formalize requirements for a specific domain: aero-engine control systems.
- *(2010) The Requirements Specification Language (RSL) of CESAR*. The CESAR research project, funded by the European Union AREMIS program, reviewed work on the use of boilerplates [14] with a view to extending and applying the approach to a number of safety-critical domains, with discussions of how to formalise the approach using ontologies.

All three of these references will be covered in greater detail in the next section.

Project Experience

The purpose of this section is to report on a number of projects that have defined and applied statement-level templates. Some of these projects were completed with no formal publication of their results; others do have related publications, and these are cited; yet others are on-going experiments.

Future Surface Combatant

In 1998, the concept of requirement statement templates was applied to a UK defence project in which requirements were being elicited for a naval warship, Future Surface Combatant (FSC). It was observed that requirement authors had difficulty in articulating certain kinds of requirement, such as timeliness and periodicity. Once shown a good example of how to do it, however, the difficulty was largely overcome. A palette of "boilerplate" requirements was established for the project containing examples of many kinds of requirement, carefully phrased to avoid ambiguity and to ensure proper quantification. (The term "boilerplate" was used to distinguish the concept from requirements document templates, which consist purely of a heading structure.)
The palette was organised into types of property or constraint, and each boilerplate consisted of a complete sentence with place-holder words that the author could replace with specific terms. The placeholders were named consistently across all the boilerplates, so that, for instance, "<stakeholder>" and "<unit of volume>" might appear in multiple boilerplates where the same concept was referenced.
The requirements writing process was defined as follows:

1. Consider what kind of requirement or property you wish to articulate.
2. Select the most appropriate boilerplate from the palette.
3. Fill out the place-holders in the boilerplate to form the specific requirement.

As enthusiasm for the concept grew, tool support was provided that supported the requirements author in steps 2 and 3. The tool encouraged the author to select to use consistent vocabulary across requirements by offering terms already used for the same placeholder in other requirements. Beyond this, no attempt at formalising an ontology or glossary was implemented.
With tool support, a requirement statement became, in effect, a boilerplate and a set of values assigned to the placeholders. By representing the requirements in this fashion, it became possible centrally to control and evolve the precise way in which different types of requirement were expressed. Simply by changing the global boilerplate, the text of all requirements that used the boilerplate was also updated.

It also became possible to abstract from the requirements all the values that had been used for the placeholders. For instance, the placeholder "<function>" gave a list of all the functions mentioned, and "<user>" gave a list of all users mentioned.
Table 1 show some examples of the boilerplates that were used. In the left-hand column are the boilerplates hierarchically organised into clauses that may be combined to construct complete requirement statements. The right-hand column shows two things: the classification of the combined clauses, and a complete example instantiation.

Table 1: Example boilerplates from FSC

BOILERPLATE	*CLASSIFICATION & EXAMPLE INSTANTIATION*
While <operational condition>,	<u>Mode</u> While *disconnected from a source of power, ...*
... the <user> shall able to <capability>	<u>Mode/Capability</u> While *disconnected from a source of power*, the *photographer* shall able to *capture still images*
... at least <quantity> times per <time unit>	<u>Mode/Capability/Rapidity (Maximise, Exceed)</u> While *disconnected from a source of power* the *photographer* shall able to *capture still images* at least *10* times per *minute*
... for a sustained period of at least <quantity> <time unit>	<u>Mode/Capability/Rapidity/Sustainability (Maximise, Exceed)</u> While disconnected *from a source of power* the *photographer* shall able to *capture still images* at least *10* times per *minute* for a sustained period of at least *4 hours*
... the <system function> shall able to <action> <entity>	<u>Mode/Function</u> While *in winds up to Beaufort scale 8* the *missile launcher* shall able to *fire missiles*
... within <quantity> <time unit>s from <event>	<u>Mode/Function/Timeliness (Minimise, Do not exceed)</u> While *in winds up to Beaufort scale 8* the *missile launcher* shall able to *fire missiles* within *5 seconds* from *receipt of warning signal*

The application of this approach in the project had a positive effect on the confidence of the requirements authors in expressing the requirements. At the end of the exercise, the project had developed a palette of about 120 boilerplates for different kinds of requirement. This palette represented an intellectual asset to the organisation to pass on to future projects.

Baggage Handling

A similar approach to the above was applied by the British Airports Authority (BAA) in gathering requirements for the London Heathrow Terminal 5 Baggage Handling System, but without specific tool support. Their boilerplates were referred to as "statement level templates" (as opposed to document templates), and templates were provided at the stakeholder level for capabilities, constraints, assumptions and definitions. A different style of template is used at the application-specific level, where more structure can be prescribed.
Table 2 shows an example capability template used in the baggage handling project.

Table 2: Example capability template from baggage handling

Templates

<Actor> shall be able to <Act> <Action_Subject> whilst <Qualification> <Qualification_Subject>.

<Actor> shall be able to <Dispatch_Act> <Action_Subject> from <Dispatch_Object>.

where

<Actor> ..... the user (person or external system) who interacts with the system to be developed in order to perform the Act.

<Act> ..... the act the actor needs to perform to achieve some business goal.

<Action_Subject> ..... the part of the application domain which will be affected by the Act.

<Qualification> ..... a description of the circumstances under which the Actor should be able to perform the specified Act.

<Qualification_Subject> the part of the application domain over which the Qualification is defined.

<Dispatch_Act> ..... the dispatch act the Actor needs to perform to achieve some business goal. (A dispatch act engages in a different grammatical construct, because it is expected to require an indirect object, the Dispatch_Object).

<Dispatch_Object>..... the location from which the dispatch takes place.

As an example how the capability template was used, an original, poorly expressed user requirement was re-expressed as shown in Table 3. By being constrained to use provided templates, requirements authors are guided into expressing requirements in a uniform style at the appropriate level of abstraction. Hence the emphasis is shifted from solution to capability.

Table 3: Re-expression of user requirement

Original requirement

Each check-in desk shall be equipped with weigh, label and take-away conveyors, feeding collector conveyors.

Re-expression as a set of capability requirements

Check-in clerks shall be able to *weigh std baggage items* whilst *stationed at the check-in desk*.

Check-in clerks shall be able to *label std baggage items* whilst *stationed at the check-in desk*.

Check-in clerks shall be able to *despatch to a take-away conveyor std baggage items* from *the check-in desk*.

Check-in clerks shall be able to *despatch to a feeding collector conveyor std baggage items* from *the check-in desk*.

Requirement author reaction to this approach was mixed: some complained that they felt the templates restricted their freedom of expression. However, the project managers persisted with the approach because they felt the benefits of uniformity of expression outweighed the constraints.

Easy Approach to Requirements Syntax (EARS)

Reported in [15] and [19] are some experiments in apply the statement-level template approach to the specification of properties of aero-engine control systems. Because the domain is so specific, and applied at a technical system level rather than user level, the experiment was able to use just 4 statement templates to cover the majority of requirements. These are shown in Table 4.

Table 4: Templates for control systems

Ubiquitous requirements The <system name> shall <system response>	*Example* The *control system* shall *prevent engine overspeed*
Event-driven requirements WHEN <optional preconditions/trigger> the <system name> shall <system response>	*Example* WHEN *continuous ignition is commanded by the aircraft*, the *control system* shall *switch on continuous ignition*
Unwanted behaviours IF <optional preconditions/trigger>, THEN the <system name> shall <system response>	*Example* IF *the computed airspeed fault flag is set,* THEN the *control system* shall *use modelled airspeed*
State-driven requirements WHILE <in a specific state> the <system name> shall <system response>	*Example* WHILE *the aircraft is in-flight*, the *control system* shall *maintain engine fuel flow above XXlbs/sec*

Note the use here of capitalised key words to make it clear exactly which template is in use.

Reference [15] reports an improvement in requirement quality when judged by a small set of criteria. The later reference [19] evolves the approach, adds some more templates, and uses more refined criteria for assessing the benefits accrued.

Context RDS

Context RDS is an experimental toolkit that develops the concept of statement templates strongly towards the use of domain-specific ontologies. The tool is being applied to projects within the aero-engine division of Rolls-Royce where templates are being used to frame requirements for engine control systems, a domain that lends itself well to stylised requirements.

In RDS, statement-level templates contain placeholders that may be of the following types:

- Process Defined
- Value by Reference
- User Defined Term
- User Defined Text

Placeholders that obtain their value by reference can only be populated by terms of one or more type, defined in a project-specific process model. For example, "While <Condition> is....." has a placeholder that can be constrained to allow references to atomic condition or complex conditions; atomic conditions have further constraints. User Defined Terms have templates with similar placeholders, thus allow terms to reference other terms.
Figure 1 shows an example template structure, and figure 2 shows an example instantiation.

Figure 1: A Context RDS Template Structure

The configuration of the tool constrains how authors are able to instantiate the templates. This ensures, for instance, the consistent use of terms and units, and prevents nonsense requirements created by meaningless instantiation.

It is considered too early in the experience of using Context RDS to be able to report specific feedback. However, it is evident that considerably more up-front work and cost is engendered by this approach; it is hoped that the investment will return dividends later in the project.

Figure 2: An instantiated template in Context RDS

The Requirements Specification Language (RSL) of the CESAR project

A main goal of the European-funded CESAR project was to consider way of improving the development process for a number of safety-critical domains, such as aerospace and rail. In their analysis of opportunities in requirements engineering, the use of boilerplates was identified as candidate for bringing about improvement.

A key CESAR report [14] considered two things together: a Requirements Specification Language (RSL) largely based on the boilerplate concept,

and a Requirements Meta Model (RMM) consisting of a general and domain-specific ontology for supporting requirements vocabulary.

The report notes that such ontology provides the semantic base for which requirements can be analysed, validated and verified.

Figure 3: An example ontology underpinning requirement statements

Figure 3 shows an example ontology, from which a variety of statement-level templates could be generated by traversing parts of the structure as appropriate. The nodes are typed entities that form the placeholders, and the arcs describe the relationships between those entities.

Requirements Authoring Tool (RAT)

Another project relates to a recently released product, the Requirements Authoring Tool (RAT) from The Reuse Company [17]. RAT allows the user to start typing a requirement, and simultaneously determines which of a palette of statement-level templates are applicable, guiding the user on permissible terminology. If the requirement that the user is typing eventually matches a template, the user is informed of the fact.

To achieve this, it combines three things:

- A set of pre-determined templates.
- A semantic parsing engine.
- A strong underlying domain-specific ontology.

The combination of these properties allows the requirements author to write requirements in an assisted way, as a counterpart to the "fill the gap" approach to placeholders. The possibility of not selecting a-priori a pattern before typing a requirement leads to the tool being considered as an assistant, as well as a coach, and users may feel their creativity is less constrained.
At present, the tool comes equipped with about 70 statement-level templates that can be combined in various ways. RAT allows the evolution of the set of templates and associated ontology to be managed as part of the enterprise's knowledge base. In doing this, it introduces the specialist role of "knowledge manager", in addition to that of "author". If authors find that the tool is able to match templates to the requirements they are trying to express, they can apply to the knowledge manager to introduce new ones, with associated terms. Such a role requires an understanding of the purpose of ontology, and how to use the tool to define it.

Future Directions

This section presents some ideas for the future development of the approach.

Extension to artefacts other than requirements

Works so far has focused on controlling the expression of requirements. Other artefacts closely related to requirements may benefit from the same approach. For instance:

- Descriptions of V&V activities, their criteria and expected outcomes.
- Requirements flow down structures and associated decomposition rationale. (See the concept of rich traceability [22].)

The two aspects just mentioned relate to two key relationships that are typically articulated during requirements management activities: the decomposition (or satisfaction) relationship between layers of requirements, and the association of test activities with requirements. In both cases, a particular type of requirement will

typically require similar patterns of decomposition and testing. This gives rise to the idea of extending the statement level templates into design patterns that bundle a requirement with its decomposition structure and test structure.

Conceptual representation of requirements

In the templating approach described here, the textual requirement is the primary representation, because there is no underlying formal semantics. In the other approaches described in section 2, the textual representation is merely a projection of a formal construct. The absence of a formal semantics makes the approach applicable to the authoring of requirements that are at a level of abstraction that does not lend itself to formal modelling.

However, it may be possible to abstract away from the textual representation by using conceptual graphs in the style of [11]. More general than any specific formal specification language, conceptual graphs are a candidate for a representation of requirements and system properties from which textual requirements could be projections. From a single conceptual representation of a requirement, a textual statement could be project in any chosen natural language. Projections from conceptual graphs into modelling notations, such as SysML may also be possible.

Conclusion

We have attempted to describe hitherto unpublished work on the use of templates – or boilerplates – for textual requirements statements. We have described the general approach, with several specific applications, and a number of tools that offer support. The only available quantitative analysis of the benefits of such an approach is based on small studies [15] [19]. Real practical experience does strongly suggest, however, that the use of templates offers benefits to authors of requirements and manager of projects.

References

[1] Guide for Writing Requirements, INCOSE Product INCOSE-TP-2010-006-01, V. 1, April 2012 (requires login) https://connect.incose.org/products/SAWG%20Shared%20Documents/Technical%20Resources/SE%20Practice%20-%20Technical_Mgmt/Guide_for_Writing_Requirements%202012-0417%20RWG%20TO%20Approved.pdf

[2] LEXIOR from Cortim, http://www.cortim.com/pageLibre0001010b.html

[3] Requirements Quality Analyser from The Reuse Company, http://www.reusecompany.com/index.php?option=com_content&view=category&layout=blog&id=171&Itemid=75&lang=en

[4] www.requirementsengineering.info/boilerplates

[5] Grunske, L.: Specification patterns for probabilistic quality properties. In: ICSE, New York, ACM (2008) 31-40

[6] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE, New York, ACM (1999) 411-420

[7] N. Abid, S. Dal Zilio, and D. le Botlan. A Real-Time Specification Patterns Language. Technical Report 11364, LAAS, 2011. http://hal.archives-ouvertes.fr/hal-00593965

[8] Friedemann Bitsch, Safety Patterns - The Key to Formal Specification of Safety Requirements, Proceedings of the 20th International Conference on Computer Safety, Reliability and Security, p.176-189, September 26-28, 2001

[9] T. Gilb. *A Handbook for Systems and Software Engineering using Planguage*, Addison Wesley 2001

[10] Suzanne Robertson and James C. Robertson. *Mastering the Requirements Process*, 2nd Edition, Addison Wesley 2006.

[11] John Sowa, *Conceptual structures: Information processing in mind and machine*, Addison Wesley, 1983

[12] Genova, G. Fuentes, J. Llorens, J. Hurtado, O. Moreno, V. *A framework to measure and improve the quality of textual requirements*, Requirements Engineering, Springer, Url: http://dx.doi.org/10.1007/s00766-011-0134-z, Doi: 10.1007/s00766-011-0134-z

[13] Renault, S. Mendez, O. Franch, X. Quer C. A Pattern-based Method for building Requirements Documents in Call-for-tender Processes. International Journal of Computer Science & Applications (IJCSA), 6(5): 175-202 (2009)

[14] Definition and Exemplication of RSL and RMM, CESAR Project Technical Report D_SP2_R2.1_M1: http://www.cesarproject.eu/fileadmin/user_upload/CESAR_D_SP2_R2.1_M1_v1.000_PU.pdf

[15] Alistair Mavin, Philip Wilkinson, Adrian Harwood, Mark Novak, Easy Approach to Requirements Syntax (EARS), 2009, 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, August 31-September 04

[16] Dipankar Majumdar, Sabnam Sengupta, Ananya Kanjilal, Swapan Bhattacharya, "Adv-EARS: A Formal Requirements Syntax for Derivation of Use Case Models", Proc of the First International

Conference on Advances in Computing and Information Technology, July, 15-17, 2011, Chennai, India. pp 40-48.

[17] Requirements Authoring Tool (RAT) from The Reuse Company, http://www.reusecompany.com/index.php?option=com_content&view=category&layout=blog&id=216&Itemid=187&lang=en

[18] E. Hull, K. Jackson and J. Dick: Requirements Engineering. First Edition, Springer 2002.

[19] Alistair Mavin, Philip Wilkinson: BIG EARS (The Return of "Easy Approach to Requirements Syntax", 2010, 18th IEEE International Requirements Engineering Conference, Sydney, Australis, September 27 - October 1

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[22] "Rich Traceability", Jeremy Dick, Proc. 1st International Workshop on Requirements Traceability, Edinburgh, Sept 2002

(all hyperlinks accessed November 2012)

About the authors

Dr. Jeremy Dick was educated at London University's Imperial College in Computing Science, majoring in formal methods of software development. He went on to pursue academic and industrial research in this field for a number of years. From 1996 to 2006, he worked for software tool supplier Telelogic (now IBM) in their UK professional services organization, as a principal consultant in tool-supported requirements management. This role has afforded him a broad exposure to requirements management practices and issues across many industry sectors. In this role, he developed the concept of Rich Traceability. He co-authored the Springer book entitled "Requirements Engineering," and toured internationally giving seminars in this subject. In 2006, he moved to UK-based consultancy Integrate Systems Engineering Ltd, where he has been helping to develop and promote the concept of Evidence-based Development. He is a past Chairman of INCOSE's Requirements Working Group.

Dr Juan Llorens received his MS degree in Industrial Engineering from the ICAI Polytechnic School at the UPC University in Madrid in 1986, and his PhD in Industrial Engineering and Robotics at the Carlos III University of Madrid in 1996. In 1987 he started his own company dealing with Artificial Intelligence applied to Database systems, named Knowledge Engineering SL (KE). Dr. Llorens worked as technical director in KE and as Marketing Director in Advanced Vision Technologies until 1992, when he joined the Carlos III University. Since 2003 he has been Professor of the Informatics Department of the University. His main subject is Software and Knowledge Reuse, which he teaches in the Informatics and Information Science graduate studies department at the University. Dr. Llorens is the leader of the KR Group (Knowledge Reuse Group) within the University. In 1998 he was invited to the Högskolan på Åland (HÅ) (Åland, Finland). From 1998 to 2008 he split his educational activities between Madrid's University and the HÅ, where he taught Object Oriented Software design. Since 2008, based on an agreement with his University, he shares his academic duties with the position of R+D officer in The Reuse Company (a brand of the University's spin-off organization named CISET) that commercializes systems engineering quality and reuse technology. Since 1999 he has been the board of CISET.

His current research involves the study of Knowledge technologies and System Engineering techniques integration towards Software and Information quality measurement and reuse. He is member of INCOSE's (International Council on Systems Engineering, www.incose.org) Requirements Working Group.

His CV is presented at http://www.kr.inf.uc3m.es/juanllorens.htm

About Integrate

integrate's vision is for companies to align strategy and execution, drive quality and ensure compliance through using systems engineering techniques as the bridge between the domains of engineering and management. Our consultancy practice and supporting technologies help clients deliver this vision. We use systems engineering as a vehicle to support informed management and direction, and to optimise solutions against both technical and business constraints. Our approach is intensely pragmatic, firmly focused on our clients' delivery needs, and founded on a thorough understanding of the underlying problem.

For more information go to http://www.integrate.biz or email info@integrate.biz

This is an edited version of a paper presented at ICSSEA 2012, the 24th International Conference on Software & Systems Engineering and their Applications

© 2012 integrate systems engineering ltd

All rights reserved. This document is copyright of Integrate Systems Engineering Ltd.

Integrate Systems Engineering Ltd is registered in England no 4588165 and has its registered office at 21 St Thomas Street, Bristol BS1 6JS

Find out more

If you require further information, please contact

Sales Manager
t: +44 (0)1225 859991
e: sales@integrate.biz

integrate systems engineering ltd

251-253 London Road East
Batheaston
Bath BA1 7RL
United Kingdom

www.integrate.biz

Software Reading Techniques

Twenty Techniques for More Effective Software Review and Inspection

—

Yang-Ming Zhu

Apress®

Software Reading Techniques

Twenty Techniques for More Effective Software Review and Inspection

Yang-Ming Zhu

Apress®

Software Reading Techniques: Twenty Techniques for More Effective Software Review and Inspection

Yang-Ming Zhu
Solon, Ohio
USA

ISBN-13 (pbk): 978-1-4842-2345-1 ISBN-13 (electronic): 978-1-4842-2346-8
DOI 10.1007/978-1-4842-2346-8

Library of Congress Control Number: 2016959789

Copyright © 2016 by Yang-Ming Zhu

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr
Acquisitions Editor: Robert Hutchinson
Developmental Editor: Laura Berendson
Technical Reviewer: Weidong Liao
Editorial Board: Steve Anglin, Pramila Balen, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Rita Fernando
Copy Editor: Angela Warner
Compositor: SPi Global
Indexer: SPi Global
Cover Image: Designed by Pio_pio - Freepik.com

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

Printed on acid-free paper

To Xiao-Hong, Harold, Alex, and the rest of my family.

Contents at a Glance

About the Author xiii

About the Technical Reviewer xv

Acknowledgments xvii

Introduction xix

■ Chapter 1: Introduction 1

■ Chapter 2: Software Review Procedures 7

■ Chapter 3: Basic Software Reading Techniques 21

■ Chapter 4: Scenario-Based Reading Techniques 35

■ Chapter 5: Requirements Reading Techniques 69

■ Chapter 6: Design Reading Techniques 77

■ Chapter 7: Code Reading Techniques 103

■ Chapter 8: Conclusion 119

Index 123

Contents

About the Author xiii

About the Technical Reviewer xv

Acknowledgments xvii

Introduction xix

Chapter 1: Introduction 1

1.1 Software Quality, Software Reviews and Inspections 1

1.2 About This Book 4

1.2.1 Organization of This Book 4

1.2.2 Intended Audience and How to Use This Book 5

1.3 References 5

Chapter 2: Software Review Procedures 7

2.1 A Generic Software Review Procedure 7

2.2 Fagan Inspection and Extensions 8

2.2.1 Fagan Inspection 8

2.2.2 Extensions to Fagan Inspection 11

2.3 Active Design Review and Extensions 13

2.3.1 Active Design Review 13

2.3.2 Extensions to the Active Design Review 15

2.4 Other Types of Reviews 15

2.5 Factors Impacting Software Reviews 16

2.6 Summary 18
2.7 References 19
Chapter 3: Basic Software Reading Techniques 21
3.1 Introduction to Software Reading 21
3.1.1 Definition of Software Reading 22
3.1.2 Purposes of Software Reading 22
3.1.3 Taxonomy of Software Reading Techniques 23
3.2 Ad hoc Reading 23
3.3 Checklist-Based Reading 24
3.3.1 Checklist Definition, Types, and Examples 24
3.3.2 Checklists with Guidance 26
3.3.3 Best Practices of Checklists 28
3.3.4 Empirical Experiences 28
3.4 Differential Reading 29
3.5 Summary 32
3.6 References 33
Chapter 4: Scenario-Based Reading Techniques 35
4.1 Principles of Scenario-Based Reading 35
4.2 Defect-Based Reading 37
4.2.1 Taxonomy of Defects in Requirements Specifications 38
4.2.2 Defect-Based Reading Techniques 39
4.2.3 Empirical Experiences 41
4.3 Perspective-Based Reading 42
4.3.1 A Generic Perspective-Based Reading 42
4.3.2 Perspective-Based Requirements Reading 45
4.3.3 Perspective-Based Design Reading 48
4.3.4 Perspective-Based Code Reading 52

4.3.5 Perspective-Based Usability Reading 54
4.3.6 Why Does Perspective-Based Reading Work? 58
4.4 Alternative Partitioning of Reading Responsibilities 64
4.4.1 Ad hoc Partition 64
4.4.2 Function Point-Based Partition 65
4.5 Summary 66
4.6 References 66
Chapter 5: Requirements Reading Techniques 69
5.1 Critical Roles of Requirements in Software Development 69
5.2 A Combined Reading Technique for Requirements 70
5.2.1 Motivations for a Combined Reading 70
5.2.2 The Combined-Reading Technique 70
5.2.3 Empirical Experiences 72
5.3 Test-Case Driven Reading for Requirements 72
5.3.1 Test-Case-Driven Reading Technique 72
5.3.2 Empirical Experiences 74
5.4 Individual Factors Impacting Requirements Reading Efficiency 74
5.5 Summary 75
5.6 References 76
Chapter 6: Design Reading Techniques 77
6.1 Introduction 77
6.2 Usage-Based Reading 77
6.2.1 Usage-Based Reading Technique 78
6.2.2 Variations of Usage-Based Reading 80
6.2.3 Empirical Experiences 80
6.3 Traceability-Based Reading 81
6.3.1 Types of Design Defects 82
6.3.2 High-Level OO Designs Using UML Diagrams 82

6.3.3 Traceability-Based Reading Techniques 83
6.3.4 Empirical Experiences 92
6.4 Architecture Reading 93
6.4.1 What Is Software Architecture? 93
6.4.2 Traceability-Based Architecture Reading 94
6.4.3 Empirical Experiences 95
6.4.4 Other Architecture Reading Techniques 96
6.5 Scope-Based Reading 97
6.5.1 What Is an Application Framework? 97
6.5.2 Scope-Based Reading Techniques 97
6.5.3 Empirical Experiences 99
6.6 Summary 100
6.7 References 100
Chapter 7: Code Reading Techniques 103
7.1 Code Reading As a Professional Skill 103
7.1.1 Importance of Code Reading 103
7.1.2 How Do People Read Code? 104
7.2 Reading by Stepwise Abstraction 105
7.3 Object-Oriented Code Reading 107
7.3.1 Challenges of Object-Oriented Code Reading 107
7.3.2 Abstraction-Driven Reading 108
7.3.3 Use-Case-Driven Reading 109
7.3.4 Empirical Experiences 110
7.4 Object-Oriented Framework Code Reading 111
7.4.1 Why Yet Another Object-Oriented Code Reading Technique? 111
7.4.2 Functionality-Based Approach to Framework Understanding 112
7.4.3 Functionality-Based Reading 113
7.4.4 Empirical Experiences 114

7.5 Task-Directed Inspection 114
7.6 Code Readability Factors 115
7.7 Summary 116
7.8 References 117

Chapter 8: Conclusion 119

Index 123

About the Author

Yang-Ming Zhu is a Principal Scientist at Philips Healthcare, currently serving as the software architect for the Recon and Imaging Physics team for Advanced Molecular Imaging. He practices and researches image processing and software engineering with a focus on software architecture, requirements engineering, best practices, software quality, and processes. He is a senior member of IEEE and has published more than 80 book chapters and papers in such journals as IEEE Software, IT Professional, IEEE Trans Medical Imaging, IEEE Trans Image Processing, IEEE Trans Signal Processing, Journal of Nuclear Medicine, Physical Review Letters, and Applied Physical Letters. He holds nine U.S. patents (additional seven are pending approval), the Software Architecture Professional Certificate from the Software Engineering Institute at Carnegie Mellon University, and advanced degrees in computer science (MS from Kent State University), biomedical engineering (MS/BS from Shanghai Jiaotong University), and bioelectronics (PhD from Southeast University).

About the Technical Reviewer

Dr.Weidong Liao is currently an associate Professor at Shepherd University, Shepherdstown, West Virginia. Dr. Liao received his Masters and Ph.D. degrees in Computer Science from Kent State University and had software development experience in Microsoft Corporation and other companies. Dr. Liao made his major research contributions in Internet Accessible Mathematical Computation (IAMC) Application Framework, Internet and Distributed Computing, and Distributed Database System, with numerous publications in these fields. He has also participated in the review process of many conference articles and journal papers in computing areas.

Acknowledgments

The idea of this book was inspired by Kollanus and Koskinen's article, Survey of Software Inspection Research, and based on my presentation to colleagues at Philips on code reading techniques. I thank the researchers and practitioners in the software review/inspection community, who created and validated many software reading techniques upon which the book is based.

Many thanks to Acquisitions Editor Robert Hutchinson, who bestowed trust on me and answered many questions a first-time book author might have; to Developmental Editor Laura Berendson, who edited the book and provided guidance on writing; to Coordinating Editor Rita Fernando, who designed the book cover and made sure I would be on schedule; to Technical Reviewer Professor Weidong Liao, who made sure the book contents are technically sound; and to the rest of the Apress team who supported the project behind the scenes. I would also like to thank Harold Zhu, who improved the presentation in Chapters 1, 3, 5, and 7, and Kevin Wang, who provided suggestions to improve the language in Chapters 2 and 6. Any remaining errors are mine, however. I am profoundly grateful to my mentor Steve Cochoff at Philips for his guidance and friendship for the past almost 20 years.

Most of all I want to thank my wife, Xiao-Hong, and two wonderful boys, Harold and Alex, for letting me disappear at nights and over weekends to research for and write this book. Harold also helped me track down many of the original papers on reading techniques. I am not certain the joy in seeing this book finally published can entirely compensate for the lost family time together.

—Yang-Ming Zhu
Solon, Ohio, September 2016

Introduction

There are many best practices in the software development arena, and there is no other practice as prominent as software review or inspection that has enjoyed the universal agreement on its effectiveness (it finds many defects), efficiency (it finds defects at a low cost), and practicality (it is easy to carry out).

Ever since Fagan published his seminal Fagan Inspection in 1976, software review or inspection has evolved in many aspects. Now the whole industry favors a lightweight, tool-assisted method. Nevertheless, the core of the software review remains the same, i.e., the software review is predominantly an individual reading and analysis activity, since many issues are observed during an individual's preparation phase. The individual reader's skill, training, and expertise play a significant role during reading and analysis and determine the success or failure of software review or inspection. In practice, a wide difference in individual's effectiveness is reported, often as big as 10 times.

As software development professionals, we are trained to write software artifacts such as requirements specifications, design diagrams and descriptions, code modules, test cases, or user interface mockups. We are not trained to read and analyze the documents written by peers, however. To increase an individual's capacity during software review or inspection, reading strategies and proven practices are packaged as reading techniques. This book is designed to promote and disseminate proven reading techniques for professional training and industrial adoption.

There are many reading techniques. To navigate them, the book first discusses basic and more advanced techniques that are applicable to many, if not all, software artifacts, then introduces techniques specially designed for requirements specifications, designs, and code modules. We hope this framework will support your roadmap to adopt the reading techniques in your organization or project.

The author enjoyed researching and writing the book. He certainly hopes you will enjoy reading it and most importantly benefit from adopting these reading techniques in your own practices.

CHAPTER 1

Introduction

Software has become an indispensable part of our daily lives, as our dependence on software is constantly increasing. Software fails, and these failures cost money, time, resources, and sometimes even lives. An organization's reputation and success depends on its ability to create and deliver high-quality software. Software quality is thus critically important. This chapter introduces the concept of software quality assurance, discusses software inspection (the most frequently performed software quality assurance activity other than testing), and outlines the contents of this book.

1.1 Software Quality, Software Reviews and Inspections

The software creation process requires a lot of manpower, from requirements and design to implementation and testing. As human beings, we all make mistakes, which are manifested as defects or bugs in software products. Software development is a complex process and tools are frequently used to assist development. People's familiarity with tools and their interaction with tools are the other complicating factors that can potentially inject defects into software products. The nontrivial software product development nowadays may involve many people and teams across the globe. Coordinating the distributed team members and tasks remains a management challenge. Despite many years of software engineering efforts, we are still unable to deliver defect-free software products to the end users. Software defects seem to be inevitable.

Software quality assurance is a structured approach to improve software quality and involves defect prevention, detection and removal, and defect containment activities. Defect detection and removal is mostly emphasized in industry practice, particularly in non-safety critical systems. If it is hard or impossible not to inject defects into software, the next best thing to do is to remove them before they reach end users. It is highly desirable to detect and remove defects as soon as they are injected into software, since defects can be several times more expensive to fix at latter stages of a software project. The escaped or residual defects in field deployment may cause interruption to the user's normal operations, resulting in substantial costs including reduced productivity, data loss or corruption, security vulnerabilities, or even physical harm.

© Yang-Ming Zhu 2016

Y.-M. Zhu, *Software Reading Techniques*, DOI 10.1007/978-1-4842-2346-8_1

Testing and inspection are the two most effective and commonly used methods to detect defects in software. In software testing, a tester runs a software system or its components, executes its functions, observes system behaviors or responses, and determines if the system behaviors conform to its requirements, specifications, or expectations. To test software, one needs a running executable, which is not feasible all the time, particularly for artifacts rather than code. The effectiveness of testing also depends on test cases, and if there is no test case to exercise a particular path of the code, there is no way to tell if there is a defect in that path or not. Dijkstra once famously said that testing can only confirm the presence of a defect, not the absence of a defect (https://en.wikiquote.org/wiki/Edsger_W_Dijkstra).

There are many other quality assurance alternatives besides testing. Software review or inspection is one of these alternatives that is widely practiced. The terms "software review" and "software inspection" are not used consistently in literature. IEEE Standard 1028 defines five types of software reviews: management reviews, technical reviews, inspections, walkthroughs, and audits (IEEE 1028); it treats software inspection as a type of software review. In general software engineering literature, inspection is specific and has a defined process, and review is a more general term. We will use these two terms interchangeably in this book.

Software inspection is a formalized peer review process applicable to any software artifact. It is a static analysis method and was first introduced by Fagan based on his practical experience at IBM in the 1970's (Fagan, 1976). Fagan inspection has been influential ever since it was published. In fact, the IEEE Standard 1028 is largely based on Fagan inspection.

Based on decades of research and industry practices, it is widely established that software inspection is effective (it finds many defects), efficient (low cost per defect), and practical (easy to carry out). We will review the status of software review and inspection in Chapter 2. It has been reported that inspection can detect (and correct) 20% to 90% of defects (Gilb & Graham, 1993, Laitenberger, 2002). The benefits and cost-effectiveness of software inspection have long been recognized. Doolan (1992) reported that every hour invested in inspection has a payback of 30 hours and Russell (1991) claimed that in his organization, every inspection hour saved 33 hours of maintenance work. Although the earlier benefits might be overstated (Porter et al., 1996; Laitenberger, 2002), more recent simulation results have revealed that code inspection saves 39% of defect costs compared to testing alone, and design inspection saves 44% of defect costs compared to testing alone (Briand et al.,1998). Software inspection nowadays is easy to conduct. Books have been published about it (e.g., Gilb, 1993) and standards on review procedures are also available (e.g., IEEE 1028).

During software development, many documents are generated by humans, e.g., requirements specifications, design, code, test plans, test cases, and user documentation. If these documents are used to guide subsequent activities, any errors in these documents may propagate to the downstream activities and artifacts. It is critical to catch and fix errors early to prevent them from propagating down the stream. It is much cheaper for the development organization to fix an error when it is introduced. It is believed that, in general, it would be 10 times more expensive to fix an error if it slips to the next phase. Defects in the released product may result in substantial costs to end users as well.

Table 1-1 shows the cost of fixing a defect depending on when the defect is injected into the software and when the defect is fixed (McConnell, 2004). The first column shows when the defect is injected and the first row shows when the defect is fixed. It assumes the cost to fix the defect injected in the same development phase is 1x. Although the real cost varies and depends on the project size and industry, the general trend is clear.

Requirements, design, and other non-executable artifacts cannot be tested by machines and have to be reviewed by humans. Requirements not implemented yet cannot be machine tested but can be analyzed by humans to see if they meet user needs. Design, particularly early design sketches, can be abstract, imprecise, and incomplete. It is not possible to test designs by machine. With inspection, both requirements and design can be read and analyzed, which does not require the artifacts to be complete or executable. Even for program code, which can be machine-tested, testing and inspection are complementary and uncover different kinds of issues (Basili & Selby, 1987; Juristo & Vegas, 2003). Issues hard to uncover via testing often can be detected by inspection relatively easily. Mantyla and Lassenius (2009) affirmed that code reviews are a good tool for detecting code evolvability defects (documentation, visual representation, and structure) that cannot be found in later phase testing, since the defects do not affect the software's visible functionality. Inspection can be applied to test plan and test cases as well, which further improves the defect detection efficiency of testing.

Table 1-1. *The Cost of Fixing a Defect (adapted from McConnell, 2004)*

	Requirements	Design	Coding	Testing	Deployment
Requirements	1x	3x	5-10x	10x	10-100x
Design	-	1x	10x	15x	25-100x
Coding	-	-	1x	10x	10-25x

The immediate benefit of software review is to detect and fix errors or problems in software artifacts. As a side effect of inspection, the software artifacts become more readable and maintainable. There are other derived benefits to individuals who participate in the review and to the organization as a whole. The author who created the artifacts can use the review outcome as a learning instrument, e.g., he can avoid similar errors in his future work. To other participants, they can also learn from another person's mistakes, particularly if they did not catch the mistakes in their own reviews. The software review can be used to cross-train team members and build a stronger team. Inspection enables a team to share technical expertise. To the organization, fewer mistakes lead to higher quality and better customer satisfaction and less rework and higher development productivity.

If the organization builds a knowledge database of past errors and problems uncovered during software inspection, they can use it to their competitive advantage. In the case of code review, one can identify the most error-prone modules and decide whether redesign or recoding is worthwhile. Certainly more review and more testing in those problematic areas are warranted before the software is released. The organization can also look at the distribution of error types via the Pareto chart and perform root cause analysis of the most common errors. For common systematic errors, the organization can then decide if additional training, better tooling, or an improved development process could help.

1.2 About This Book

In Fagan's inspection and its derivatives, individuals typically read software artifacts, then come together to discuss their findings and hopefully uncover new ones. The defect collection meeting is considered critical. There is convincing evidence, however, that defect detection in software artifacts is primarily an individual effort and happens during individual preparation and reading (Johnson & Tjahjono, 1998; Votta, 1993), and the purpose of the meeting is mostly to agree on true defects and dismiss false positives. In the state-of-industry practice, the meeting is either removed or not the emphasis anymore. Instead, how the inspectors examine the software artifacts becomes important. To improve an individual's effectiveness, various reading techniques have been proposed and tested. Reading is a key technical activity for verifying and validating software (Basili, 1997). This book is devoted to software reading techniques.

1.2.1 Organization of This Book

Many reading techniques have been developed and tested. This book categorizes them based on their characteristics. Some reading techniques are generic and can be used for many, if not all, software artifacts, while other techniques are only applicable to specific artifacts. This book discusses general reading techniques first, then specific techniques for software artifacts such as requirements, design, and code. The following lists the contents of each chapter.

Chapter 2 discusses the software review procedure. In particular, we discuss Fagan inspection and its extensions. Active design review is discussed in detail as well, since it is the basis of some recent reading techniques such as scenario-based reading.

Chapter 3 defines the terms of software reading and software reading techniques. It discusses reading purposes and classifies existing reading techniques. Ad hoc reading and checklist-based reading are discussed in detail, including various extensions to checklist-based reading, since they are frequently used as baselines and other reading techniques are compared to them. Chapter 3 also discusses differential reading, which can be used to inspect artifacts under evolution.

Chapter 4 focuses on scenario-based reading. Although many reading techniques are classified as scenario-based reading, we discuss defect-based reading, perspective-based reading, and function-point-based reading, as examples of scenario-based reading techniques. Perspective-based reading has been adapted to inspect requirements specifications, design documents, source code files, and user interface usability. The cognitive process of perspective-based reading is also discussed. A simple analytical model is used to shed light on when perspective-based reading could outperform other reading techniques.

Chapter 5 discusses reading techniques specific to requirements specifications. We present a combined reading technique which takes advantages of the strengths of individual reading techniques, while compensating for their weaknesses. Test-driven reading is introduced as an economic reading technique for organizations with limited resources.

Chapter 6 discusses specific reading techniques for design. While most reading techniques gear to find as many defects as possible, not all defects have the same impact on end users. Usage-based reading uses prioritized use-cases as a guiding light to focus readers' attention on defects that matter to users most. This chapter also discusses traceability-based reading that checks for consistency among all design artifacts and between design and requirement artifacts. It is a family of techniques organized as

horizontal and vertical readings. Traceability-based reading is also applied to architecture review. Although most of the time we read software artifacts to detect defects, we read software for construction under some circumstances. Scope-based reading is designed to enhance a reader's ability to understand object-oriented application frameworks, which the reader can then use to design and implement their own applications.

Chapter 7 is devoted to specific code reading techniques. It covers reading by stepwise abstraction first and then illustrates how it is extended as abstraction-driven reading for object-oriented code. There are unique challenges in object-oriented code reading. While abstraction-driven reading focuses on the static behavior of an object-oriented system, use-case-driven reading is concerned with the dynamic behavior. Legacy software applications are abundant without much documentation. Task-directed reading can be used to improve code quality and complete the necessary documentation on code and design.

Chapter 8 concludes the book and encourages readers to apply reading techniques to their practice.

1.2.2 Intended Audience and How to Use This Book

This book is intended for software engineering practitioners. We suggest that readers start with Chapters 2 and 3 to get an overview of software inspection and preliminary reading techniques.

- If you are interested in inspecting software requirements specifications, you can continue with Chapters 4 and 5. You may skip non-requirements applications of perspective-based reading in Chapter 4, however.
- If you are interested in inspecting design documents, you can read part of Chapter 4 (Section 4.3.3) and Chapter 6.
- If you are interested in inspecting source code, you can read part of Chapter 4 (Section 4.3.4) and Chapter 7.
- If you are interested in usability inspection, you can read Chapter 4 (Section 4.3.5).

This book is also intended for software engineering students and can be used to supplement courses such as software engineering, software quality, and software testing. As students are preparing themselves to join the software industry as software engineers, it is beneficial for students to read the entire book.

This book can be used by software engineering researchers, particularly if you are interested in software quality assurance, software inspection, and software reading. This book has the most comprehensive material on software reading. You can find references to the original articles where particular reading techniques were first proposed, as well as information on the most recent developments and experiences with those reading techniques.

1.3 References

(Basili, 1987) V.R. Basili and R.W. Selby, Comparing the effectiveness of software testing strategies, IEEE Transactions on Software Engineering, vol.13, no,12m pp.1278-1296, 1987.

(Basili, 1997) V.R. Basili, Evolving and packaging reading technologies, Journal of Systems and Software, vol.38, no.1, pp.3-12, 1997.

(Briand, 1998) L. Briand, K. EI Emam, O. Laitenberger, and T. Fussbroich, Using simulation to build inspection efficiency benchmarks for development projects, Proceedings of the International Conference on Software Engineering, pp.340-349, 1998.

(Doolan, 1992) E. Doolan, Experience with Fagan's inspection method, Software Practice and Experience, vol.22, o.2, pp.173-182, 1992.

(Fagan, 1976) M.E. Fagan, Design and code inspections to reduce errors in program development, IBM Systems Journal, vol.15, no.3, pp.182-211, 1976.

(Gilb, 1993) T. Gilb and D. Graham, Software Inspection, Addison-Wesley, 1993.

(IEEE 1028) IEEE Std 1028-2008, IEEE Standard for Software Reviews and Audits, 2008.

(Johnson, 1998) P.M. Johnson and D. Tjahjono, Does every inspection really need a meeting? Empirical Software Engineering, vol.3, pp.9-35, 1998.

(Juristo, 2003) N. Juristo and S. Vegas, Functional testing, structural testing and code reading: what fault type do they each detect? Lecture Notes in Computer Science, vol.2765, pp.208-232, 2003.

(Laitenberger, 2002) O. Laitenberger, A survey of software inspection technologies, in Handbook on Software Engineering ad Knowledge Engineering, vol.2, pp.517-555, 2002.

(Mantyla, 2009) M.V. Mantyla and C. Lassenius, What types of defects are really discovered in code reviews? IEEE Transactions on Software Engineering, vol.35, no.3, pp.430-448, 2009.

(McConnell, 2004) S. McConnell, Code Complete, 2nd ed., Microsoft Press, 2004.

(Porter, 1996) A. Porter, H. Siy, and L. Votta, A review of software inspections, Advances in Computers, vol.42, pp.39-76, 1996.

(Russell, 1991) G.W. Russell, Experience with inspection in ultra large-scale developments, IEEE Software, vol.8, no.1, pp.25-31, 1991.

(Votta, 1993) L.G. Votta Jr., Does every inspection need a meeting? Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering, pp.107-114, 1993.

CHAPTER 2

Software Review Procedures

Although software peer review has been practiced for more than four decades, software literature uses inconsistent and in many cases conflicting terms to refer to more or less the same activities. According to IEEE Std-1028 (IEEE 1028), inspection is "A visual examination of a software product to detect and identify software anomalies, including errors and deviations from standards and specifications," and review is "A process or meeting during which a software product, set of software product, or a software process is presented to project personnel, managers, users, customers, user representatives, auditors or other interested parties for examination, comment or approval." The standard defines five types of reviews (management reviews, technical reviews, inspections, walkthroughs, and audits), and software inspections are a kind of review. We use review as a general term. This chapter describes a generic procedure for software review, then treats Fagan inspection and active design review in more detail. Factors impacting the effectiveness of software review are also discussed.

2.1 A Generic Software Review Procedure

All software review procedures share some commonalities. IEEE Std-1028 listed the following five steps for all the five types of reviews: (1) planning the review, (2) overview of the procedures, (3) preparation, (4) examination/evaluation/recording of results, and (5) rework/follow-up (IEEE 1028). Laitenberger used a reference model for software inspection processes, which has six phases: planning, overview, defect detection, defect collection, defect correction, and follow-up (Laitenberger, 2002). Tian abstracted a generic software review procedure with three steps, and various software review procedures can be considered as extensions to or specialization of this generic one (Tian, 2005). We discuss Tian's abstraction in this section.

The generic software review procedure has three stages of activities: planning and preparation, conducting the review, and corrections and follow-up, as illustrated in Figure 2-1.

Figure 2-1. *A generic software review procedure*

© Yang-Ming Zhu 2016

Y.-M. Zhu, *Software Reading Techniques*, DOI 10.1007/978-1-4842-2346-8_2

In the planning and preparation stage, one typically defines objectives of a review and decides what artifacts are subject to review, who will create them, who will review them and who else will be involved and in what capacity, when will the review happen, and what are the overall process and follow-up activities if needed. Before conducting a review, the document authors will assemble the material, decide the avenue for the review, and handle the review logistics, etc.

In the reviewing stage, people get together as a team face-to-face or on-line, synchronously or asynchronously. The team goes through the material under review in some pre-determined manner, discusses issues reported before the meeting or spotted in session, agrees on the observation or dismisses false positives. The focus of this stage is to uncover and collate issues in the document under review, and hence it is often called collection. In the end, the review team agrees whether a follow-up review session is warranted.

In the correction and follow-up stage, the author corrects issues that have surfaced during the review. The dispositions of the issues shall be agreed by the review team and the corrections or fixes shall be verified. A follow-up review can be conducted for that purpose, if the extent of changes is large; otherwise a lightweight follow-up shall suffice.

We discuss two classical software review procedures, namely Fagan inspection and active design review, which can be considered as an extension to this generic review. We use the term Fagan inspection for historical reasons.

Software review or inspection is independent of software development models, like waterfall or agile. Software review can be applied to a software artifact as soon as it is ready for review. Software review is considered a best practice and it is an important activity of software quality assurance (Tian, 2005).

2.2 Fagan Inspection and Extensions

The earliest and most influential software review procedure was proposed by Fagan in 1976 (Fagan, 1976). The method was initially intended for design and code inspection and later adapted to inspect virtually any software artifacts such as requirements, user documentation, and test plans and test cases as long as such artifacts can be made visible and readable (Fagan, 1986). Fagan inspection has been so influential that it is almost synonymous with the term inspection.

2.2.1 Fagan Inspection

Fagan inspection consists of six steps or operations as originally called: planning, overview, preparation, inspection, rework, and follow-up, as depicted in Figure 2-2. The planning step was not in Fagan's seminal paper (Fagan, 1976) but was added later (Fagan, 1986).

Figure 2-2. *Steps in Fagan inspection*

We discuss these six steps in the context of design and code inspection. The principal ideas behind Fagan inspection can be applied to inspecting any software artifacts.

- Planning: The objectives of the planning step are to define inspection entry criteria for the materials subject to inspect, to arrange the availability of the appropriate participants, and to arrange the meeting place and time.
- Overview: The objectives of the overview step are communication and education, as well as assigning the inspection roles to participants. This step involves the whole inspection team. Typically a meeting is held, during which the project overview and the specifics of the artifact to be inspected are given. The inspection materials are distributed at the end of the meeting.

- Preparation: The objective of the preparation step is for participants to study the material individually to fulfill their respective roles. One of the key ideas in the inspection is to assign different roles to the individual participants based on their respective expertise. The roles of the participants are discussed below. To facilitate the preparation, a checklist of recent error types can be used, or other kinds of reading techniques can be adopted. Software reading techniques are discussed in the rest of the book.
- Inspection: The objective of the inspection step is to find the errors in the material under inspection. A formal meeting is held and the entire team participates in the discussion. At the beginning of the meeting, if code files are under inspection, then the implementer (author) can show the implementation of the design. In the course of the meeting, errors are discussed; false positives are dismissed and true errors are recognized and noted, with possible error type classification and severity identification. It is important to note that the team should not hunt for solution nor discuss alternatives. After the inspection has been held, a written report of the findings is released in a timely manner.
- Rework: The objective of the rework step is to fix all errors or provide other responses. The author of the software artifact is responsible for the rework and responses.
- Follow-up: The objective of the follow-up step is to ensure all fixes are effective and there are no newly introduced problems. The moderator decides if another round of inspection is needed. For example, if the errors are minor and the changes are limited, then he can declare there is no need for another round of inspection. Regardless of whether there is another round of inspection, the team needs to pay attention to "bad fixes." Empirical data show that almost one of every six fixes are incorrect or create other defects (Fagan, 1986).

We can view Fagan inspection in the framework of the generic review. The first three steps of the Fagan inspection—planning, overview, and preparation—fit into the "planning and preparation" step in the generic review diagram; the inspection step of the Fagan inspection directly maps to the "review" block in the generic review; and the last two steps of the Fagan inspection—rework and follow-up—map to the "correction and follow-up" step in the generic review. Both the generic review and the Fagan inspection allow an optional iteration.

As mentioned above, the Fagan inspection defines the participant roles that each participant plays. There are four roles: moderator, author, reader, and tester. The moderator leads the inspection team and takes care of logistics; the other roles represent the viewpoints of those with their respective expertise during the inspection.

The moderator is the key person in a successful inspection. He or she possesses strong interpersonal and leadership skills, coaches and guides the inspection team, and handles meeting logistics, including scheduling the meeting and publishing the outcome of the inspection. The moderator must be neutral and objective. The author is

the person who created the software artifacts under inspection. The author is responsible for producing the artifacts and fixing the errors in the artifacts, with possible help from others. The reader is an experienced peer who can be a subject matter expert on the software artifact under inspection. The tester is responsible for writing and/or executing test cases for the software module or the product.

The Fagan inspection team typically consists of four people, large enough to allow group interaction to detect errors in software artifacts but small enough to allow individual voices to be heard. To have a healthy group dynamic, an ideal mix of participants can include people with different background and experience.

The Fagan-style reviews have a few noticeable drawbacks. One of them is the heavy process involved, which requires a series of formal meetings and documentations. This limitation is overcome by the introduction of modern lightweight reviews. The other drawback is that the quality of review varies widely, since the participants may be passively engaged with the review. This latter shortcoming is remediated by the active review, which is discussed in the next subsection.

2.2.2 Extensions to Fagan Inspection

It has been 40 years since Fagan published the Fagan inspection. Fagan inspection has been studied by researchers and embraced by practitioners (Aurum et al., 2002; Kollanus & Koskinen, 2009). The inspection procedure has been extended in different ways to further improve its efficiency or customize to unique situations. We summarize a few important extensions below.

2.2.2.1 Meeting or No Meeting

Proposed improvements to Fagan inspection often center on the importance and cost of group meetings, particularly the defect collection meeting. Fagan insisted on having a defect collection meeting, but other researchers questioned the importance of meetings. Reasons often cited to support a team meeting include:

- Synergy: More people working together will find more defects than they would working alone.
- Learning and knowledge sharing: Meetings are good opportunity for beginners to gain domain knowledge from experienced participants.
- Milestone: Meetings serve as project milestones.

However, it takes time and effect to schedule a group meeting, particularly when it involves many people. Researchers questioned whether the meeting creates synergy at all. Votta (1993) reported that most defects were found during the individual preparation stage, which was confirmed by many others, including Porter et al. (1995). Johnson and Tjahjono (1998) also reported that meeting-based reviews were significantly more costly than non-meeting-based reviews, and meeting-based reviews did not detect significantly more detects. However, meeting-based reviews were significantly better at reducing false positive defects, and reviewers preferred meeting-based reviews over non-meeting-based reviews.

The general consensus now is to not have a large group meeting or at least not emphasize it. A few alternatives are proposed to replace full-team meetings. As an example, a few experts can go through defects reported by individual reviewers during their preparation and decide the nature of defects (true vs false positives) (Sauer et al., 2000). Meeting-less inspections further evolved into modern lightweight inspections, which are discussed later.

2.2.2.2 What Is the Right Team Size?

Fagan suggested a team size of 4 people. A large team presumably allows a different kind of defects to be found since each reviewer has different expertise and experience. The argument of cost-effectiveness favors a smaller team. Bisant and Lyle (1989) proposed a two-person inspection involving just the author and reviewer, which makes the inspection accessible to teams or organizations that don't have access to larger team resources. Porter and colleagues (1997) conducted code inspection experiments in situ by varying the number of inspectors on each inspection team (1, 2, or 4) and concluded that, while 1 inspector was significantly less effective than 2 or 4 inspectors, there was little difference in the inspection effectiveness of 2 or 4 inspectors.

Instead of a single large team, Schneider et al. (1992) split a large team into N smaller teams for critical projects, let the N smaller teams to inspect requirements documents on parallel and independently, and aggregated defects from each smaller team at the end, which is known as N-fold inspection. They reported that independent teams found more defects than a single team.

Perhaps there is no optimal team size. The right team size will depend on the artifacts under review (types and complexity), organizational environment (whether it has access to large resources), etc. For important documents such as a requirements specification, more points of view are certainly beneficial. It is also a good idea to have more people to review the design than code. Complex artifacts also warrant more independent reviewers with different expertise.

2.2.2.3 Other Extensions

Gilb and Graham (1993) introduced a process brainstorming meeting right after the inspection meeting. This meeting's function is root cause analysis so that similar defects can be prevented from happening in future projects or activities in the same project.

Knight and Myers (1993) studied a phased inspection (for code inspections), which consists of multiple phases or mini-inspections, each focusing on detecting one class of defects such as issues with language, code layout, programming constructs, etc. Defects have to be fixed before the next phase can start.

Many software artifacts are generated in the course of project development. It might be infeasible to inspect all documents due to resource constraints. To address this concern, Thelin et al. (2004) developed a sampling-driven inspection, which utilizes a pre-inspection to identify a partial list of documents that can benefit from a focused inspection. The decision can also be based on the historical defect data and the characteristics of the document itself, e.g., code complexity metrics.

2.3 Active Design Review and Extensions

The active design review was introduced by Parnas and Weiss (1985). Although the publication came much later than Fagan's, according to Weiss, they conceived the idea independently about the same time as Fagan published his work.

2.3.1 Active Design Review

The purpose of design review is to find errors in design and its documentation. There are many kinds of design errors—e.g., inconsistency (different assumptions), inefficiency (inefficient to implement or use), ambiguity (allows different interpretation or lack of clarity), and inflexibility (does not accommodate change). Conventional reviews such as Fagan inspection tend to be incomplete and shallow. Parnas and Weiss noticed that reviews have variable quality and many factors contributed to it:

- The amount, quality, and time of delivery of the design documentation varied widely
- The time that the reviewers put in preparation varied widely
- The participation of the reviewers varied widely
- The expertise and roles of the reviewers varied widely

The active design review was proposed to reduce the variability and promote a consistent review quality. The key part of the active design review is the use of questionnaires to define the reviewer's responsibilities and to ensure they play a more active role. The main ideas behind the active design review, when compared to the Fagan-style review, include:

- The required knowledge and skills reviewers possess are explicitly identified before selecting the reviewers.
- Reviewers focus their efforts on the design aspects related to their experience and expertise.
- The designers pose questions to the reviewers rather than the reviewers asking questions. Each question is carefully designed such that its answer requires careful study of the design under review or some aspects of the design.
- Reviewers are actively involved in the review and make positive assertions on the design instead of merely skimming over the design for obvious or trivial errors.
- Reviewers and designers meet in a small group to resolves issues.

An active design review has five phases:

1. Make the design reviewable. A good design shall be well structured, simple, efficient, adequate, flexible, practical, implementable, and standardized. Design assumptions shall be made explicit. The design document can include redundant information for error and consistency checking. The document shall be structured in such a way that modules and submodules can be reviewed separately.
2. Identify the review types. A design review shall be focused and have a well-defined purpose. It is thus easy to identify expertise needed to support the review. Different reviews can concentrate on detecting different error types such as assumption validity, sufficiency, consistency between assumptions and functions, and adequacy.
3. Classify reviewers. Reviewers shall be specialists, potential users, those familiar with the design methods and technologies used, or those skilled at finding issues. The review shall exploit the skills and knowledge of the reviewers to detect as many errors as possible.
4. Design questionnaires. The questions shall not be trivial. They ensure the reviewers take an active role and use the design document to answer the questions. The questions shall be rephrased to avoid yes/no answers.
5. Conduct the review. There is no big meeting. Instead, the designer and reviewer have 1:1 or small group discussion. This phase has three stages:
    a. An overview meeting to discuss the material under review and how the process works.
    b. Assign reviewers to specific sections of the document, with a deadline of when the questions shall be answered and returned to the designer. The designer/reviewer meeting is also scheduled.
    c. The designer collects and reads the completed questionnaires and meets with reviewers individually to understand and resolve questions. The designer also updates the design document afterwards.

The active design review has a few challenges. It is usually hard to find subject matter experts to serve as reviewers and get their time commitment, as everyone has a busy schedule and other commitments in the same or different projects. There is no big meeting, and review is managed as individual tasks, thus it takes diligence to keep the review on track and complete on time. Lastly, it takes significant effort to design a set of questions whose answers are not obvious and easy to find. The questions for the reviewers to answer shall be carefully designed and non-trivial, which forces reviewers to play the roles of users of the design and write a program to implement the design. Answering the questions makes the reviewer active. The questions shall also be tailored to the reviewer's expertise and the aspects of the document under review.

2.3.2 Extensions to the Active Design Review

Britcher (1988) took the active design review one step further by incorporating correctness arguments into inspection. The artifact author and inspectors collaborate in the pursuit of correctness, developing questions and answers together centering on four program attributes: topology, algebra, invariance, and robustness. The purpose for this extension is not to improve inspections but to improve programming.

The active design review was espoused the architecture tradeoff analysis method and the new method is called active review for intermediate designs (Clements, 2000). The hybrid method fills a niche that only a portion of a system, not a complete architecture, is available, but the designer would like to get early feedback on the design approach.

The ideas behind the active design review have been used in scenario-based reading, as we will discuss in Chapter 4.

2.4 Other Types of Reviews

In addition to the Fagan Inspection and active design review and their extensions, many different kinds of software review exist in practice with different levels of discipline and flexibility. There are, however, no commonly agreed-upon definitions of terms. The usages are generally inconsistent, which causes confusions in many cases. Even for the same type of reviews, the levels of procedural details can vary significantly. Fagan, in his seminal paper, recognized that walkthroughs were practiced in many different places with varying regularity and thoroughness (Fagan, 1976). It was also reported that one person's walkthrough can be another person's inspection (Wiegers, 2001).

Wiegers (2001) organized different types of reviews based on their levels of formality and rigor. He discussed, in the decreasing order of formality, inspection, team review, walkthrough, pair programming, peer desk-check or pass-around, and ad hoc review. Cohen et al. (2006) discussed five types of review: formal inspections, over-the-shoulder reviews (screen or desk reviews), e-mail pass-around reviews, pair programming, and tool-assisted reviews. IEEE Standard for Software Reviews and Audits defined five types of software reviews (management reviews, technical reviews, inspections, walkthroughs, and software audits) (IEEE 1028). In terms of formality, walkthroughs are least formal, inspections less formal, followed by management and technical reviews, with audits being most formal. These types of reviews and audits are considered systematic, with the following attributes: team participation, documented results of the review, and documented procedures for conducting the review.

The review or inspection procedures mentioned above are applicable to any software artifacts, with the exception of pair programming, which is mostly applicable to source code. There are specific review procedures for particular kinds of software artifacts as well. Given the importance of software architecture in software life cycle, architecture review and evaluation methods were an active area of research and now have become mature. We use techniques (or procedures) and methods as synonyms, although some authors distinguish them deliberately (Basili, 1997). The ways to assess software architectures are packaged as "methods" in literature. Interested readers can find more information in books on architecture evaluation such as Clements et al. (2002).

We do not intend to define various types of reviews, rather suggest adopting the definitions in the IEEE standard. We do want to point out the trend in industry practices. A lightweight process is strongly favored by practitioners due to their busy schedule and heavy workload, and therefore the synchronous, real-time, face-to-face meeting in Fagan inspection can be impractical. The synergy of the defect collection meeting has been questioned and the community generally agrees that its value is marginal. Thus software reviews can be conducted without meetings. The IEEE standard permits reviews to be held without physical meeting in a single location. Given the advancement of telecommunication and telepresence, physical meetings can be replaced with telephone conference, video conference, web conference, or other groupware and group electronic communications.

Software tools to support software reviews also have made progress. Web-based tools allow authors to upload software artifacts to web servers, invite reviewers, and set up a review online. Reviewers can enter review comments online and see each other's comments instantly. Issues raised during review are tracked by tools for closure. Synchronous and asynchronous notifications keep the author and reviewers in the same loop on progress. Review metrics are automatically collected to provide input for future process improvement. With these features and capabilities, the overhead of classic inspections is alleviated.

At least for code review, many organizations, including Facebook, Google, and Microsoft, are adopting lightweight, tool-assisted reviews (Bacchelli & Bird, 2013). Bacchelli and Bird called this modern code review. Referring to Fagan's inspection, only preparation, collection (inspection), and rework stages are present in the lightweight review, and defect collection is facilitated with tools. There is empirical evidence that support the efficiency and effectiveness of the lightweight, tool-assisted modern practice (Bacchelli & Bird, 2013; Cohen et al., 2006).

2.5 Factors Impacting Software Reviews

When the purpose of software reviews is to detect defects in the software artifacts, one is interested in how many defects are detected and how quickly they are detected. The number of detected defects is related to the effectiveness of software reviews. The more defects are detected, the more effective a software review is. How quickly defects are detected (number of detected defects per unit of time) is related to the efficiency and cost of software reviews. The quicker the defects are detected, the lower the cost of software review. Many factors impact the effectiveness and efficiency of software reviews, and individual performance, meetings, preparation, the amount of inspected materials, team size, tools, and training are frequently mentioned factors. Meetings and team sizes have been discussed earlier. Both commercial and research tools exist to support software inspection, mostly in the areas of asynchronous communication, artifact comprehension and visualization, and defect tracking. We discuss the remaining factors here.

Individual performance. Large difference (more than 10 times) in individual performance in software engineering was observed a long time ago (Boehm, 1981). Hatton (2008) recently reported the same in code inspection. A few attributes contributed to the difference, including expertise on the programming language (Knight, 1993) and software reading expertise (Sauer, 2000). The rest of the book is devoted to software reading techniques.

Preparation. The importance of individual preparation to defect detection was reported by many authors—e.g., Gilb (1993). The more time a reviewer spent on preparation, the more defects are typically reported. It is generally agreed that most defects are found during individual's preparation phase before defects are aggregated.

Amount of materials. Due to people's limited attention span, when the amount of materials to be inspected is large, readers become overwhelmed and fatigued, which negatively impacts the review effectiveness. For code inspection, the rate of inspection often suggested in literature is 100-200 lines of code per hour. If readers are not given enough time to examine the material, then the inspection loses its rigor and readers tend to report trivial findings.

Training. The level of training readers receive impacts the review effectiveness. This should not be a surprise, given the importance of individual's skills in inspection. Researchers found that practical training on defect finding skills was more impactful than the training on process.

Software development is a highly personal endeavor. Without care, software review or inspection can easily cause anxiety and tension among participants. The social psychological effects of computer programming were recognized from the early days of programming (Weinberg, 1998). Weinberg published his famous book, The Psychology of Computer Programming, in 1971, before Fagan inspection was introduced and practiced. Authors should have a thick skin and have their egos checked when participating in the reviews. Reviewers should recognize the IKEA effect on the artifact authors. The IKEA effect is a cognitive bias in which the artifact creators place a disproportionately higher value on the artifact they created (Norton et al., 2012). More labors lead to deeper affection. Successful inspection depends on the individual capability of each team member and how well individuals work in teams.

Before we close this section, let's discuss the eight maxims Kelly and Shepard (2004) compiled based on their observations, which focus on people forces at work in inspections.

1. Use structured inspection techniques. By inspection techniques, Kelly and Shepard meant reading techniques. The techniques shall be appropriate for the inspection goals and inspectors' experience.

2. Set standards of acceptability. Inspections are expensive. There should be entrance criteria to start the inspection. Artifacts shall be cleaned up and superficial issues shall be rooted out, preferably with tools, before inspections start.

3. Match skills to tasks. If skills and tasks are matched, both effectiveness and the comfort level of the inspector improve. Inspectors who are familiar with the artifacts under review shall be chosen first. Skills here also include soft skills such as verbal and written communication skills, which are needed to interact with artifact authors.

4. Find physical, mental, and schedule space. Inspections are mentally demanding and require concentration for an extended period of time. Inspection shall be conducted at a quiet place without interruption, and inspectors shall be given enough time to complete the inspection.

5. Encourage an inspection-based process. At minimum, the inspection shall be planned, and the project schedule shall reflect that. Roles and responsibilities shall be clearly specified ahead of time.

6. Promote responsibility, ownership, and authority. Responsibility and ownership lead to improved inspection efforts. Inspectors and authors jointly own the artifact and are responsible to its quality. Inspectors shall be granted the authority to access needed documents and resources to complete the inspection.

7. Ensure clear inspection goals are set. Clear goals affect the scope of the inspection. It shall be clear to all participants if alternative solutions shall be proposed or not. Also ensure terminologies are defined and used consistently to avoid potential confusion.

8. Use metrics cautiously. Metrics can be used and interpreted in different ways. There is no commonly agreed-upon definition of "defect," nor its granularity and severity. The number of defects is strongly related to the complexity of the task itself. Metrics shall not be used to evaluate an individual's job performance.

2.6 Summary

Software review and inspection is considered a best practice and has been practiced over four decades. There are many flavors of reviews and inspections. We focused on the classic Fagan inspection and active design review, as well as their various extensions. Fagan inspection is not only the first formal inspection method but also the foundation of many other modified versions. The ideas of the active design review are used in many software reading techniques, as we shall see in later chapters.

Laitenberger and DeBaud (2000) contextualized software inspection in the software lifecycle. They provided suggestions on how Fagan inspection and its various extensions can be customized to inspect different software artifacts created during the software lifecycle, with different reading techniques. IEEE Std 1028 has a comprehensive list of software artifacts that can be subject to review or inspection (IEEE 1028). Many factors impact the performance of software reviews and inspections, chief among which is individual factor. It is well established that software review and inspection is primarily an individual, not a group, activity. How the reviewer reads and extracts information from the software artifact under review impacts his or her performance. The rest of the book focuses on reading techniques, which are meant to equip the readers and enhance their review and inspection capacities.

2.7 References

(Aurum, 2002) A. Aurum, H. Petersson, and C. Wohlin, State-of-the-art: software inspections after 25 years, Software Testing, Verification and Reliability, vol.12, pp.133-154, 2002.

(Bacchelli, 2013) A. Bacchelli and C. Bird, Expectations, outcomes, and challenges of modern code review, Proceedings of the 35th IEEE/ACM International Conference on Software Engineering, pp.712-721, 2013.

(Basili, 1997) V.R. Basili, Evolving and packaging reading technologies, Journal of Systems and Software, vol.38, no.1, pp.3-12, 1997.

(Bisant, 1989) D.B. Bisant and J.R. Lyle, A two-person inspection method to improve programming productivity, IEEE Transactions on Software Engineering, vol.15, no.10, pp.1294-1304, 1989.

(Boehm, 1981) B.W. Boehm, Software Engineering Economics, Prentice-Hall, 1981.

(Britcher, 1988) R.N. Britcher, Using inspections to investigate program correctness, IEEE Computer, vol.21, no.11, pp.38-44, 1988.

(Clements, 2000) P.C. Clements, Active Reviews for Intermediate Designs, Technical Note, CMU/SEI-2000-TN-009, 2000.

(Clements, 2002) P. Clements, R. Kazman, and M. Klein, Evaluating Software Architectures: Methods and Case Studies, Addison-Wesley, 2002.

(Cohen, 2006) J. Cohen, S. Teleki, and E. Brown, Best Kept Secrets of Peer Code Review, Smart Bear Inc., 2006.

(Fagan, 1976) M.E. Fagan, Design and code inspections to reduce errors in program development, IBM Systems Journal, vol.15, no.3, pp.182-211, 1976.

(Fagan, 1986) M.E. Fagan, Advances in software inspections, IEEE Transactions on Software Engineering, vol.12, no.7, pp.744-751, July 1986.

(Gilb, 1993) T. Gilb and D. Graham, Software Inspection, Addison-Wesley, 1993.

(Hatton, 2008) L. Hatton, Testing the value of checklists in code inspections, IEEE Software, vol.25, no.4, pp.82-88, 2008.

(IEEE 1028) IEEE Std 1028-2008, IEEE Standard for Software Reviews and Audits, 2008.

(Johnson, 1998) P.M. Johnson and D. Tjahjono, Does every inspection really need a meeting? Empirical Software Engineering, vol.3, pp.9-35, 1998.

(Kelly, 2004) D. Kelly and T. Shepard, Eight maxims for software inspectors, Software Testing, Verification and Reliability, vol.14, pp.243-256, 2004.

(Knight, 1993) J.C. Knight and E.A. Myers, An improved inspection technique, Communications of the ACM, vol.36, no.11, pp.51-61, 1993.

(Kollanus, 2009) S. Kollanus and J. Koskinen, Survey of software inspection research, The Open Software Engineering Journal, vol.3, pp.15-34, 2009.

(Laitenberger, 2000) O. Laitenberger and J.M. DeBaud, An encompassing life cycle centric survey of software inspection, Journal of Systems and Software, vol.50, no.1, pp.5-31, 2000.

(Laitenberger, 2002) O. Laitenberger, A survey of software inspection technologies, in Handbook on Software Engineering and Knowledge Engineering, vol.2 Emerging Technologies, World Scientific, 2002.

(Norton, 2012) M.I. Norton, D. Mochon, and D. Ariely, The IKEA effect: When labor leads to love, Journal of Consumer Psychology, vol.22, no.3, pp.453-460, 2012.

(Parnas, 1985) D.L. Parnas and D.M. Weiss, Active design reviews: principles and practices, 8th International conference on Software Engineering, pp.215-222, 1985.

(Porter, 1995) A.A. Porter, L.G. Votta, and V.R. Basili, Comparing detection methods for software requirements inspection: A replicated experiment, IEEE Transactions on Software Engineering, vol.21, no.6, pp.563-575, 1995.

(Porter, 1997) A.A. Porter, H.P. Siy, C.A. Toman, and L.G. Votta, An experiment to assess the cost-benefit of code inspections in large scale software development, IEEE Transactions on Software Engineering, vol.23, no.6, pp.329-346, 1997.

(Sauer, 2000) C. Sauer, D.R. Jeffery, L. Land, and P. Yetton, The effectiveness of software development technical reviews: A behaviorally motivated program of research, IEEE Transactions on Software Engineering, vol.26, no.1, pp.1-14, 2000.

(Schneider, 1992) M. Schneider, J. Martin, and W. Tsai, An experimental study of fault detection in user requirements documents, ACM Transactions on Software Engineering and Methodology, vol.1, no.2, pp.188-204, 1992.

(Thelin, 2004) T. Thelin, H. Petersson, P. Runeson, and C. Wohlin, Applying sampling to improve software inspections, Journal of Systems and Software, vol.73, no.2, pp.257-269, 2004.

(Tian, 2005) J. Tian, Software Quality Engineering: testing, quality assurance, and quantifiable improvement, IEEE Computer Society Press, 2005.

(Votta, 1993) L.G. Votta Jr., Does every inspection need a meeting? Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering, pp.107-114, 1993.

(Weinberg, 1998) G.M. Weinberg, The Psychology of Computer Programming: Silver Anniversary Edition, Dorset House, 1998.

(Wiegers, 2001) K.E. Wiegers, Peer Reviews in Software: A Practical Guide, Addison-Wesley Professional, 2001.

CHAPTER 3

Basic Software Reading Techniques

As software professionals we were all trained to create software artifacts. For example, requirements engineers were taught how to define, document, and maintain requirements; software designers were trained to conceptualize the system, come up with architecture to deliver ever-demanding user experience, and document the architectural design; software developers were trained to write programs in programming languages using various constructs and concepts. However, we were not taught how to read those artifacts and source code authored by others.

There is ample empirical evidence that the software review is primarily an individual effort and many anomalies are uncovered in the course of individual reading (Johnson & Tjahjono, 1998; Votta, 1993). Thus the kinds of reading techniques the individual uses are paramount to the outcome and effectiveness of software review. This chapter defines what software reading is and surveys the basic software reading techniques.

3.1 Introduction to Software Reading

Everyone who received primary education knows how to read. However, the effectiveness of reading varies. Wikipedia defines reading as (https://en.wikipedia.org/wiki/Reading_(process), accessed on Dec. 15, 2015. http://www.encyclopedia.com/topic/reading.aspx essentially contains the same elements as Wikipedia summarizes)

> *Reading is a complex cognitive process of decoding symbols in order to construct or derive meaning. … [I]t is a complex interaction between the text and the reader which is shaped by the reader's prior knowledge, experiences, attitude, … The reading process requires continuous practice, development, and refinement. In addition, reading requires creativity and critical analysis. … Readers integrate the words they have read into their existing framework of knowledge and schema.*

Based on this discussion, we conclude that (1) reading has a purpose, (2) the reader's background plays a role in reading, (3) reading techniques can be learned, and (4) reading requires critical analysis. Those general characteristics exhibit in software reading as well.

© Yang-Ming Zhu 2016

Y.-M. Zhu, *Software Reading Techniques*, DOI 10.1007/978-1-4842-2346-8_3

3.1.1 Definition of Software Reading

Based on the Encyclopedia of Software Engineering (Shull, 2002), software reading is defined as the process by which a developer gains an understanding of the information encoded in a work product sufficient to accomplish a particular task. We adopt the same definition. The "work product" refers to the software artifact, ranging from requirements specification, design documentation, code files, test plan, test cases, test reports, to user documentation, etc. The "particular task" is related to the purpose of reading, whether the reading is for gaining knowledge of the system, detecting defects, or implementing the design. The purposes of reading are systematically treated in the next subsection.

Associated with software reading is the software reading technique. Again, based on the Encyclopedia of Software Engineering (Shull, 2002), a software reading technique is a series of steps for the individual analysis of a textual software product to achieve the understanding needed for a particular task. A series of steps is a set of instructions that guide the reader how to read the artifacts, what areas to focus on, and what problems to look for. Again software reading is primarily an individual activity.

Ever since the Fagan inspection was introduced, it was recognized that some sort of reading technique was needed. Fagan suggested using checklists (Fagan, 1976). By now there is a big body of knowledge on software reading techniques. The software reading technique has become an active research area since 1994 after Porter and Votta (1994) published their paper on scenario-based reading. Between 1980 and 2008, 16% of software inspection research papers were on software reading techniques (Kollanus & Koskinen, 2009). This chapter surveys a few early and popular reading techniques. Latter chapters discuss more advanced or specialized reading techniques.

3.1.2 Purposes of Software Reading

We read software artifacts to accomplish a particular task. The task is defined by the purpose of reading. Broadly speaking, we read software for analysis and for construction (Basili et al., 1996).

In reading for analysis, we read and understand the document, then analyze and assess the qualities and characteristics of the document. The primary objective of reading for analysis is to detect defects in the document. When reading the requirements specifications, we may detect various types of requirement errors such as incorrect facts, omission, ambiguity, and inconsistency. When reading the code, we may detect various types of coding errors such as logic errors, assumption errors, incorrect function calls, etc. Other objectives of reading for analysis include performance predictions, requirement tracing, usability assessment, etc. One of the main reasons to have an architecture document is to support analysis and prediction (Clements et al., 2011). For example, we can predict system performance by using queue theory when there are one or multiple processing queues involved. By reading the system requirements specification and subsystem requirements specification, we can reason whether the system requirements specification is sufficiently decomposed to the subsystem requirements specification, and whether the subsystem requirements specification can indeed trace back to its system requirements specification origin and have a sufficient coverage of the system requirements specification.

In reading for construction, we attempt to identify if any requirements, design, code or test cases can be reused in the same project or in a different project. We also examine the high-level design document to come up with the low-level design or read the design document to see how we may implement the design properly. Sometimes we read the software just to gain the knowledge. It might be true, however, we have an interest to maintain the software in future.

Reading for defect detection is the focus of most researchers and is also the main concern of this book.

3.1.3 Taxonomy of Software Reading Techniques

Software reading techniques can be classified along different dimensions. Based on whether the reading technique is structured (systematic) or not, the reading techniques can be put into the following categories:

- Unstructured, or unsystematic reading: Ad hoc reading falls into this category. Ad hoc reading is discussed in this chapter.
- Semi-structured reading: Checklist-based reading is in this group and is also discussed in detail in this chapter.
- Structured or systematic reading: Perspective-based reading falls into this group, along with many other techniques. These reading techniques collect knowledge about the best practices for defect detection into a single procedure. To some extent, they serve a similar role as design patterns to design. These techniques are covered in other chapters of this book.

When we discuss the benefits and shortcomings of each reading technique, keep in mind that structures of any kind simultaneously enable and limit human activities. This is known as the Paradox of Structure (Jablokow, 2003): While a structured reading technique enables one to find anomalies in the software artifact, the same reading technique may limit one to find anomalies only in certain categories.

Reading techniques can be classified according to the software artifacts to which they are applied. For example, checklist-based reading can be applied to review almost every software artifact, while stepwise abstraction, which is discussed in a later chapter, is only applicable to code review.

3.2 Ad hoc Reading

When there is no specific method provided to the reader to detect issues or defects in the software artifacts under review, we call it an ad hoc reading. This is an unstructured, or unsystematic, reading technique. The reader simply attempts to uncover as many issues and defects as possible by examining the artifact using whatever intuition, skills, knowledge, and experience he or she may have. The effectiveness of the ad hoc reading is entirely up to the individual reader. The individual defect detection performance can vary by a factor of 10 in terms of defects found per unit time (Hatton, 2008). Nevertheless, it is the most common reading technique.

One of the advantages of the ad hoc reading is that there is no training needed for the reader. However, it has many disadvantages, among which is the wide variability of the results. In fact, the outcome to a large extent depends on the skills, knowledge, and experience of the reader. It is slow for readers to acquire expertise; thus inexperienced readers will not be productive when reviewing software artifacts to uncover issues. Once the expertise has been acquired, it is very difficult to teach or transfer the expertise to others. Since the effectiveness of the ad hoc reading depends on individual expertise, readers adopting this reading technique may miss major areas of concern.

3.3 Checklist-Based Reading

Checklists are ubiquitous. You may use a shopping list. Grade school kids often have a school supply list for the next school year. If you are planning a family vacation, you most likely have a packing list. The example list goes on and on.

Checklists are widely used in software reading and they serve a similar purpose as the above-mentioned checklists do. In fact, Fagan (1976) suggested using a checklist during software inspection in his seminal paper. Since then checklists for software review have flourished. In the 1999 survey, Brykczynski (1999) reviewed 117 checklists. And checklists are continuously being proposed for software reading.

3.3.1 Checklist Definition, Types, and Examples

So what is a checklist? A checklist is a list of questions to provide reviewers with hints and recommendations for finding defects during the examination of software artifacts. Since a question can be rephrased as an imperative sentence, the checklist does not have to be composed of questions only. The questions or imperative sentences in the checklist draw reviewers' attention to defect-prone areas based on historical data. A checklist may also serve other purposes. For example, it can be used to ensure important areas are covered by the artifact under review.

Checklists can be classified into two groups (Tian, 2005): property-based checklists and artifact-based checklists. Checklists for coding standards and guidelines, standard or process conformance, etc. are property-based. Checklists for requirements specifications, design documents, code files, or test cases are artifact-based. Checklists for requirements specifications and design documents may contain items to ensure correctness, consistency, and completeness of the requirements or design, while the checklists for code review may include items for generally accepted programming practices and for particular programming languages.

Panel 3-1 shows a partial checklist for a requirements review. Note each checklist item is an imperative sentence. When a reader is reviewing a requirements specification document, he or she can check each requirement against individual items on this checklist.

PANEL 3-1: A SAMPLE (PARTIAL) CHECKLIST FOR REQUIREMENTS REVIEW

1. Requirements specifications shall be testable.
2. Requirements specifications shall not conflict with other requirements specifications.
3. Conditional requirements specifications shall cover all cases.
4. Numerical values in requirements specifications shall include physical units if applicable.

Panel 3-2 shows a sample checklist for code review. Here each checklist item is a question. The checklist for code review tends to be big and the items can be grouped into different areas of concern such as control, input/output, performance, etc. For object-oriented code reading, the areas of concern can be aligned with the features of object-oriented programming, such as encapsulation, inheritance, and polymorphism.

PANEL 3-2: A SAMPLE (PARTIAL) CHECKLIST FOR CODE REVIEW

1. Have resources (e.g., memory, file descriptor, database connection) been properly freed?
2. Are shared variables protected/thread-safe?
3. Is logging implemented?
4. Are comments updated and consistent with the code?
5. Is data unnecessarily copied, saved, or reloaded?
6. Is the number of cores checked before spawning threads?

Checklists are typically developed based on the analysis of past team defects in the same or different projects. They can also be based on others' experience, but customized for one's project team. Checklists can be tailored to an individual as well. Individuals can have a personal defect checklist that compiles the problematic areas in which the individual tends to make mistakes. The Personal Software Process prescribes the use of such a personal checklist (Humphrey, 1997).

Compared to ad hoc reading, checklist-based reading reduces the variability of reading results, i.e., the results are less dependent on the reviewers' skills, knowledge, and experience. It also ensures coverage of important areas and is thus effective at detecting omissions. As recent research suggests, in addition to supporting defect detection, checklist-based reading improves software understanding and comprehension, which makes the subsequent software modification easier (McMeekin et al., 2008).

As the Paradox of Structure suggests, checklist-based reading might detect the defects of particular types covered by the checklist, i.e., those previously encountered from which the checklist was created. Therefore insidious defects, which require deep understanding of the artifacts, are often missed. The other disadvantages of checklist-based reading are related to the checklist itself. The checklist often includes generic items that may not be applicable to the project or the artifact. A lengthy checklist may overwhelm readers. The "best practices of checklists" discussed later can remediate the issues with generic and lengthy checklists.

3.3.2 Checklists with Guidance

Checklist-based software reading is considered semi-structured, as it does not tell the reader how to use the checklist and there is little verification that the reader actually conducts an analysis relating to checklist items. To remedy this shortcoming, an active guidance can be added to the traditional checklist-based reading (Winkler et al., 2005).

Winkler et al. focused on design document inspection. The readers are given a tailored checklist that provides active guidance. The specific checklist leads the reader through the inspection process. The active guidance to be used with a checklist is shown in Panel 3-3 (adapted from Winkler et al. [2005]).

PANEL 3-3: AN ACTIVE GUIDANCE USED WITH CHECKLIST-BASED READING

1. Analyze requirements and system functions in the requirements document.
2. Prioritize the correlations between requirements and system function according to the reader's own knowledge of the application domain.
3. Trace the requirements and functions in the design document according to their priorities or importance.
4. Report any differences as defects.
5. Pick the next most important requirement and repeat steps 3 and 4, until done.

This checklist with active guidance promotes a deep understanding of the specification document, the system requirements, and system functions, which enables the readers to uncover more defects in the design document. It also allows the reader to focus on more important requirements due to the prioritization performed before the start of the inspection. Thus it uncovers crucial defects.

Alternatively, guidance on how to use a checklist can be implicitly built into the checklist itself. Dunsmore et al. (2002) compiled a checklist for object-oriented code reading. They took into account the structure of object-oriented code and deliberately ordered the checklist questions in such a way as to support readers in building up a deep understanding of the code under review. Their checklist has three ordered sections. Each

section has a number of ordered groups, and each group has a list of ordered questions. Their checklist is adapted in Table 3-1. It has three sections: the class section, the method section, and the final section. The class section is about inheritance and constructor issues. The method section is concerned with issues related to methods, such as data and object referencing, messaging, and method behavior. The final section is on method overriding.

Table 3-1. *A Checklist for Object-Oriented Code Reading (adapted from Dunsmore et al. [2002])*

Section	Feature	Checklist item
For each class	Inheritance	Q1. Is all inheritance required by the design implemented in the class?
		Q2. Is the inheritance appropriate?
	Constructor	Q3. Are all instance variables initialized with meaningful values?
		Q4. If a call to base class is required in the constructor, is it present?
For each method	Data referencing	Q5. Are all parameters used within a method?
		Q6. Are the correct class constants used?
		Q7. Are indices of data structures operating within the correct boundaries?
	Object messaging	Q8. Is the correct method being called on the correct object?
		Q9. Are the correct values passed as parameters in the correct order?
	Object referencing	Q10. Should a reference to an object be used instead of a distinct copy?
	Selection and iteration	Q11. Are all relational and logical operators sufficient and correct?
		Q12. Is the correct sequence of code executed for any conditional outcome?
		Q13. Is the use of an iterator or loop appropriate when destructive operations are occurring on a collection?
	Method behavior	Q14. Are all assignment and state changes made correctly?
		Q15. For each return statement, is the value returned and its type correct?
		Q16. Does the method match the specification?
For each class	Method overriding	Q17. If inherited methods need to behave differently, are they overridden?
		Q18. Are all uses of method overriding correct?

The purposely structured checklist accomplishes two objectives: "where to look" and "how to detect." "Where to look" is related to the potential problem areas and is supported by the list of sections and features. The "how to detect" part is supported by the checklist questions. As the readers navigate through the different groups of questions, they successively move from a high-level view to a more detailed level view. At the end, the readers can readily conclude whether the method implementation code matches the specification (Q16).

3.3.3 Best Practices of Checklists

Over the years, people have come up with a few heuristics on what makes good checklists and what to avoid in checklists (Brykczynski, 1999). We call this general advice "the best practices of checklists":

- Checklists should be periodically revised based on historical data to include new items and remove outdated items. If the checklists are updated regularly, the reviewers may be more likely to read and use the checklists. If the checklists are updated to reflect the most common issues, more likely it will help reviewers in finding defects.
- Checklists should be concise and fit on one page. A reviewer is less likely to flip through multiple pages. The single-paged checklist can be hung on the office wall or put on the desk close to where the reviewer is examining the software artifact.
- Checklist items should not be too general. A general item is hard to apply or subject to varying interpretation.
- Checklist items should not be used for conventions which are better checked or enforced with software tools.

3.3.4 Empirical Experiences

Checklists have been used in software reading since Fagan (1976) reported the Fagan Inspection. They were advocated by many authors, e.g., Gilb and Graham (1993) and Clements et al. (2011). Checklist-based reading as well as ad hoc reading is frequently chosen as a baseline, and other newly proposed reading techniques are compared to them. Later chapters include many examples. Here we selectively discuss the results of a few empirical studies.

Winkler et al. (2005) compared the traditional checklist-based reading and checklist-based reading with guidance. Their experimental subjects were software engineering students. Defects were seeded into the design document, with three levels of severity: critical, major, and minor. They reported that checklist-based reading with guidance has a slightly longer preparation time but a shorter inspection time. When both preparation and inspection times were considered, the total times were not significantly different (although the total time of the checklist with guidance was slightly shorter). The difference was notable in the number of uncovered defects. The checklist with guidance technique uncovered more defects in all three levels of severity. It also registered less false positive defects.

Dunsmore et al. (2002) compared the deliberately designed checklist for object-oriented code reading with a few other reading techniques. In short, they found that the checklist-based reading is the most effective reading technique for object-oriented systems. The details can be found in Chapter 7, Code Reading Techniques.

Rong and colleagues (2014) reported a case study of code reading techniques in a small-sized software company. Novice software inspectors used both ad hoc reading and checklist-based reading to read source code of 20 modules. They observed that readers using the checklist tended to have a slower reading speed but discovered more defects.

The content of the checklists is accessible to both the authors of the artifact and the readers. Thus the authors can use the checklists to uncover the defects themselves before sending the artifact out for review. All the empirical studies mentioned previously ignored this fact when the checklists were used in practice. The effectiveness of checklist-based reading largely depends on the content of the checklists.

3.4 Differential Reading

Many software reading techniques assume that the reader will read the entire document, be it a requirements specification, design document, source code file, or test case. As a matter of fact, there are a few situations where developers typically deal with the difference between the existing software artifact and the one that is being modified:

- Software applications are frequently released incrementally via different projects. New features and defect fixes are added in a latter release. Many software artifacts including requirements, design, code implementation, and testing can be reused. New requirements, design, code, and test cases are typically embedded in the existing documents.
- Even for software applications that are started from scratch, an iterative and incremental development process may be adopted. New features and defect fixes are implemented in a latter iteration or sprint. Along the way, documentations are also written, revised, expanded, and reviewed incrementally.
- Regardless of how the project is structured and what development process is used, a software artifact is reviewed, a rework might be required, and the updated document might be subject to re-review again.

In all those situations, there are at least two versions of the software artifacts available. It is not worthwhile for the reader to read the entire document from beginning to end each time, particularly if the reader is already familiar with the previous versions.

There is no published reading technique to deal with the situations above. We adopt the instructions in Panel 3-4, so that the reader can focus on the changes and assess whether those changes meet the intentions without negative side effects. We call it differential reading, as it draws readers' attention to the part of changes and focuses the changes in the context. Don't be deceived by the amount of changes, however. For source code, a simple innocent change may have significant ramifications. Mernhart et al. (2010) reported their positive experience on a continuous differential-based method and tool for code reviews in agile software development.

There are many tools to highlight the changes in a document. Microsoft Word is frequently used to capture requirements and describe designs. To keep track of changes across different revisions of the document, one can enable "Track Changes". There are many tools to track the source code changes. For example, the Subversion client TortoiseSVN is integrated with the Windows Explore and the Diff command from the context menu can highlight the code change against the repository.

We illustrate one differential reading example below. The C# code reads in a script file, replaces some strings with other strings in each line, and writes the replaced script to a new file. The strings and the strings to replace them are predefined and read in from a file. However, it was noticed recently that some strings were replaced incorrectly. This happened when the prefix of a long string was mistakenly replaced first. The source code was modified in three places to fix the problem. The first place is simply to define a string list to store the sortedKeys (code not shown). Figure 3-1 shows the second section of code where the change happened. The top shows the old file and the bottom displays the new file. New code was added in the new file from line 52 to line 55. The code simply gets the list of strings and sorts them in reverse order. The comment explains why it is done.

PANEL 3-4: INSTRUCTIONS FOR DIFFERENTIAL READING

1. Get familiar with the existing software artifact if not already.
2. Understand what drives the modification of the existing document, be it new features, defect fixing, or some other nature.
3. Use a diff tool to highlight what has been changed in the newly updated document.
    a. Pick a block of changes to focus on and read the surrounding text where the changes are part of.
    b. Pay attention to all change types: addition, deletion, and modification.
    c. If the amount of change is significant, consider it new and use any reading techniques available to you or agreed upon by the team.
    d. If the amount of change is not significant:
        i. Check whether the change is consistent with the change driver, the assumptions or styles the document already took, etc. If not, log as a defect.
        ii. Check if there are any side effects. If there is a side effect, log as a defect.
4. Repeat step 3 until all changes have been read and analyzed.

C:\yzhu\SoftwareReading\codes\Form1_old.cs

2/18/2015 11:25:32 PM 6,286 bytes C,C++,C#,ObjC Source UTF-8 BOM PC

```
41  try {
42      StreamReader sr = new StreamReader(keyFile);
43      string line = sr.ReadLine();
44      while (line != null) {
45          string[] tokens = line.Split('\t');
46          keyMapping[tokens[1].Trim()] = tokens[0].Trim();
47          line = sr.ReadLine();
48      }
49      sr.Close();

50  } catch (Exception ex) {
```

C:\yzhu\SoftwareReading\codes\Form1_new.cs

5/26/2016 9:20:57 AM 6,587 bytes C,C++,C#,ObjC Source UTF-8 BOM PC

```
42  try {
43      StreamReader sr = new StreamReader(keyFile);
44      string line = sr.ReadLine();
45      while (line != null) {
46          string[] tokens = line.Split('\t');
47          keyMapping[tokens[1].Trim()] = tokens[0].Trim();
48          line = sr.ReadLine();
49      }
50      sr.Close();
51
52      // sort keys in reverse order so we replace longest keys (not prefix) first
53      sortedKeys = keyMapping.Keys.ToList<string>();
54      sortedKeys.Sort();
55      sortedKeys.Reverse();
56  } catch (Exception ex) {
```

52: 58 Comment

Figure 3-1. *Code difference (part 1) in updated file to illustrate the differential reading*

Figure 3-2 shows the third and last section where the code was changed. Inside the loop in which the foreach clause is embedded, lines are processed sequentially. The original string and the string to replace it are maintained in the dictionary (keyMapping). The old code simply loops over every possible word in the dictionary, checks if it appears in the line, and replaces it if it does. The modified code however, loops over reversely sorted words. Based on the reading and understanding of the code, the modified code seems working as intended.

C:\yzhu\SoftwareReading\codes\Form1_old.cs

2/18/2015 11:25:32 PM 6,286 bytes C,C++,C#,ObjC Source UTF-8 BOM PC

```
116
117         foreach (var kvp in keyMapping) {
118             if (line.Contains(kvp.Key)) {
119                 line = line.Replace(kvp.Key, kvp.Value);
120                 break;
121             }
122         }
123         sw.WriteLine(line);
```

C:\yzhu\SoftwareReading\codes\Form1_new.cs

5/26/2016 9:20:57 AM 6,587 bytes C,C++,C#,ObjC Source UTF-8 BOM PC

```
122
123         foreach (string mykey in sortedKeys) {
124             if (line.Contains(mykey)) {
125                 line = line.Replace(mykey, keyMapping[mykey]);
126                 break;
127             }
128         }
129         sw.WriteLine(line);
```

52: 58 Comment

Figure 3-2. *Code difference (part 2) in updated file to illustrate the differential reading.*

In this example, the old and new code is displayed in two panels, which are arranged vertically for display purpose here. Two panels can also be arranged horizontally. Neither vertical nor horizontal arrangements utilize the screen optimally. It is possible to display old and new code in a single panel, however (Lanna & Amyot, 2011).

3.5 Summary

Software professionals are trained to write software documents. But reading, understanding, analyzing, assessing quality, and utilizing the software document are equally important. This chapter defines software reading and software reading techniques and classifies the software reading techniques based on their characteristics. This chapter then discusses ad hoc reading, checklist-based reading, and differential reading, which can be applied to any software artifacts. Due to its long history, checklist-based reading has a few variations and the community accumulated some heuristics on checklist best practices. Ad hoc reading and checklist-based reading are the two most practiced reading techniques. According to a recent industry survey (Ciolkowski et al., 2003), ad hoc reading is used in 35% of the software reviews and checklist-based reading is used in 50% of the reviews. Ten percent of the reviews use some specific or advanced reading techniques and the remaining 5% use simulation or other techniques. Ad hoc and checklist-based reading techniques are also frequently chosen as a baseline and other reading techniques are compared to them. Next chapter discusses advanced reading techniques, scenario-based reading.

3.6 References

(Basili, 1996) V. Basili, G. Caldiera, F. Lanubile, and F. Shull, Studies on reading techniques, In Proceedings of the Twenty-First Annual Software Engineering Workshop, SEL-96-002, pp.59-65, 1996.

(Brykczynski, 1999) B. Brykczynski, A survey of software inspection checklists, Software Engineering Notes, vol.24, no.1, pp.82-89, 1999.

(Ciolkowski, 2003) M. Ciolkowski, O. Laitenberger, and S. Biffl, Software reviews: the state of the practice, IEEE Software, vol.20, no.6, pp.46-51, 2003.

(Clements, 2011) P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, Documenting Software Architectures: Views and Beyond, 2nd ed., Addison-Wesley, 2011.

(Dunsmore, 2002) A. Dunsmore, M. Roper, and M. Wood, Further investigations into the development and evaluation of reading techniques for object-oriented code inspection, Proceedings of the 24th International Conference on Software Engineering, pp.47-57, 2002.

(Fagan, 1976) M.E. Fagan, Design and code inspections to reduce errors in program development, IBM Systems Journal, vol.15, no.3, pp.182-211, 1976.

(Gilb, 1993) T. Gilb and D. Graham, Software Inspection, Addison-Wesley, 1993.

(Hatton, 2008) L. Hatton, Testing the value of checklists in code inspections, IEEE Software, vol.25, no.4, pp.82-88, 2008.

(Humphrey, 1997) W.S. Humphrey, Introduction to the Personal Software Process, Addison-Wesley Publishing Company, 1997.

(Jablokow, 2003) K.W. Jablokow, Systems, man, and the paradox of structure, IEEE International Conference on Systems, Man and Cybernetics, vol.3, pp.2374-2380, 2003.

(Johnson, 1998) P.M. Johnson and D. Tjahjono, Does every inspection really need a meeting? Empirical Software Engineering, vol.3, pp.9-35, 1998.

(Kollanus, 2009) S. Kollanus and J. Koskinen, Survey of software inspection research, The Open Software Engineering Journal, vol.3, pp.15-34, 2009.

(Lanna, 2011) M. Lanna and D. Amyot, Spotting the difference, Software: Practice and Experience, vol.41, no.6, pp.607-622, 2011.

(McMeekin, 2008) D.A. McMeekin, B.R. von Konsky, E. Chang, and D.J.A. Cooper, Checklist based reading's influence on a developer's understanding, 19th Australian Conference on Software Engineering, pp.489-496, 2008.

(Mernhart, 2010) M. Mernhart, A. Mauczka, and T. Grechenig, Adopting code reviews for agile software development, IEEE Agile Conference, pp.44-47, 2010.

(Porter, 1994) A.A. Porter and L.G. Votta, An experiment to assess different defect detection methods for software requirements inspections, in Proceedings of the 16th International Conference on Software Engineering, pp.103-112, 1994.

(Rong, 2014) G. Rong, H. Zhang, and D. Shao, Investigating code reading techniques for novice inspectors: an industrial case study, Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE'14), Article No.33, 2014.

(Shull, 2002) F. Shull, Software reading techniques, in Encyclopedia of Software Engineering, John Wiley and Sons, 2002 (http://onlinelibrary.wiley.com/doi/10.1002/0471028959.sof273/abstract, accessed on Dec. 17, 2015).

(Tian, 2005) J. Tian, Software Quality Engineering: testing, quality assurance, and quantifiable improvement, IEEE Computer Society Press, 2005.

(Votta, 1993) L.G. Votta Jr., Does every inspection need a meeting? Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering, pp.107-114, 1993.

(Winkler, 2005) D. Winkler, S. Biffl, and B. Thurnher, Investigating the impact of active guidance on design inspection, LNCS 3547, pp.458-473, 2005.

CHAPTER 4

Scenario-Based Reading Techniques

In the last chapter we discussed basic reading techniques such as ad hoc reading and checklist-based reading. Those are the most frequently used reading techniques in industry software reviews (Ciolkowski et al., 2003). This chapter discusses recent developments of software reading techniques and focuses on the family of reading techniques called scenario-based reading.

4.1 Principles of Scenario-Based Reading

Scenario-based reading was developed by Porter and Votta (1994), and their paper triggered an active research on software reading techniques. Their method was later renamed "defect-based reading" and the term "scenario-based reading" was reserved for a group of reading techniques that use different ways to decompose reading scenarios. The family of scenario-based reading techniques includes defect-based reading, perspective-based reading, etc. Defect-based reading concentrates on specific defect classes and perspective-based reading focuses on viewpoints of consumers of a document. Defect-based reading was originally employed for requirements inspection but could be applied to other software artifacts as well.

Scenario-based reading was motivated by the ideas behind the active design review proposed by Parnas and Weiss (1985). In an active design review, reviewers are selected based on their expertise. Each reviewer is given a specific area to focus on and required to answer specifically designed questions that can only be answered after a critical reading and analysis of the design, which solves the issue of passive participation. Scenario-based reading is based on operational scenarios that give specific guidance to readers. The guidance can be a set of questions, an assignment, or explicit instructions on how to conduct reviews and look for defects. To improve effectiveness, the overlap of scenario assignments shall be minimized. Because of detailed instructions on reading and defect detection, this group of reading techniques is considered systematic.

The working of scenario-based reading can be best depicted by the diagram in Figure 4-1, where the rectangle shape represents a software artifact to be reviewed, and both open and filled circles with various sizes represent different kinds of defects in the artifact. Two reading techniques are schematically illustrated in Figure 4-1.

© Yang-Ming Zhu 2016

Y.-M. Zhu, *Software Reading Techniques*, DOI 10.1007/978-1-4842-2346-8_4

Figure 4-1. *Comparison of unsystematic and systematic (scenario-based) reading (adapted from Porter & Votta [1994])*

To the left of Figure 4-1 is an unsystematic reading, where one or more readers scan through the document and look for defects, which are shaded as vertical, horizontal, or slanted lines. There are at least two problems with that approach owning to its unsystematic nature: (a) there are overlaps on the portions covered by different readers, and (b) there are regions that are not covered by any reader. To the right is a systematic reading, where different readers are purposely selected to exam the software artifact, based on their individual expertise. They are also given specific instructions on where and how to detect defects. These two ensure that there will be no or minimal overlaps among what they will cover and the entire document is covered. Systematic approaches with specific responsibilities improve coordination and reduce gaps, which increases the overall defect detection effectiveness of review.

The scenario used here refers to a process-driven or operational scenario, not to be confused with scenarios in the use-case context. It is expressed in the form of algorithms that readers can apply to traverse a document with a particular emphasis. In practice several scenarios must be combined to provide an adequate coverage of a document, since each scenario is focused, detailed, and specific to a particular viewpoint (Basili et al., 1996).

Before analyzing and detecting defects in a document, one must understand the document first. In cognitive science, comprehension is often characterized as the construction of a mental model that represents the objects and relationships in the text document (Schulte et al., 2010). An operational scenario has a set of activities to help a

reader build a model of the software artifact and a set of review questions related to the model. The model is simply an abstraction of the software artifact under review. In the defect-based reading, the model can be different types of defects the document may have. In the perspective-based reading, if the reader reads the code from a tester point of view, the model can be related to how testing can be performed. While the reader builds a model and answers the questions based on the analysis of the model with a particular emphasis, he or she can note down any anomaly and log defects. To minimize the potential reading overlap, scenarios shall be orthogonal to each other. One team member is responsible for one scenario and multiple members provide complete coverage. The findings of individual team members are aggregated via meeting or non-meeting, which depends on the inspection process bestowed by the organization. The choice of model or abstraction and the types of questions depend on artifacts, the organization's problem history, or the objectives of reading.

Compared with other reading techniques such as ad hoc or checklist-based reading, the scenario-based reading techniques have the following benefits (Shull et al., 2000):

- *Systematic*: Specific steps for individual reading are well-defined.
- *Focused*: Different readers focus on distinct aspects of the document.
- *Allow controlled improvement*: Based on experience and feedback, one can identify and improve specific aspects of reading techniques.
- *Customizable*: Organizations can customize reading techniques to a specific project or organization setting.
- *Allow training*: One can use reading techniques to train oneself in applying the techniques.

4.2 Defect-Based Reading

Defect-based reading is the first proposed reading technique in the family of scenario-based reading and focused on detecting specific types of defects. The main idea is that if each reader uses different but systematic techniques to search for different, specific classes of defects, he or she and the whole team will have a better chance to detect defects effectively than readers applying ad hoc reading or checklist-based reading (Porter et al., 1995). In this defect-based software reading, each reader is given specific steps to discover a particular class of defects. Each reader's role or responsibility is specific and narrowly defined, such as ensuring appropriate use of hardware interfaces, identifying untestable requirements, or checking conformance to coding standards and guidelines. The defect-based reading technique has been applied to detecting defects in software requirements specifications (SRS) and we discuss it in the context of SRS reading.

4.2.1 Taxonomy of Defects in Requirements Specifications

Defects in SRS can be divided into two broad types: omission and commission (Porter et al., 1995). An omission defect means important information is missing from the SRS, while a commission type defect means incorrect, redundant, ambiguous, or conflicting information is presented in the SRS. Depending on what is missing, the omission type can be further divided into four groups:

- *Missing functionality*: The specification of the internal operational behavior of the system is not included in the SRS.
- *Missing performance*: The performance specification is not present in the SRS or is not acceptable by end users in an acceptance testing.
- *Missing environment*: The environment in which the system will be operated is not specified or is specified incompletely in the SRS, including infrastructure, middleware, hardware, software, database, personnel, and so on.
- *Missing interface*: The interfaces through which the system interacts and interoperates with outside or communication mechanisms or protocols through which the system exchanges data with outside is not included in the SRS.

The commission type of defects also has four subcategories: ambiguous information, inconsistent information, incorrect or extra functionality, and wrong section:

- *Ambiguous information*: Technical terms, phrases, or anything essential for readers to correctly understand and interpret the SRS is undefined or defined in such a way to cause misunderstanding or misinterpretation.
- *Inconsistent information*: Information in different parts of the SRS contradicts each other directly or indirectly, or actions or behaviors cannot both be correct or carried out by the system.
- *Incorrect fact*: Specifications in the SRS cannot be true in the given context.
- *Wrong section*: The SRS is poorly organized or the specification is put in a wrong section of the SRS.

There are other defect taxonomies. IEEE Std. 1028-2008 categorizes anomaly (defects) as missing, extra (superfluous), ambiguous, inconsistent, not conforming to standards, risk-prone, incorrect, unachievable, and editorial (IEEE 2008). In the recent survey, Walia and Carver (2009) attributed the defects to their sources, i.e., people, process, and documentation.

4.2.2 Defect-Based Reading Techniques

When readers are assigned to review and detect defects in a software artifact, they can have the same general responsibilities, e.g., detect as many defects as they can, and their reading activities are not coordinated. Alternatively, in a coordinated team, readers can have separate and different responsibilities. As the active design review suggests, individual readers can be more effective if they are assigned specific responsibilities and provided systematic techniques to meet those responsibilities (Parnas & Weiss, 1985). The defect-based reading technique is motivated by the ideas behind the active design review. In this reading technique, a collection of procedures (operational scenarios) for detecting particular classes of defects is developed. Each reader executes one single scenario, and multiple readers are coordinated to improve the coverage while minimizing the overlap (Porter et al., 1995).

One way to come up with operational scenarios is to make use of the defect taxonomy, as discussed by Porter and his colleagues (1995). The general defect taxonomy discussed earlier can be given to ad hoc readers and used to define their responsibilities. To support checklist-based reading, detailed and concrete questions as checklist items can be developed under each defect classes. The checklist is thus a refinement of the taxonomy. Those questions can be motivated by industrial checklists. For example, to detect defects related to inconsistent information, one can have following checklist items (Porter et al., 1995):

- Are the requirements mutually consistent?
- Are the functional requirements consistent with the system overview?
- Are the functional requirements consistent with the operating environment?

The actual defect types, groups, and classes shall depend on the software artifact under concern and be based on the organizational experience. Checklist items shall be customized to detect those defects accordingly. Since the checklist is derived from the defect taxonomy, checklist responsibilities are a subset of the ad hoc responsibilities.

Scenarios for defect-based reading are further derived from the checklist by substituting the checklist items with procedures designed to implement them. Hence, the responsibilities of defect-based reading is a subset of checklist-based reading. If defect-based reading scenarios cover all checklist items, defect-based reading and checklist-based reading can detect the same kinds of defects. The diagram in Figure 4-2 shows the relationship among the ad hoc, checklist-based, and defect-based reading techniques. The horizontal axis labels the reading technique and denotes the increased degrees of details in reading procedures. As the reading technique becomes more sophisticated (moving to the right), the defect coverage decreases. The reading responsibilities and the classes of defects targeted for the latter reading technique are the subset of the former reading techniques (moving to the left).

Figure 4-2. *The relationship among ad hoc reading, checklist-based reading, and defect-based reading (adapted from Porter et al. [1995])*

Porter et al. (1995) developed three defect detection scenarios for their defect-based reading. Each scenario was designed for a specific subset of checklist items. Those scenarios were intended to detect defects in areas of data type inconsistency, incorrect functionality, and ambiguous or missing functionality. According to their estimate, those scenarios cover about 50% of the defects present in the SRS. Their Incorrect Functionality Scenario is reproduced in Panel 4-1: Defect-Based Reading (Incorrect Functionality Scenario), and interested readers can refer to their original paper for the other two reading scenarios.

PANEL 4-1: DEFECT-BASED READING (INCORRECT FUNCTIONALITY SCENARIO)

1. For each functional requirement, identify all input and output data objects.
   a. Are all values written to each output data object consistent with its intended function?
   b. Identify at least one function that uses each output data object.
2. For each functional requirement, identify all specified system events.
   a. Is the specification of these events consistent with their intended interpretation?

3. Develop an invariant for each system mode or state, i.e., under what conditions must the system exit or remain in a given mode.
   a. Can the system's initial conditions fail to satisfy the initial mode's invariant?
   b. Identify a sequence of events that allows the system to enter a mode without satisfying the mode's invariant.
   c. Identify a sequence of events that allows the system to enter a mode but never leave.

The operational scenario leads a reader through the SRS and gives the reader instructions where and how to find defects. For example, in (1), "where" would be the input and output data object related to a functional requirement, and "how" is to check if all values generated for the output data object is consistent with its intended function and check whether the output data object is used by some functions. From this scenario, one can clearly see the influence of active design review. It turns a passive reader into an active reader (e.g., step 3) and therefore improves readers' effectiveness.

Defect-based reading is a systematic technique that focuses the readers on detecting the specific class of defects that the scenario targets, providing a mechanism to focus the reading according to project or organizational objectives. As scenarios are typically derived from checklist items, the technique suffers the similar drawbacks as checklist-based reading. For maximal coverage of defects, multiple scenarios have to be defined, which could be an elaborated process. In practice, defect-based reading has to be applied in combination with other reading techniques so that defects not covered by the defect scenarios can still be detected.

Defect-based reading has been applied to requirements reading. In theory, it can be applied to other artifacts. There are no other applications reported yet.

4.2.3 Empirical Experiences

Porter et al. (1994; 1995) used graduate students in computer science in their experiments to test the effectiveness of defect-based reading by comparing it with ad hoc reading and checklist-based reading. To some degree, they replicated and expanded their experiment and came to the same conclusion. The taxonomy was provided to all readers, but ad hoc readers were not given further instructions. The checklist and the scenarios were developed in the way discussed earlier. They concluded that defect-based reading had the highest defect detection rate (about 35% improvement over the other two reading techniques), followed by ad hoc reading, and checklist-based reading had the lowest effectiveness. Further analysis confirmed that defect-based scenario readers were more effective than ad hoc and checklist readers at finding defects the scenario was designed to detect, and all readers were equally effective at detecting other defects not targeted by scenarios. Scenarios helped readers focus on specific defect types, and they didn't compromise readers' ability to detect other defect types not covered by scenarios. Porter and Votta (1998) replicated the same experiment using industrial professionals and reported the similar results, although the performances of students and professionals were different.

Porter et al. studies were replicated by other researchers (Fusaro et al., 1997; Miller et al., 1998; Sandahl et al., 1998). However, the results were not consistent with the original findings, mainly defect-based reading was not significantly better than other ad hoc and checklist-based readings. Hayes used meta-analytic (analysis of analyses) methods to synthesize the series of replicated experiments with seemingly contradictory results (Hayes, 1999). He suggested that the differences might be caused by subjects' familiarity with the formal notation used in the SRS and with the software domains used in the experiment (e.g., cars with cruise control were popular in the United States but not in the Europe).

4.3 Perspective-Based Reading

Perspective-based reading was developed by Basili et al. (1996). It provides a set of operational scenarios. Each reader reads a software artifact under review using one of the scenarios from a particular perspective, and the combination of different perspectives yields a better coverage of the document. It was originally applied to requirements documents but subsequently applied to high-level designs, source code, and user interface. Perspective-based reading in fact includes a family of reading techniques. In this section we present perspective-based reading in its generic form first, then discuss specifics when the technique is applied to different types of software artifacts. Finally we shed light on why the reading techniques are effective in practice.

4.3.1 A Generic Perspective-Based Reading

Software artifacts are created with various purposes. It is important to ask their stakeholders to read and assess if the documents meet their needs and quality expectation, from a particular point of view, which is called perspective. Perspective-based reading focuses on a point of view or needs of the main customers of a document. Each reader reads from their respective perspective with a narrowly defined focused view, which leads to an in-depth analysis of potential defects or anomalies in the document. The union of multiple perspectives provides an extensive coverage of the document so that no defects will be missed.

Defect reports from individual readings are aggregated at the end of reading. It is not necessarily true that we select one reader for each perspective. How many readers are assigned to a given perspective depends on the organizational and project context. For example, if more omission defects are expected in a requirements specification, one may assign more readers to the user role, since a user perspective offers better opportunity for exposing omission defects. If there are not enough resources available, one reader may be assigned to multiple roles and he or she can simulate each perspective in turn.

When read with a particular perspective, defects that a perspective is targeting at would have a higher probability to be detected, while other defects not targeted by the perspective might have a lower probability to be detected. Thus an individual using perspective-based reading may not be necessarily more effective and efficient than one using other reading techniques. Collectively on the team level, perspective-based reading could be more effective and efficient as each perspective detects different kinds of defects.

The idea of perspective-based reading is not completely new, however. It synthesizes ideas that have already been applied in software inspection, but have never been worked out in great detail. The influence of the active design review was already mentioned. Fagan (1976) stated that the code should be inspected by its tester. When reporting AT&T experience on in-process inspection, Fowler (1986) stated, "[E]ach meeting participant takes a particular point of view when examining the workproduct." Graden et al. (1986) also reported that when collecting the software inspection data, each participant must denote the perspective (customer, requirements, design, test, maintenance) by which they have evaluated the deliverable.

4.3.1.1 Reading Scenarios and Template

Perspective-based reading has a set of operational scenarios that provide reading guidance to readers. In an ideal case, those scenarios shall have a minimal overlap in defects detection, which is very hard to achieve in practice (Regnell et al., 2000). What scenarios to provide to readers depends on many factors such as artifact types, organization's history on defects, as well as organizational objectives of software reviews.

The scenarios are developed using a scenario template, which has three sections:

- *Introduction*: The introduction section explains the stakeholder's interests in the artifact.
- *Instruction*: The instruction section provides guidance regarding how to read and extract relevant information from artifacts or from the descriptions of artifacts and to optionally create other artifacts such as test cases from a tester's perspective or a high-level design from a designer's perspective. It helps a reader focus on the most relevant information and actively work with the document to gain a deep understanding of it.
- *Question*: The question section is for reviewers to answer by following the instructions above. Questions are derived from past defect classes.

It shall be noted that to review software artifacts, one can ask real stakeholders and experts to review. One can also ask other readers to review, who will mimic real stakeholders' perspective. For the latter case, the introduction section of a scenario is important to set up the scope and expectation. Even when real stakeholders and experts are invited to review a document, they will appreciate the concrete guidance provided in the latter two sections of the scenario. However, people with less experience seem to follow the prescribed perspectives while people with more experience are more likely not to conform to their prescribed perspectives (Basili et al., 1996).

4.3.1.2 Developing Reading Scenarios

Scenarios play a pivotal role in perspective-based reading. How are the scenarios created from the first place? Herein this section, we provide some guidance on scenario creation based on Laitenberger and Atkinson (1999).

1. Document identification. Identify documents subject to reading. This might be a software requirements specification document, software architectural or design description including diagrams and graphical models, source code files, a test plan including detailed test cases, or a graphical user interface mockup, etc.

2. Stakeholder identification. For the identified artifacts and their descriptions, identify key stakeholders who have a vested interest in the artifacts to be reviewed. Those stakeholders can be subject matter experts, users, or maintainers. They can be the consumer of the artifacts or the producer of upstream documents from which the current documents are based. They can also be peers of the author who created the artifacts. Prioritize those stakeholders according to the potential gain from the reading activities and resource constraint.

3. Perspective understanding. For the identified key stakeholders, understand their perspectives by deciding and collecting what information and descriptions are important to them. You can interview and survey stakeholders to collect their inputs. After having understood the stakeholders, the scenario writer identifies which documents contain that information and how to identify and extract the information.

4. Introduction development. The scenario writer writes the introduction to a particular perspective and highlights stakeholders' interests in the artifacts.

5. Instruction development. The scenario writer develops the detailed instruction on how to identify and extract, step by step, the required information from the artifact or the descriptions of the artifact.

6. Questions development. The scenario writer compiles a list of questions for a reader to answer based on the extracted information and the reader's understanding of the artifact. The checklist, if available, as well as the historical defects can be a source for those questions. The writer makes sure only pertinent questions are included, i.e., those the reader could answer by reading and analyzing the documents from the given perspective.

7. Scenario review, testing, and updating. After the scenarios are written, the scenario writer should invite stakeholders to review and pilot its usage to get rid of kinks in the scenarios. The above steps shall be iterated as necessary. The scenarios shall also be kept up to date. When new defects are uncovered, or the contents and structures of the documents are changed, the scenario instruction and questions shall be updated as well.

4.3.1.3 Characteristics of Perspective-Based Reading

Perspective-based reading has been developed to satisfy a variety of goals, which are discussed below (Ciolkowski et al., 1997):

- *Adaptable*. Perspective-based reading is adaptable to particular software artifacts, being it requirements specification, design description, source code, and user interface. It is also adaptable to whatever notation used in the document. For example, the requirements might be specified using natural languages or a more formal specification language.
- *Tailorable:* Perspective-based reading is tailorable to organizational and project setting: The number of perspectives, the number of readers using a scenario, and the reading instruction and questions in a scenario are all customizable.
- *Detailed*: A reading scenario gives a reader a well-defined instruction how to traverse a document. Hence the reading process is repeatable and the effectiveness of defect detection becomes less dependent on individual's experience and expertise.
- *Focused*: A particular perspective provides particular coverage of part of the document and multiple perspectives combined provide coverage of the whole document. This gives individual reader a focus, which enables an in-depth analysis. At the same time, multiple perspectives ensure no areas are left unread.
- *Specific*: Each reader has a specific, narrowly defined responsibility, with a specific tool (scenario) to meet this responsibility. The reader knows what perspective to simulate, which scenario to use, and where and how to detect defects.

Those benefits will become more evident as we go through the reading techniques for specific types of documents in the remainder of this chapter. Perspective-based reading is not without drawbacks. The most noticeable one is the demanded effort. The improved rate of defect detection comes at the price of a higher effort on the reviewer. Basili et al. (1996) reported about 30% more effort, depending on the document under review. Much of the extra effort was spent on creating additional model and documentation, which might be required in a later stage of the development life cycle, e.g., test cases created during the requirement reading can be reused later in the testing phase

4.3.2 Perspective-Based Requirements Reading

Many researchers have applied perspective-based reading to requirements documents. In fact, it was first applied to the requirements document when the reading technique was developed. This should not be a surprise, given the importance of a requirements document in the software development process. Any problems and issues in requirements specification detected and corrected early can save a lot of time and money of the development organization, since it will reduce potential rework of design, implementation, and testing, or avoid embarrassment if the product is shipped to customers with residual defects.

4.3.2.1 Reading Scenarios for Requirements Specifications

There are many possible stakeholders of a requirements document. Basili et al. (1996) identified three key consumers of a requirements document, i.e., end user, tester, and designer. The requirements describe functionalities and performance expectations that the finished software product must meet; thus the end user is an important stakeholder. The requirements are the basis for system design and the designer must create a design that provides the functionalities with the performance constraints documented in the requirements; therefore the designer is another important stakeholder. The finished software system must be verified to be conformant to functional and performance specifications, which is typically conducted by a tester; hence the tester is yet another important stakeholder of the requirements document.

From the user's perspective, requirements must be complete and correct and provide necessary system functionality. From the tester's perspective, requirements must be testable with unambiguity so that the tester can construct test cases. From the designer's perspective, requirements must have enough details with accuracy so that the designer can design major system components. Failing to satisfy any of the above needs amounts to a requirement defect. Operational scenarios must be developed for the perspectives of the end user, tester, and developer.

Given the overarching impact of a requirements specification, it often involves more inspectors or readers than other types of software artifacts do. Other stakeholders and their perspectives might be important or even critical, depending on a particular environment. For example, for the software requirements for a nuclear power plant-related system, designer, tester, user, maintainer, verifier, regulator, and contractor might be identified as key stakeholder (Lahtinen, 2011). As a nuclear power plant has a long lifetime (60 years), maintainer is a critical stakeholder as well.

One of the key ideas behind the active design review is to engage reviewers for active participation. With the same spirit, when a reader reads from a perspective, he or she is asked to create a physical model and answer questions from the same perspective based on the analysis of the model. When reading from the tester's perspective, the reader is asked to design a set of test cases and answer questions related to the activities being performed. Similarly, when reading from the developer's perspective, the reader is asked to create a high-level design, and when reading from the user's perspective, the reader can develop use cases or draft a user's manual. The exact document to generate can vary according to the reader's experience or the organization's needs. Shull et al. (2000) provided a test-based reading scenario that asked a reader to develop a test case using the equivalence partitioning technique (generating and testing the equivalent sets). Chen et al. (2002) took a step even further, where they proposed a problem-driven approach to select a method for test suite construction. The questions to answer are tied to the defects taxonomy, and additional questions can be added for newly found defect classes.

An example test-based reading scenario is show in Panel 4-2: PBR – Requirement Reading Scenario for Testers. It was adapted from Basili et al. (1996). Question 1 is more operable than a general question one might find on a checklist such as "Are all items sufficiently and unambiguously described?" and intended to detect defects of the "missing information" class. Question 2 is intended to detect defects of the "inconsistent" class. Question 4 is intended to detect defects of the "ambiguous" class. Question 5 is intended to detect defects of the "incorrect facts" class.

4.3.2.2 Empirical Experiences

Basili et al. (1996) tested their perspective-based reading techniques on requirements documents using professional software developers as subjects. The techniques were compared to the existing techniques at NASA Goddard Space Flight Center Software Engineering Laboratory. Readers were given unique roles (user, tester, and designer) when reading four requirement documents, two from NASA domain and the other two generic. They reported that perspective-based reading provided significantly better coverage of both domain-specific and generic documents. The experiment of Basili et al. was replicated and original findings were confirmed in an academic environment (Ciolkowski et al., 1997). In the replicated experiment, perspective-based reading was compared to ad hoc reading. Perspective-based reading of requirements was replicated by many other groups. However, the positive effects were not observed all the time. Regnell et al. (2000) questioned whether all the perspectives were really orthogonal and people using them found different kinds of defects. To address that, Maldonado et al. (2006) replicated the experiment with students as subjects. They confirmed that, in general, different perspectives detect different kinds of defects. They also reported that perspective-based reading didn't outperform checklist-based reading and perspective-based reading and checklist-based reading appeared to be complementary on defect detection.

PANEL 4-2: PBR – REQUIREMENT READING SCENARIO FOR TESTERS

Introduction: Perspective-based reading is built around the concept that customers of a document are at the best position to judge if the document meets their needs and quality expectation. Different customer reads the document and detects defects from their respective perspective. This reading scenario assumes a tester's perspective. From the tester's perspective, requirements must be testable and unambiguous.

Instructions: For each requirement or functional specification, create a test case to verify that the implementation satisfies the requirement. Compose test cases into a test suite. While composing the test suite, answer questions below.

Questions:

1. Do you have all information to identify the item being tested and to identify your test criteria? Can you create reasonable test cases for each item based on the criteria?
2. Is there another requirement for which you would create a similar test case but would get a contradictory result?
3. Can you be sure the test generated will output the correct value in the correct units?

4. Are there other interpretations of the requirement that the implementer might make due to the particular description of the requirement? Will this impact the test case and its oracle?
5. Does the requirement make sense based on your domain knowledge or from what is specified in the general description?

4.3.3 Perspective-Based Design Reading

Most software reading techniques target textual documents such as requirements specifications and source code files. Design documents are typically written as textual descriptions annotated with graphical models and diagrams, which is particularly true for object-oriented designs. An object-oriented approach is more popular nowadays than other paradigms such as procedure or structured programming, but it is much more complicated and error-prone in both design and implementation. This section discusses how the perspective-based reading technique can be customized to read the object-oriented designs.

4.3.3.1 Reading Scenarios for Design Documents

Unified modeling language (UML) is often used in object-oriented modeling at analysis and design levels. Use-case diagrams are used to capture the interaction with the system and the requirements of the system. Class diagrams are used to document the static structure. The dynamic behaviors can be captured using state diagrams, activity diagrams, sequence diagrams, or collaboration diagrams. The last two kinds of diagrams capture identical information from different perspectives. What UML diagrams and textual documents to generate depend on the project and team needs. Laitenberger and Atkinson (1999) suggested operation schema, class diagrams, collaboration diagrams, operation pseudocode, and data dictionary, where the operation schema is simply a method description, the operation pseudocode is the pseudocode for the method, and the data dictionary is a catalog of each modeled object with description.

We can identify the following stakeholders of a design: requirements engineer, designer, implementer, tester, and maintainer. A requirements engineer has a high interest to make sure requirements specifications have been fully and correctly considered in the design. A designer is primarily interested in the correctness of the design models and the analysis documents. An implementer wants to make sure all necessary information is provided in the design models so that he or she can realize the design. A tester wants to know what functionalities are implemented and how to develop test cases to verify the correctness of those operations. A maintainer will extend the design and implement new functionalities in the future and he or she has a vested interest if the design has an acceptable maintainability. The number of reading scenarios to develop for stakeholders depends on the organizational context, history, and objectives. Laitenberger et al. (2000) used three reading scenarios (designer, implementer, and tester), but they used four (requirements engineer, designer, tester, and maintainer) in (Laitenberger & Atkinson, 1999). Sabaliauskaite et al. (2003) used three reading scenarios (user, designer, and implementer).

Stakeholders have vested interests in different documents assembled for a design. A possible association is shown in Table 4-1. The concern of a requirements engineer is the correctness and completeness of the functional specification at the end of the analysis. Particularly, various analysis models shall be consistent. Thus the requirements engineer has a high interest in the data dictionary, method descriptions (operation schema), as well as the responsibility assigned to each class.

Table 4-1. *Stakeholders' Interests in Different Design Description Documents (Adapted from Laitenberger et al. [1999; 2000])*

	Requirements Engineer	Designer	Implementer	Tester	Maintainer
Operation schema	x	x		x	
Class diagram	x	x	x		
Collaboration diagram		x	x	x	x
Operation pseudocode			x	x	x
Data dictionary	x				

The designer concerns the correctness and completeness of the design diagrams with respect to the analysis. In terms of correctness, design diagrams shall be consistent with analysis diagrams. In terms of completeness, all elements in analysis diagrams shall be reflected in design diagrams with sufficient details. Thus he or she is interested in the operation schema, collaboration or sequence diagrams, and class diagrams. A sample design reading scenario for designer is given in Panel 4-3: PBR – Design Reading Scenario for Designers, which is adapted from Laitenberger et al. (2000). One can customize it when experimenting or adopting perspective-based reading.

PANEL 4-3: PBR – DESIGN READING SCENARIO FOR DESIGNERS

Introduction: Assume a designer's perspective when reading the design artifacts. A designer concerns the correctness and completeness of design diagrams with respect to analysis diagrams.

Instructions: Get familiar with the purpose of the system and the requirements (use cases). Locate the class diagrams, compare them with analysis diagrams, and make sure classes, attributes, methods, associations, constraints, and other abstract concepts are consistent. As design diagrams are a refinement of analysis diagrams, design diagrams sometimes contain additional elements. Locate collaboration diagrams and compare them to the corresponding operation schema, and make sure the messages, parameters and their types, and the resulting behaviors are consistent.

Questions: While following the instruction, answer questions:

1. Is there anything in analysis documents but not in design documents?
2. Are the messages appropriate and consistent from the collaboration diagrams and operation schema?
3. Are the system start-up conditions clear and correct?
4. Are there corresponding sent messages in operation schema and collaboration diagrams?
5. Is the order of messages correct in the collaboration diagram?
6. For every change of attributes, objects, states, or links in the operation schema, is there a corresponding sent message in the collaboration diagram?
7. Are there any discrepancies of functionality described in the operation schema and in the collaboration diagram?

The main responsibility of an implementer is to translate the design into code. As such, the implementer is concerned with the consistency and completeness of the design, as well as whether the design is implementable. To check for consistency, the implementer reads and detects any contradictions among design elements scattered in multiple diagrams. To check for completeness, the implementer reads and detects any missing elements from diagrams based on the analysis and design models. The implementer makes a professional judgment regarding whether the design is implementable based on project constraints, technologies and development environment, as well as other considerations. The implementer has high vested interests in the class diagram, collaboration diagram, and pseudocode. He or she will verify the messages received by an object as documented in the sequence or collaboration diagram are implemented by the class, and the collaboration diagram is consistent with the operation pseudocode.

The tester is concerned if the operations are testable. A productive approach is to walk through test cases and ensure test cases are correct with respect to the associated functions. The tester is thus interested in the operation schema and their collaboration diagram as well as pseudocode that implements the collaboration. A design reading scenario for testers is illustrated in Panel 4-4: PBR – Design Reading Scenario for Testers, which is adapted from Laitenberger et al. (2000).

We contrast this tester-based reading scenario with that for the requirements specification. Here since the internal implementation is available as revealed in the pseudocode and collaboration diagrams, white-box testing becomes feasible, i.e., one can develop test cases to cover the various branches and loops, which is not feasible when only the requirements are available.

The maintainer is concerned with the maintainability of the system. To improve the maintainability or modifiability of a system, the designer shall follow well-established design principles such as low coupling and high cohesion as well as simplicity of the design. Bass et al. (2013) have written about the tactics of improved modifiability. The proper places to check whether best practices are followed are collaboration diagrams and pseudocode.

PANEL 4-4: PBR – DESIGN READING SCENARIO FOR TESTERS

Introduction: Assume a tester's perspective when reading design artifacts. Ensure the correctness and robustness of all operations and their testability. For this perspective-based reading, test cases are used. If they don't exist yet, have them created.

Instructions: Get familiar with the requirements (use cases). Locate the operation schema and identify test cases derived from the schema. Locate the collaboration diagram associated with the operation schema. Walk through the collaboration diagram and simulate the execution of the test cases; check if the correct results are generated, all branches and paths are covered, and the resulting behavior is consistent with schema.

Questions: While following the instruction, answer questions:

1. Do the branches in the collaboration diagram match the condition outcomes in the operation schema?
2. Are all possible inputs and their combinations properly addressed in the operation schema and the pseudocode?
3. Are the effects of each collaboration diagram consistent with the corresponding operation schema?

4.3.3.2 Empirical Experiences

Laitenberger et al. (2000) compared perspective-based design reading to checklist-based reading, with practitioners (experienced students) as subjects. The software artifacts under reading were UML design documents. Their checklist was developed from scratch. Three reading perspectives were employed (designer, implementer, and tester). For both reading techniques, defects were pooled through the defect collection meeting attended by three team members. They reported that teams found more defects using perspective-based reading than using checklist-based reading and, on average, the improvement of detection effectiveness is 41%. When considering the defect detection effort, perspective-based reading was also cost-effective for the defect detection phase, the meeting phase, and the overall inspection. The readers also believed that applying perspective-based reading helped them improve their understanding of the systems.

Sabaliauskaite et al. (2003) also compared perspective-based reading and checklist-based reading of UML design documents with students as subjects. The UML diagrams include class diagrams, activity diagrams, sequence diagrams, and component diagrams. They used three reading perspectives (user, designer, and implementer). The authors concluded that defect detection effectiveness using both reading techniques was similar, but checklist-based reading was more effective for three-person simulated virtual teams.

4.3.4 Perspective-Based Code Reading

In industry practice, code documents are still the most frequently inspected software artifacts. It is thus natural to apply perspective-based reading to code. This section discusses the work in that area.

4.3.4.1 Reading Scenarios for Code Modules

Code modules subject to inspection include implementation source code files as well as their operational descriptions as specification. We can identify two main stakeholders of those artifacts: code analyst and tester (Laitenberger et al., 2001). While a code analyst is concerned if the code implements the right function, a tester is concerned if the code implements the function right.

One could certainly identify more stakeholders. Laitenberger and DeBaud (1997) split the tester into the module tester and the integration tester. A maintainer has a high interest in the source code and he or she is concerned if the code is easily understandable to allow future modification with minimal effort. Code can also be checked if it is compliant to coding standards and guidelines. This latter checking can be accomplished with modern tools. Laitenberger et al. (2001) recommended two perspectives for reading, which is the minimal set of viewpoints. They argued that there is some quantitative evidence that a two-person inspection team decreases inspection cost while maintaining its effectiveness (Bisant & Lyle, 1989); there is little difference in the inspection effectiveness of two- and four-person inspections (both are significantly better than one-person inspection) (Porter et al., 1997); and code in practice is inspected by a small team (Gilb & Graham, 1993).

A code analyst locates different functions in the code file, reads and understands them, and forms high-level abstractions. When reading source code, he or she can use any code reading techniques such as stepwise abstraction for structured code or abstraction-driven reading for object-oriented code, which are discussed in a later chapter. The reader then compares the high-level abstraction to the code specification and notes any deviation as a potential defect. The reader has a set of limited questions to answer, specifically designed for his or her reading perspective, based on the code documents and the understanding the reader achieved at this point. A sample reading scenario is shown in Panel 4-5: PBR – Code Reading Scenario for Code Analysts, which is adapted from Laitenberger et al. (2001).

PANEL 4-5: PBR – CODE READING SCENARIO FOR CODE ANALYSTS

Introduction: Assume the code analyst reading perspective, who has to ensure the right functionalities are implemented in code.

Instructions: Locate the functions implemented in the code. Use the stepwise abstraction reading technique if it is a structured code; use the abstraction-driven and use-case-driven reading techniques if it is object-oriented code. Start with the least-dependent code and read bottom-up to form a high-level abstraction of the coded function. For each function, check if the abstraction you derived matches the specification; if there is a deviation, decide if it is a defect and log your findings.

Questions: While following the instruction above, answer questions:

1. Does the operation implemented in code match the one described in the specification?
2. Is there any operation described in the specification, but not implemented in the code?
3. Are data (constants, variables, etc) and their type used correctly?
4. Is the calculation performed correctly?
5. Is usage of the interface between different modules semantically correct?

A code tester identifies different functions implemented in the code and develops and runs test cases on them to ensure their correctness. The reader then mentally simulates the execution of each function guided by test cases and compares the outcome with specifications. Any deviation shall be noted as a potential defect. If test cases have been developed during design reading, those test cases can be reused here. The reader also has a set of limited questions to answer, specifically designed for this tester reading perspective. Those questions are intended to focus the reader's attention. A sample tester reading scenario is shown in Panel 4-6: PBR – Code Reading Scenario for Code Testers, which is adapted from Laitenberger et al. (2001).

PANEL 4-6: PBR – CODE READING SCENARIO FOR CODE TESTERS

Introduction: Assume the tester reading perspective, which has to ensure the functionalities implemented in code are correct.

Instructions: Locate the functions implemented in code. Determine the dependencies among these functions and build a call graph from them. Start with the nodes with the least dependencies and traverse the call graph backward, and develop or reuse test cases that allow you to mentally simulate the execution of the function implemented by the node to cover all branches and loops. For each simulated execution, compare the outcome with that in specification. If there is a discrepancy, decide if it is a defect and log your findings.

Questions: While following the instruction above, answer questions:

1. Do you have all required information to identify a test case?
2. Are branch conditions implemented correctly?
3. Can you generate test cases for all branches and loops? Can you traverse all branches by some use cases? Does the outcome of each test case match the specification?

4. Is memory allocated and deallocated properly?
5. Can you traverse the call graph?

As we move down from design to implementation, the details are revealed gradually. In this code reading scenario, we are able to ask concrete questions, e.g., on the memory usage and branch implementations and coverage. This reading scenario is meant for module or unit testers as well as module integration testers.

4.3.4.2 Empirical Experiences

Laitenberger et al. (2001) ran a quasi-experiment and two replicates on professional software developers to compare the performances of perspective-based reading and checklist-based reading of C programming files. It is a quasi-experiment, since they piggybacked experiments on their industrial training opportunities and they didn't have control over all experimental conditions. They used two reading perspectives, analyst and tester. Defects were pooled through defect collection meetings and defects gain or loss effects in meetings were almost non-existing, thus ignored. Meta-analysis was conducted on the three experiments whenever possible. The researchers reported that perspective-based code reading had a slightly higher team defect detection effectiveness than checklist-based reading. The difference was statistically significant. Although individual using perspectives seems to require more effort (time) to detect a defect, the extra effort seems to be justified on the team level, since the team has a better cost per defect. The difference was significant on the team level using meta-analysis. The meeting cost of perspective-based reading was lower. Conjectured reasons include an increased understanding enabled by perspectives. When all costs (individual reading and defect collection meeting) were considered, perspective-based reading had a lower cost per defect and less variability than checklist-based reading.

4.3.5 Perspective-Based Usability Reading

Usability is concerned with how easy it is for a user to perform a desired task and what supports the system provides to the user to accomplish the task. It comprises the following aspects: learning system features, using the system efficiently, minimizing the impact of errors, adapting the system to user needs, and increasing user confidence and satisfaction (Bass et al., 2013). Experiences indicate that focusing on usability is the easiest and cheapest way to improve the user's perceived system quality. In this section we discuss how the ideas of perspective-based reading can be applied to inspect usability of software applications.

4.3.5.1 Reading Scenarios for Usability

Usability problems can be detected with usability inspection or usability testing. In usability testing, usability problems are found through the observation of and interaction with users while they use or comment on the user interface. In usability inspection, problems are found through the expertise of the inspectors and the inspection techniques

they employ. The latter costs much less than the former, and heuristic evaluation, cognitive walkthrough, and guidelines and checklists are common inspection techniques in practice. However, their effectiveness (percent of problems detected) is rather low, particularly for non-experts (Zhang et al., 1999).

To improve the problem of detection effectiveness, Zhang et al. (1999) proposed a perspective-based usability inspection method known as use-based reading. They divided the large variety of usability issues along different perspectives (novice use, expert use, and error handling) and held multiple inspection sessions, each focusing on one particular perspective. Each perspective provides the inspector a point of view, a list of questions to check, which represents typical usability issues related to the perspective, and a specific procedure to perform the inspection. Just as perspective-based reading of other software artifacts, the focused attention, and well-defined procedure improve the inspector's problem detection performance related to the given perspective, and the combination of multiple orthogonal perspectives provides a better coverage of the usability spectrum and uncovers more usability issues.

People use computers to accomplish a task, and the usability or lack of it is exhibited during the human-computer interaction (HCI), which can be abstracted as the repetition of the following sequence of steps: form the goal, form the intention, identify the action, execute the action, perceive the system response, interpret the results, understand the outcome, and deal with errors if any. This sequence of steps can be summarized as execution and evaluation, with possible error correction. Therefore, usability issues are categorized as the gulf of execution (the mismatch between the user's intention and the computer action) and the gulf of evaluation (the mismatch between the user's expectation and system's representation) (Zhang et al., 1999). This simplistic HCI model guides the development of reading perspectives.

Usability perspectives are scenarios of HCI, and different perspectives emphasize different stages in the HCI model or different aspects of the same stage. Based on whether the user knows how to achieve a certain goal and whether the user executes the action correctly, Zhang et al. identified three perspectives: novice use, expert use, and error handling.

For the novice use perspective, the user does not have a deep knowledge regarding how to use the system to achieve his or her goal. Thus the usability goal is for the novice users with minimum knowledge to accomplish basic tasks. For the expert use perspective, the user knows how to use the system but wants to accomplish the task efficiently or easily or to achieve other higher goals. Thus the usability goal is for the expert users to complete tasks efficiently and easily, customize the system to their desire, and use advanced functions or features for improved productivity. For the error handling perspective, the user has a problem with the effect of the previous action and needs to correct the problem. Thus the usability goal is to minimize the chance to commit an error (error prevention), help the user to understand the situation once errors happen, allow the user recover from errors (error recovery), and deal with system failure appropriately. Questions in reading scenarios help the reader decide if the usability goals can be met or not and report usability gaps if not.

Panel 4-7: PBR – Expert Use Reading Scenarios shows the expert use reading scenarios adapted from Zhang et al. (1999). The scenarios are designed specifically for a web-based data collection form application.

Although the exact questions depend on the application being assessed, the intention is similar. The above expert use reading scenario aims at checking the user interface for its efficiency, flexibility, and consistency in supporting the user tasks, as well

as the appropriateness of its visual appearance and organization, which supports the usability goal of the expert use discussed above.

While the perspective-based usability inspection of Zhang et al. partitions reading scenarios from the use point of view (novice, expert, and error handling), there are alternative ways to partition reading scenarios. Conte et al. (2009) decomposed scenarios from design perspectives. They identified conceptual, presentation, and navigation design perspectives for web applications that are used to guide the interpretation of commonly used heuristics. Combinations of the heuristics and design perspectives have yielded a set of specific reading guidance. The researchers also developed and evaluated a tool to support their reading technique (Vaz et al., 2013). Interested readers can refer to their papers for details.

PANEL 4-7: PBR – EXPERT USE READING SCENARIOS

Introduction: Assume you are an expert user. Your goal is to fill out the web form and submit it, which can be decomposed into multiple sub-goals in order. For each sub-goal, go through the following stages and check the questions. If your answer to a question is no, raise a usability issue.

Reading Instruction 1: Scan through instruction, objects, and actions in the user interface.

Question set 1:

1. Is the text easy to read?
2. Is the information organized such that the most important information can be read first?
3. Are the more frequently selected items arranged on top in the list control?
4. Is redundant information avoided or minimized?

Reading Instruction 2: Execute the actions for achieving the sub-goal, using shortcuts whenever available. For each action, check the following questions.

Question set 2:

1. Are common shortcuts available, e.g., to allow user to step to the next text field via keyboard?
2. Are possible default values provided?
3. Does the system compute and remember information for the user whenever possible?

4. Can the user make a selection by clicking on a larger area associated with the object to be selected?
5. Are unproductive activities avoided or minimized, e.g., navigation, mouse movement, hand movement, eye movement, etc.?
6. Are stressful actions avoided or minimized, e.g., clicking mouse multiple times in a short period of time, clicking on a small object to make a selection, etc.?
7. Is the text easy to read?

Reading Instruction 3: Wait for system to respond, if necessary.

Question set 3:

1. Does each user action immediately generate perceivable results in the user interface?

General question: In addition to the specific questions in Question set 3, consider this high-level question: Can the web form be redesigned to reduce the user's unproductive activities?

4.3.5.2 Empirical Experiences

Zhang et al. used professionals (programmers, domain experts, technique researchers, and cognitive researchers) as subjects and compared the use-based reading and the heuristic evaluation in inspecting the usability of a web-based data collection form. Heuristic evaluation is a popular inspection technique in practice. It gives the same general responsibility and provides a list of usability issues to participants, which is similar to ad hoc reading discussed earlier. The heuristics used include: speak the users' language, remain consistency, minimize the users' memory load and fatigue, flexibility and efficiency of use, use visually functional design, design for easy navigation, conduct validation checks, facilitate data entry, and provide sufficient guidance. They reported that perspective-based usability inspection found not only more issues related to their assigned perspectives but also more overall problems. When issues from three perspective readings were aggregated, the use-based reading uncovered about 30% more problems. Assigning inspectors more specific, focused responsibilities leads to improved performance, and combining multiple perspective focused inspections is a good strategy for creating high usability user interfaces.

4.3.6 Why Does Perspective-Based Reading Work?

As discussed earlier, perspective-based reading has been adapted and applied to software artifacts such as requirements specification, design description and graphical models and diagrams, source code and specifications, and user interface usability. Some experiments have been replicated internally and externally. Researchers reported both positive and negative experiences, and sometimes contradicting findings. To navigate through the collection of replicated experiments and associated findings, Ciolkowski (2009) performed a meta-analysis, which proved to be challenging, given the fact that the data were reported differently and important information was sometimes missing. Nevertheless, the author concluded that for requirements documents, perspective-based reading was significantly more effective than ad hoc reading; for design and code documents, perspective-based reading was significantly more effective than checklist-based reading, but for requirements documents, it was less effective.

This section discusses a cognitive analysis that sheds light on what is going on during perspective-based reading and how people's experiences play a role during reading. We also discuss a simple analytical model that provides indicative information when perspective-based reading can outperform other reading techniques.

4.3.6.1 Cognitive Analysis of Perspective-Based Reading

The cognitive analysis from cognitive science has been applied to the understanding of software inspection meetings. Letovsky et al. (1987) videotaped a representative code inspection meeting at IBM and analyzed it in detail. Their observational results suggested that participants were attempting to achieve three goals (clarity, correctness, and consistency) by executing three corresponding behaviors (design reconstruction, mental simulation, and document cross-checking). These categories accounted for 89% of the duration of the session. Robbins and Carver (2009) have applied a similar protocol analysis to perspective-based reading of requirements. Unlike the defect collection meeting, individual's reading of the software document typically happens quietly. To work around that, readers were requested to think aloud, i.e., they just spoke whatever in their mind at the moment during their reading. To facilitate the analysis, verbal reports during reading were coded and categorized.

Perspective-based reading is a complex problem-solving task that is focused on the problem of detecting defects in documents. A simplistic view of the task is illustrated in Figure 4-3, where the sensory input of the documents, including the requirements document, reading scenario description, and worksheet to log defects, goes through the sensory memory that feeds to the short-term memory. An individual's knowledge on the application domain, software and programming experience, the general knowledge on reading techniques, as well as the knowledge on the reading document newly acquired during reading are stored in the long-term memory, which is retrieved and sent to the short-term memory. All problem-solving tasks are performed in the short-term memory and the outcome is formed as a response.

Figure 4-3. *Basic memory model in software reading (Adapted from Robbins & Carver [2009]).*

Verbal reports during reading were audio-recorded. Words or rephrases spoken by readers were captured and sometimes cross-checked with the defect worksheet, which were coded according to the model above:

- Any mentioning of reading of the requirements document, scenario description, and worksheet was coded as stimulus inputs with their respective codes.
- Any mentioning or recalling of previous knowledge or facts on the requirements document, training, perspective reading technique, domain knowledge, software, or programming knowledge was coded as retrievals from long-term memory using their respective codes.
- Any mentioning of combining existing knowledge into a single fact or making assumption on the document was coded as manipulation in short-term memory with their respective codes.
- Any mentioning or action of writing on the worksheet or logging defect was coded as response output with their respective codes.

The researchers ran the experiment using graduate and upper-level undergraduate students as subjects. The requirements document was about a web-based conference paper submission and review system. Only user and tester perspectives were used. The codes were analyzed using a frequency-based method. The authors were able to confirm:

- Most defect identifications were preceded by combining knowledge actions, which corresponds to the "defect trigger."
- The type of defects found by readers was related to the assigned reading perspectives, user, or tester. That is, the perspectives actually lead to detection of perspective-related defects.
- Perspective specific experience significantly affected the perspective-based reading process.

The protocol analysis provides a robust approach to understanding the cognitive mechanisms supporting perspective-based reading. It has a potential to reconcile all seemingly contradictory observations on the reading technique.

4.3.6.2 An Analytical Model

We can use a simple model to analyze when perspective-based reading outperforms other reading techniques such as checklist-based reading. Assume we have two reading scenarios A and B. Referring to Figure 4-4, Scenario A is designed to uncover defects in oval A, and similarly Scenario B is designed to uncover defects in oval B. In terms of defect detection, two reading scenarios have an overlap, which is the common area of ovals A and B. There are A number of defects that are covered by Scenario A, but not B, B number of defects that are covered by Scenario B, but not A, and C number of defects that are covered by both Scenarios A and B. We denote those regions also as A, B, and C. To put it another way, regions A and C are covered by Scenario A and regions B and C are covered by Scenario B. There are A+B+C total number of defects. Further assume there are no other defects and both perspective-based and checklist-based readings are perfect and they can potentially detect all defects existing in the document.

Figure 4-4. *Defects distribution*

We adopt a probabilistic view on defect detection. With Scenario A, defects that Scenario A is targeting will be detected with probability p_A, and other defects that Scenario A is not targeting will be detected with probability q_A. A reasonable assumption is that $p_A>q_A$. Similarly with Scenario B, we have probabilities p_B and q_B. To simplify the matter, we assume $p_A=p_B=p$ and $q_A=q_B=q$.

Assume that defect detections using Scenarios A and B are independent. When pooling defects together via, e.g., a defect collection meeting, there is no meeting gain or loss, which is a reasonable assumption according to Porter et al. (1995) and many others (e.g., Laitenberger et al. [2001]). Also assume that whether individual defects will be detected is independent of each other. A defect in region A will be missed if and only if both Scenarios A and B miss it, the probability of which is (1-p)(1-q). Similarly, a defect in region B will be missed with the same probability (1-p)(1-q). A defect in region C will be missed with probability $(1-p)^2$, however. Therefore, perspective-based reading using Scenarios A and B by a two-reader team will report, on average, D_{pbr} number of defects, as show below:

$$D_{pbr} = A \cdot \left[1-(1-p)(1-q)\right] + B \cdot \left[1-(1-p)(1-q)\right] + C \cdot \left[1-(1-p)^2\right]$$

The first term on the right-hand side of the equation is the number of defects uncovered from region A, the second term is the number of defects uncovered from region B, and the last term is the number of defects uncovered from region C, by Scenarios A and/or B. With simple algebraic operations, this equation can be turned into

$$D_{pbr} = (A+B) \cdot (p+q-pq) + C \cdot (2p-p^2)$$

For checklist-based reading, let's assume there are two independent readers with the same defect detection probability s. It is logical to assume p>s>q. Otherwise there is no point to develop and utilize perspective-based reading. The checklist covers all potential defects. Therefore checklist-based reading from two readers will report, on average, D_{cbr} number of defects:

$$D_{cbr} = (A+B+C) \cdot \left(1-(1-s)^2\right)$$

With simplification, this amounts to

$$D_{cbr} = (A+B+C) \cdot (2s-s)^2$$

We shall find the difference D_{pbr}-D_{cbr}:

$$\Delta = D_{pbr} - D_{cbr} = (A+B) \cdot (p+q-pq-2s-s)^2 + C \cdot (2p-p^2-2s+s^2)$$

Since $2p-p^2$ is an increasing function for p within 0 and 1 and p>s, the second term on the right-hand side is always positive. We shall seek one sufficient condition so that perspective-based reading outperforms checklist-based reading, which is

$$p+q-pq-2s+s^2 \geq 0$$

This is not a necessary condition, since the second term on the right-hand side (i.e. the term involving C) is always positive if p>s, which may offset the first (negative) term (i.e., the term involving A+B) and still make perspective-based reading better.

The sufficient condition can be re-rendered, following the steps below:

$$p-pq-s+sq \geq s-q-s^2+sq$$
$$(p-s)(1-q) \geq\sim (1-s)(s-q)$$
$$p-s \geq \frac{1-s}{1-q}(s-q)$$

This is the same equation as A28 in Laitenberger et al. [2001]. The difference p-s can be considered as the gained efficiency of defect detection for defects covered by reading scenario, and s-q the lost efficiency for defects not covered by reading scenario. That is, when the gained and lost efficiencies satisfy the above relationship because of the scenario focusing, perspective-based reading can outperform checklist-based reading.

We shall note that $\frac{1-s}{1-q} < 1$ when s>q.

Let's use some real numbers to put it into quantitative basis. In Figure 4-5 we plot the relationship between p and q for s=0.4, 0.5, and 0.6. For those s values, the corresponding defect detection effectiveness defined in percentages are 64%, 75%, and 84%, respectively. For a given curve, when p is above the curve line, perspective-based reading will be more effective than checklist-based reading. The smaller q is, the more focused the reading is. The diagram also points the direction to how one can make perspective-based reading more effective. If we suppress the defect detection probability for the non-scenario-targeted defects, we have to increase the detection probability for the scenario-targeted defects and the quantitative relationship of loss and gain is related by the formula given earlier. If one completely misses the defects not targeted by the reading scenario (i.e., q=0), the detection effectiveness of the targeted defect p has to be at least $2s-s^2$ in order to be better than the baseline reading technique.

Figure 4-5. *The p-q phase diagram*

Contradicting experimental results can be understood within this theoretical framework.

1. We first point out that this simple relationship is obtained under the assumption that reading techniques, perspective-based reading, and any other techniques that perspective-based reading is compared to can uncover all defects. In reality, experimental conditions may not satisfy this assumption. We didn't see any analysis to confirm that all defects could be uncovered, in theory, using the investigated reading techniques.
2. In practice, we also find that some defects are easy to uncover, while others are not so. That is, each defect has a different probability to be detected. The analysis assumes a uniform detection probability in the defect groups (the perspective targeted or not targeted, or the entire defects).
3. When designing reading perspectives for their targeted defects, we purposely minimize the defects overlap, i.e., C region in the diagram. This turns out to be a very difficult objective to achieve. There is no discussion from experimental designers regarding how the defect overlapping region looked like in their experiment. The existence of a common region favors perspective-based reading, however (see the second term in D_{pbr}-D_{cbr}).
4. The defect detection probability, p, q, and s, strongly depends on an individual's experience and skill. In all published experiments, there was no explicit estimate of those probabilities. Hence, when perspective-based reading outperforms other reading techniques, we are not clear why and how we attribute its success; when it does not outperform, we don't know why it is so.
5. For future experiments, we suggest researchers collect or estimate those data so that the results can be analyzed quantitatively and the success can be properly attributed to.

Let's examine the common defects detected by two reading perspectives. This is not the common region C in the diagram, as the defects not targeted by a reading scenario are detected at a probability q. In the two-person team, on average, the common defects reported by both team members are, respectively, for perspective-based reading and checklist-based reading:

$$D^c_{pbr} = (A+B)\cdot pq + C\cdot p^2$$
$$D^c_{cbr} = (A+B+C)\cdot s^2$$

With those two quantities, we can rewrite the difference as

$$\Delta = (A+B)\cdot(p+q-2s)+2C\cdot(p-s)+\left(D^{c}_{cbr}-D^{c}_{pbr}\right)$$

In literature it was believed that the effectiveness of perspective-based reading was due to the reduction of overlapping defect detection (i.e. reduced value of D^{c}_{pbr}). We argue this is not necessarily the case. Suppose we have perfectly focused reading scenarios that do not detect any defects the scenarios are not targeting, i.e., q=0. In that case the overlapping defects detected by perspective-based reading is $D^{c}_{pbr}=0$ when C=0 or A+B>>C, and of course, to ensure Δ>0, p has to satisfy some condition, which would be $p>2s-s^2$. Now suppose for a given p and q, two reading techniques give the same performance, i.e., Δ=0. Now holding q constant, we improve the reading scenarios such that all targeted defects are detected with probability p=1. Δ increases as p increases and perspective-based reading outperforms checklist-based reading. Since we fix the value of q and improve the value of p, the overlapping defects detected by perspective-based reading D^{c}_{pbr} increases. In other words, the outperforming perspective-based reading has an increased number of defects detected by both reading scenarios. How the overlapping defects are detected by perspective-based reading is not the driving force if perspective-based reading outperforms the baseline reading technique. Rather it is the consequence of the change in detection capabilities for the defects targeted and not targeted by the perspectives. How the number of detected overlapping defects will change is determined by the relative changes of those detection capabilities.

4.4 Alternative Partitioning of Reading Responsibilities

We have discussed two scenario-based reading techniques, defect-based reading and perspective-based reading. While defect-based reading partitions scenarios based on defect classes, perspective-based reading decomposes scenarios based on stakeholders' perspectives to the documents under review. When a defect or perspective scenario is assigned to a reader, the portion or aspect of the document is assigned to the reader implicitly. There exist other alternatives that partition the reading responsibilities and coordinate reading activities, however.

4.4.1 Ad hoc Partition

A document under review usually has multiple parts. If a template is used to write the document, it might have different chapters and sections. For example, a requirements specification of an IT application may have chapters on web interface, billing, database, etc. Among all those parts, a group of reviewers can decide how to assign different portions to different people. Cheng and Jeffery (1996) called this self-set strategy. The document author or review coordinator can make the assignment as well, based on readers' interests and expertise. Once the document review responsibilities are divided among readers, each reader can use whatever reading techniques they may possess, e.g., ad hoc reading or checklist-based reading. This ad hoc assignment reduces the work

overload to individual readers and provides a specific area of focus for them, compared to the case where the whole document is given to reviewers and the reviewers are simply asked to detect as many defects as they could.

4.4.2 Function Point-Based Partition

Cheng and Jeffery (1996) developed a set of scenarios, function point scenarios, for requirements reading based on function point analysis. Functional user requirements can be categorized into five types: inputs, outputs, inquiries, internal files, and external interfaces. Each of those types can be easily mapped to end-user business function, e.g., an input mapped to a data entry, an inquiry mapped to a user query. Function point analysis is frequently used to estimate the complexity and effort of the software. The authors, however, used the five functional types to dissect the software system into orthogonal areas, which provides the basis for reading scenario construction. Five function point scenarios are:

- *Overview*: This is an overall description of the entire system. The purpose of this reading scenario is to check the overall relevance, completeness, and correctness of a requirements specification.
- *File*: This combines the internal files and external interfaces functional types, given that they have similar characteristics with respect to the requirements review. This contains the attributes for the purpose of the application.
- *Input*: This is a process that requests the inputs from a user or another system. The input may come as screen input or batch input.
- *Output*: This is the opposite as the input and generates information for a user or another system, which can be in a form of report, or data transferred to other processes.
- *Inquiry*: A process that retrieves data as a response to an input from a user or a process, and outputs the data to a user or another system. Data update is not involved in this process.

Key issues covered by each function point analysis are constructed into domain-specific questions, which are augmented by available checklist items. Those questions are used directly in a scenario description, which also provides instruction to focus the reader on specific areas of the requirements document related to the scenario/functional type.

Cheng and Jeffery used senior undergraduate and graduate students with industrial experience as subjects. They compared the defect detection performances of self-set strategy and function point scenarios and reported that the self-set strategy group detected more defects. The difference was not statistically significant though. They further reported that there was no significant difference in defect types either. While the reading using the self-set strategy showed a dependency on the individual experience, the same dependence was not observed in the subjects using the function point scenarios. The researchers could not tell if function point scenario-based reading helped the inexperienced readers or inhibited the experienced readers.

4.5 Summary

We have discussed three scenario-based reading techniques, defect-based reading, perspective-based reading, and function-point-based reading. The first and last reading techniques have been applied to requirements specification documents. Perspective-based reading has been customized to requirements, design, code, and usability reading and inspection. Due to experimental variabilities and human nature, both positive and negative findings were reported. We discussed a cognitive analysis that shed light on what is going on during perspective-based reading and analyzed a simple model to suggest when perspective-based reading can outperform other reading techniques. Perspective-based reading started to see some industrial adoption (Lahtinen, 2011). Later chapters will discuss additional scenario-based reading techniques, e.g., traceability-based reading.

4.6 References

(Basili, 1996) V.R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sorumgard, and M.V. Zelkowitz, The empirical investigation of perspective-based reading, Empirical Software Engineering, vol.1, no.2, pp.133-164, 1996.

(Bass, 2013) L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 3rd ed., Addison-Wesley, 2013.

(Bisant, 1989) D. Bisant and J. Lyle, A two-person inspection method to improve programming productivity, IEEE Transactions on Software Engineering, vol.15, no.10, pp.1294-1304, 1989.

(Chen, 2002) T.Y Chen, P.L. Poon, S.F. Tang, T.H. Tse, and Y.T. Yu, Towards a problem-driven approach to perspective-based reading, Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering, pp.221-229, 2002.

(Cheng, 1996) B. Cheng and R. Jeffery, Comparing inspection strategies for software requirement specifications, Proceedings of Australian Software Engineering Conference, pp.201-211, 1996.

(Ciolkowski, 1997) M. Ciolkowski, C. Differding, O. Laitenberger, and J. Munch, Empirical investigation of perspective-based reading: A replicated experiment, Technical Report No.13, International Software Engineering Research Network, 1997.

(Ciolkowski, 2003) M. Ciolkowski, O. Laitenberger, and S. Biffl, Software reviews, the state of the practice, IEEE Software, vol.20, no.6, pp.46-51, 2003.

(Ciolkowski, 2009) M. Ciolkowski, What do we know about perspective-based reading? An approach for quantitative aggregation in software engineering, 3rd International Symposium on Empirical Software Engineering and Measurement, pp.133-144, 2009.

(Conte, 2009) T. Conte, J. Massolar, E. Mendes, and G.H. Travassos, Web usability inspection technique based on design perspectives, IET Software, vol.3, no.2, pp.106-123, 2009.

(Fagan, 1976) M.E. Fagan, Design and code inspections to reduce errors in program development, IBM Systems Journal, vol.15, no.3, pp.182-211, 1976.

(Fowler, 1986) P.J. Fowler, In-process inspections of workproducts at AT&T, AT&T Technical Journal, vol.65, no.2, pp.102-112, 1986. Also in Software Inspection: An Industry Best Practice, D.A. Wheeler, B. Brykczynski, and R.N. Meeson Jr., IEEE Computer Society Press, 1996.

(Fusaro, 1997) P. Fusaro, F. Lanubile, and G. Visagggio, A replicated experiment to assess requirements inspection techniques, Empirical Software Engineering, vol.2, no.1, pp.39-57, 1997.

(Gilb, 1993) T. Gilb and D. Graham, Software Inspection, Addison-Wesley, 1993.

(Graden, 1986) M.E. Graden, P.S. Horsley, and T.C. Pingel, The effects of software inspections on a major telecommunication project, AT&T Technical Journal, vol.65, no.3, pp.32-40, 1986. Also in Software Inspection: An Industry Best Practice, D.A. Wheeler, B. Brykczynski, and R.N. Meeson Jr., IEEE Computer Society Press, 1996.

(Hayes, 1999) W. Hayes, Research synthesis in software engineering: a case for meta-analysis, Proc, of the 6th International Software Metrics Symposium, pp.143-151, 1999.

(IEEE, 2008) IEEE, IEEE Standard for Software Reviews and Audits, IEEE Std 1028-2008.

(Lahtinen, 2011) J. Lahtinen, Application of the perspective-based reading technique in the nuclear I&C context, CORSICA work report, 2011.

(Laitenberger, 1997) O. Laitenberger and J.M. DeBaud, Perspective-based reading of code documents at Robert Bosch GmbH, Information and Software Technology, vol.39, no.11, pp.781-791, 1997.

(Laitenberger, 1999) O. Laitenberger and C. Atkinson, Generalizing perspective-based inspection to handle object-oriented development artifacts, Proceedings of the 21st International Conference on Software Engineering, pp.494-503, 1999.

(Laitenberger, 2000) O. Laitenberger, C. Atkinson, M. Schlich, K. El Emam, An experimental comparison of reading techniques for defect detection in UML design documents, Journal of Systems and Software, vol.53, no.2, pp.183-204, 2000.

(Laitenberger, 2001) O. Laitenberger, K. El Emam, and T.G. Harbich, An internally replicated quasi-experimental comparison of checklist and perspective-based reading of code documents, IEEE Transactions on Software Engineering, vol.27, no.5, pp.387-421, 2001.

(Letovsky, 1987) S. Letovsky, J. Pinto, R. Lampert, and E. Soloway, A cognitive analysis of a code inspection, in Empirical Studies of Programming, Chapter 15, G. Olson, S. Sheppard, E. Soloway, eds., pp.231-247, 1987. Also in Software Inspection: An Industry Best Practice, D.A. Wheeler, B. Brykczynski, and R.N. Meeson Jr., IEEE Computer Society Press, 1996.

(Maldonado, 2006) J.C. Maldonado, J. Carver, F. Shull, S. Fabbri, E. Doria, L. Martimiano, M. Mendonca, and V. Basili, Perspective-based reading: A replicated experiment focused on individual reviewer effectiveness, Empirical Software Engineering, vol.11, pp.119-142, 2006.

(Miller, 1998) J. Miller, M. Wood, and M. Roper, Further experiences with scenarios and checklists, Empirical Software Engineering, vol.3, no.1, pp.37-64, 1998.

(Parnas, 1985) D.L. Parnas and D.M. Weiss, Active design reviews: principles and practices, 8th International conference on Software Engineering, pp.215-222, 1985.

(Porter, 1994) A.A. Porter and L.G. Votta, An experiment to assess different defect detection methods for software requirements inspections, in Proceedings of the 16th International Conference on Software Engineering, pp.103-112, 1994.

(Porter, 1995) A.A. Porter, L.G. Votta, and V.R. Basili, Comparing detection methods for software requirements inspection: A replicated experiment, IEEE Transactions on Software Engineering, vol.21, no.6, pp.563-575, 1995.

(Porter, 1997) A.A. Porter, H.P. Siy, C.A. Toman, and L.G. Votta, An experiment to assess the cost-benefits of code inspection in large scale software development, IEEE Transactions on Software Engineering, vol.3, no.6, pp.329-346, 1997.

(Porter, 1998) A.A. Porter and L.G. Votta, Comparing detection methods for software requirements inspection: A replication using professional subjects, Empirical Software Engineering, vol.3, no.4, pp.355-379, 1998.

(Regnell, 2000) B. Regnell, P. Runeson, and T. Thelin, Are the perspective really different? – Further experimentation on scenario-based reading of requirements, Empirical Software Engineering, vol.5, no.4, pp.331-355, 2000.

(Robbins, 2009) B. Robbins and J. Carver, Cognitive factors in perspective-based reading (PBR): A protocol analysis study, 3rd International Symposium on Empirical Software Engineering and Measurement, pp.145-155, 2009.

(Sandahl, 1998) K. Sandahl, O. Blomkvist, J. Karlsson, C. Krysander, M. Lindvall, and N. Ohlsson, An extended replication of an experiment for assessing methods for software requirements inspection, Empirical Software Engineering, vol.3, no.4, pp.327-354, 1998.

(Sabaliauskaite, 2003) G. Sabaliauskaite, F. Matsukawa, S. Kusumoto, and K. Inoue, Further investigations of reading techniques for object-oriented design inspection, Information and Software Technology, vol.45, no.9, pp.571-585, 2003.

(Schulte, 2010) C. Schulte, T. Busjahn, T. Clear, J.H. Paterson, and A. Taherkhani, An introduction to program comprehension from computer science educators, Innovation and Technology in Computer Science Education Working Group Reports ITiCSE-WGR'10, pp.65-86, 2010.

(Shull, 2000) F. Shull, I. Rus, and V. Basili, How perspective-based reading can improve requirements inspections, IEEE Computer, vol.33, no,7, pp.73-79, 2000.

(Vaz, 2013) V.T. Vaz, T. Conte, G.H. Travassos, Empirical assessments of a tool to support web usability inspection, CLEI Electronic Journal vol.16 no.3, paper 6, 2013.

(Walia, 2009) G.S. Walia and J.C. Carver, A systematic literature review to identify and classify software requirement errors, Information and Software Technology, vol.51, pp.1087-1109, 2009.

(Zhang, 1999) Z. Zhang, V.R. Basili, B. Schneiderman, Perspective-based usability inspection: an empirical validation of efficacy, Empirical Software Engineering, vol.4, no.1, pp.43-69, 1999.

CHAPTER 5

Requirements Reading Techniques

The general reading techniques, ad hoc reading, checklist-based reading, defect-based reading, and perspective-based reading, have all been applied to inspect software requirements specifications. A detailed account of these techniques and their empirical experiences can be found in Chapters 3 and 4. Here we discuss additional reading techniques that are applicable specifically to requirements specifications. We also discuss what factors impact the effectiveness of requirements reading, which can be used to guide reading team selection and training.

5.1 Critical Roles of Requirements in Software Development

A requirements specification is an important document generated in the early stages of a software development project. It defines the functionality, scope, and constraints of the software system. It can also serve as a basis for contract negotiations or communication between the software development organization and end users.

The importance of the requirements specification cannot be emphasized enough. If the development is based on an incomplete and incorrect requirements specification, then the finished software product will not fulfill users' needs. Defects in a vague or ambiguous specification can be propagated down to subsequent development phases such as design and/or coding. Designers or developers would potentially still be able to catch these problems, but it will be at the expense of schedule delays, which can be costly. Even worse, if the defects remain undetected and a faulty software product is delivered to users, the users' working environment could be damaged. So it is paramount to read, detect, and correct requirements issues early in the development process.

© Yang-Ming Zhu 2016

Y.-M. Zhu, *Software Reading Techniques*, DOI 10.1007/978-1-4842-2346-8_5

5.2 A Combined Reading Technique for Requirements

The number and types of defects each reading technique can detect vary from one technique to another. A combined-reading (CR) technique is intended to take advantage of the strengths of individual reading techniques while compensating for their weaknesses.

5.2.1 Motivations for a Combined Reading

Checklist-based reading (CBR), defect-based reading (DBR), and perspective-based reading (PBR) techniques have all been applied to software requirements specifications. Maldonado et al. (2006) reported that PBR and CBR are complementary to each other for reading some requirements specifications. Alshazly et al. (2014) compared the three reading techniques and reported that, in terms of total number of defects detected, PBR > DBR > CBR (">" means "is better than"); in terms of ambiguous and omission defect detection capability, PBR > DBR > CBR; in terms of inconsistent defect detection capability, PBR > DBR ≈ CBR ("≈" means "roughly the same as"); in terms of incorrect defect detection capability, DBR > CBR > PBR; and in terms of superfluous defect detection capability, CBR = DBR > PBR. Each reading technique has its own strength and weakness.

Thus it is difficult to detect most or all defects using a single reading technique alone. An apparent solution is to combine multiple reading techniques, which exploits the advantages of each reading technique and avoids its limitations at the same time.

5.2.2 The Combined-Reading Technique

A software requirements specification document has multiple constituent parts. For example, IEEE STD 830-1998 has a recommended structure for a software requirements specification (this recommendation has been superseded by ISO/IEC/IEEE 29148:2011). Not all classes of defects could occur in all parts of the document, and there is only a limited number of defect classes that could appear in a given part. The main idea behind CR is to examine every constituent parts of the software requirements specification in detail. For software organizations, the historical defect patterns and root causes can be analyzed to determine what classes of defects can occur in what parts of the document and how the defects are introduced. After that, specific questions can be developed for each part of the document to cover different types of defects from the perspectives of different stakeholders, taking into account defect history and the purpose of the individual parts of the requirements specification.

The influences of CBR, DBR, and PBR are apparent. Combined reading is based on defect class analysis, and specific questions are designed for it (DBR). The questions are designed to cover the defects from the perspectives of different stakeholders (PBR). The guidance is constructed as a list of yes/no questions (CBR).

An excerpt of the reading guidance is shown in Table 5-1, where checked items are the areas of concern, defect types are typical types of defects the organization may have, defect sources list the main and sub-error sources, and the checklist items are the questions for the reader to answer throughout the course of reading.

Table 5-1. *An Excerpt Guidance of the Combined Reading Technique (adapted from Alshazly et al. [2014])*

Q	Checked item	Defect type	Defect source		Checklist item
			Main	Sub	
1	External interfaces	Omission	Process	Analysis	Have all inputs and outputs of the system been described in detail?
2	User interfaces	Omission	People	Concentration	Have all interfaces between the software product and its users been specified?
3	Constraints	Omission	Process	Analysis	Do all significant consumers have scarce resources, such as memory, network bandwidth, processor capacity, etc. identified? Has the anticipated consumption of resources been specified?
4	Use-case name and number	Ambiguous	People	Concentration	Does the use-case name reflect its goal?
5		Inconsistent	Documentation	Organization	Are the requirements arranged numerically according to the logical order of occurrence and in order to prevent confusion between the requirements?
6		Inconsistent	Process	Traceability	Is there a conflict between the functional requirements names and the relevant use-cases names?
…					
68	Non-functional requirements	Omission	Process	Elicitation	Are the necessary non-functional requirements specified, including reliability, availability, security, maintainability, and portability?
69	General questions	Inconsistent	Documentation	Documentation standards	Did the author of the document use numbering levels to identify sections of the document and the functional requirements?

The CR technique guides the reader through what is being inspected and provides instruction on how the inspection should be conducted. The required training is minimal and a novice reader can benefit from it. As the empirical evidence shows, it is more effective than CBR, DBR, and PBR.

An apparent limitation of the CR technique is that the list of questions is relatively large (69 questions in total). It could be overwhelming to a single reader and would be more manageable if the questions were divided among multiple readers. The other drawback is, just like the checklist, it is effective for known defects and defect types but might not be so for defects not previously seen.

5.2.3 Empirical Experiences

Alshazly et al. (2014) applied the CR technique to four requirements specification documents and compared its defect detection effectiveness with those of CBR, DBR, and PBR in case studies. In terms of the total number of defects detected, CR > PBR > DBR > CBR; in terms of ambiguous defect detection capability, CR > CBR > DBR > PBR; in terms of inconsistent defect detection capability, CR > PBR > DBR ≈ CBR; in terms of incorrect defect detection capability, CR > DBR ≈ CBR > PBR; in terms of omission defect detection capability, CR > PBR > DBR > CBR; in terms of superfluous defect detection capability, CR > CBR ≈ DBR ≈ PBR; and in terms of non-conforming to standard defect detection capability, CR ≈ CBR ≈ DBR ≈ PBR. These findings have not been confirmed by independently replicated experiments yet.

5.3 Test-Case Driven Reading for Requirements

Most of the time when we read requirements, we detect and remove defects for a full-fledged project to avoid propagating issues down to subsequent development phases. The requirements document is typically inspected by multiple stakeholders. There are other situations in which we inspect the requirements to ensure that we have good enough requirements to support decision making during planning and before the project is kicked off (pre-project), e.g., what requirements to select for realization and cost estimation. Companies operating in a market-driven environment typically have a large amount of requirements coming from different sources with different levels of quality. With limited resources, they want to inspect the requirements in a lean manner and amortize the effort across different development phases, since they have many more pre-project requirements than their development organization can handle. Gorschek and Fogelstrom (2005) and Fogelstrom and Gorschek (2007) developed a test-case-driven requirements reading to satisfy the above needs. They proposed an inspection process in which the test-case-driven reading is embedded. Here we focus on the reading technique.

5.3.1 Test-Case-Driven Reading Technique

The goal of the test-case-driven reading technique is to enable the software development organization to inspect the requirements effectively with minimal cost. It utilizes the test-cases as a tool for inspection and involves real testers at an early stage of the pre-

project decision making. The test- cases developed during reading can be used in latter development phases, and thus the cost and time demand in the early requirement phase is spread along the project phases.

The tester ensures the testability, completeness, and non-conflicting aspects of those requirements. If the tester can create test-cases based on a requirement, then the requirement is considered testable. Since the tester is an expert in the use of the system, any missing or incomplete functionality can be easily detected. This provides another angle for the tester to examine the system, from the perspective of the end-user. When the tester goes through the set of requirements thoroughly by creating test-cases along the way, any conflicting and inconsistent requirements can be caught.

We shall contrast test-case-driven reading with perspective-based reading to further understand the test-case-driven reading technique:

- To support PBR, reading scenarios, including the tester's reading scenario, must be developed and maintained from different stakeholders' perspectives. In test-case-driven reading, no reading scenario is required since the real testers are employed.
- In the tester's PBR, individuals such as requirements engineers or developers—not necessarily the real testers—are employed to simulate the tester's perspective. Thus education, training, and practice will be required for these readers. In test-case-driven reading, the experienced testers are doing the reading, which not only represents the tester's perspective better than the simulated one, but also alleviates the education and training costs. This might also utilize resources better, since typically, the testers could have a light workload during the requirements definition phase if there are no other active projects.
- In the tester's PBR, the reader develops test cases. Since these test cases are developed by non-testers, it is always in question whether these test cases can be reused in a later phase. In test-case-driven reading, the tester's competence and efforts are exploited to develop test-cases, which serves the inspection for now, and could also be reused in later development and testing phases. The test-cases can be used to augment the requirements so that developers understand the requirements better, which reduces the chance of misinterpreting the requirements.
- Perspective-based reading typically takes advantages of the perspectives of multiple stakeholders, with each reader representing one perspective. However, test-case-driven reading utilizes one tester who brings two perspective: tester and end-user. The tester's perspective is an obvious one. The tester is familiar with the real usage of the system. He or she is thus a system functionality expert and an expert at reading and interpreting the requirements.

5.3.2 Empirical Experiences

Gorschek and Fogelstrom (2005) conducted a pilot study and concluded that the benefit of test-case-driven requirements reading was substantial based on the subjective feedback from people who were involved. Product managers felt the inspected and reworked requirements were of high quality, offered better support for decision making, prevented unviable requirements from passing down to the next step, and improved the product manager's requirements specification skills. The testers felt it enabled them to create a realistic test plan early. However, they also reported that non-functional requirements were hard to inspect due to potential conflicts among different quality attributes.

In a follow-up study, Fogelstrom and Gorschek (2007) compared test-case-driven requirements reading to CBR, using software engineering graduate students as subjects. They reported that CBR found a high amount of false positive defects, but fewer major defects. Conflicting requirements, missing requirements, and missing or wrong information in requirements are considered major. When only major defects were considered, test-case-driven reading is 44% to 55% more effective than CBR, where the effectiveness is defined as the percentage of major defects uncovered among the total major defects. When efficiency is concerned (number of major defects uncovered per hour), the result is not conclusive. In one case, test-case-driven reading is more efficient, and in the other case, CBR is more efficient, but the difference is not significant in both cases.

5.4 Individual Factors Impacting Requirements Reading Efficiency

Researchers have reported that individual defect detection performance can vary by a factor of 10 in terms of defects found per unit time (Hatton, 2008). This wide variation in the defect detection effectiveness (total number of defects or the percentage of defects detected) is also reported, even when readers are using the same technique and same process on the same artifact. For example, Basili et al. (1996) reported that the effectiveness of individual readers ranged from 10% to 90% when applying PBR on requirements specifications. Similar variation exhibits at the team level as well (Schneider et al., 1992). We often see different, sometimes contradicting, empirical findings in replicated experiments (Hayes, 1999; Ciolkowski, 2009). People's backgrounds and experiences play a significant role during software reading and analysis. It is thus prudent to study what factors impact the effectiveness of requirements reading.

Miller and Yin (2004) studied a characteristic of readers not related to software engineering expertise, the Myers-Briggs personality type. Miller and Yin reported that personality type was not a good predictor of the individual effectiveness of requirements reading. However, a team formed by people of different personalities with diverse information processing strategies maximizes the number of different defects detected.

Biffl and Halling (2002) studied the characteristics directly related to software engineering expertise, development skills, quality assurance experience, and performance on a pre-test with a mini-inspection. Biffl and Halling concluded that only readers' performance on a pre-test of mini-inspection, not their development and quality assurance capability and experience, was considerably correlated to reading performance. Two other software engineering characteristics, IT experience and

data-flow diagram experience, were investigated by Hungerford et al. (2004) with experienced software developers. Although the researchers did not find that these characteristics affect reading performance, they reported that the ability to rapidly switch between diagrams impacted the reading performance.

Carver et al. (2008) studied the impact of educational background (field of study), education degree, and requirements writing experience on requirements defect detection effectiveness using professionals as subjects. The researchers reported that readers with an education background not related to computing such as engineering, math, science, business, or the arts, were significantly more effective in detecting omission and inconsistency defects. Upon further analysis, readers with computer science or software engineering background were the least effective. Readers with requirements writing experience were significantly more effective than those without, regardless of whether the experience was gained through industry practice or classroom learning. Their effectiveness was not related to particular defect categories. Education degree was not found to be significantly related to defect detection performance. The experiment of Carver et al. was replicated by Albayrak and Carver (2014) with industry practitioners in a different country. The original findings were confirmed in the replicated study.

The reason people with computing-related education backgrounds have poor defect detection effectiveness is not well understood. One possible reason is that people with computing backgrounds tend to think in terms of design and coding immediately upon reading requirements. Meanwhile, people with non-computing backgrounds may approach requirements as a user, rather than designer or implementer. Another possible explanation might be that precise writing was part of training in non-computing fields.

The research findings discussed earlier have interesting implications to software inspection practice in industry. When selecting people to inspect software artifacts, it is a wise decision to select people with different education backgrounds, particularly those with non-computing-related disciplines. If defect collection meetings will be held, people with different personality types should be chosen as well. When inspecting software requirements specification documents, it is good to include people with requirements writing experience.

5.5 Summary

Software requirements specifications play a critical role during the software development process and it is important to get them right as early as possible. Ad hoc reading and CBR are mostly used in practice. Scenario-based reading, DBR, and PBR in particular, have been applied to requirements reading with reported success. This chapter discusses a few newly proposed reading techniques: CR, which combines the benefits of the CBR, DBR, and PBR techniques, and test-case-driven reading, which is applicable to organizations with limited resources and can provide good-enough requirements for project decision making early in the software development lifecycle. It is known that individuals with different backgrounds exhibit a varying performance in software reading. Factors that impact requirements reading performance are discussed, which can be utilized when assembling and training a requirements inspection team.

5.6 References

(Albayrak, 2014) O. Albayrak and J.C. Carver, Investigation of individual factors impacting the effectiveness of requirements inspections: a replicated experiment, Empirical Software Engineering, vol.19, pp.241-266, 2014.

(Alshazly, 2014) A.A. Alshazly, A.M. Elfatatry, and M.S. Abougabal, Detecting defects in software requirements specification, Alexandria Engineering Journal, vol.53, pp.513-527, 2014.

(Basili, 1996) V.R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sorumgard, and M.V. Zelkowitz, The empirical investigation of perspective-based reading, Empirical Software Engineering, vol.1, no.2, pp.133-164, 1996.

(Biffl, 2002) S. Biffl and M. Halling, Investigating the influence of inspector capability factors with four inspection techniques on inspection performance, Proceedings of 8th IEEE Symposium on Software Metrics, pp.107-117, 2002.

(Carver, 2008) J.C. Carver, N. Nagappan, and A. Page, The impact of educational background on the effectiveness of requirements inspections: an empirical study, IEEE Transactions on Software Engineering, vol.34, no.6, pp.800-812, 2008.

(Ciolkowski, 2009) M. Ciolkowski, What do we know about perspective-based reading? An approach for quantitative aggregation in software engineering, 3rd International Symposium on Empirical Software Engineering and Measurement, pp.133-144, 2009.

(Fogelstrom, 2007) N.D. Fogelstrom and T. Gorschek, Test-case driven versus checklist-based inspections of software requirements – An experimental evaluation, Workshop emEngenharia de Requisitos, pp.116-126, 2007.

(Gorschek, 2005) T. Gorschek and N.D. Fogelstrom, Test-case driven inspections of pre-project requirements – Process proposal and industry experience report, Proceedings of the Requirements Engineering Decision Support Workshop held in conjunction with the 13th IEEE International Conference on Requirements Engineering, 2005.

(Hatton, 2008) L. Hatton, Testing the value of checklists in code inspections, IEEE Software, vol.25, no.4, pp.82-88, 2008.

(Hayes, 1999) W. Hayes, Research synthesis in software engineering: a case for meta-analysis, Proc, of the 6th International Software Metrics Symposium, pp.143-151, 1999.

(Hungerford, 2004) B.C. Hungerford, A.R. Hevner, and R.W. Collins, Reviewing software diagrams: A cognitive study, IEEE Transactions on Software Engineering, vol.30, no.2, pp.82-96, 2004.

(Maldonado, 2006) J.C. Maldonado, J. Carver, F. Shull, S. Fabbri, E. Doria, L. Martimiano, M. Mendonca, and V. Basili, Perspective-based reading: A replicated experiment focused on individual reviewer effectiveness, Empirical Software Engineering, vol.11, pp.119-142, 2006.

(Miller, 2004) J. Miller and Z. Yin, A cognitive-based mechanism for constructing software inspection teams, IEEE Transactions on Software Engineering, vol.30, no.11, pp.811-825, 2004.

(Schneider, 1992) G.M. Schneider, J. Martin, and W.T. Tsai, An experimental study of fault detection in user requirements documents, ACM Transactions on Software Engineering and Methodology, vol.1, no.2, pp.188-204, 1992.

CHAPTER 6

Design Reading Techniques

The general reading techniques discussed in Chapters 3 and 4, ad hoc reading, checklist-based reading, and perspective-based reading, have all been applied to inspect design documents. This chapter discusses a few more reading techniques applicable specifically to design artifacts, namely usage-based reading, traceability-based reading, and scope-based reading.

6.1 Introduction

In a software development process, the development of software architecture and design is between requirements engineering and code implementation. Thus software design needs to satisfy a requirements specification and provide guidance to subsequent implementation. As far as software design inspection is considered, a reasonable assumption is that the requirements specification as input is free of error. Of course, any issues with the requirements specification uncovered during design inspection shall be captured and dealt with. The focus of design reading and inspection is, however, to make sure the design is correct and complete with respect to requirements and the design itself is consistent and clear and has adequate information to start the subsequent detailed design and implementation. Design reading or inspection is crucial since the design quality directly affects the quality of and effort required for the implementation.

6.2 Usage-Based Reading

One of the reasons to read software artifacts is to detect defects early in a development process. However, not all defects have the same impact on end users. Thus the goal of software reading should not be to find as many defects as possible but to find the most critical defects that matter to users. Usage-based reading is developed just for that. A similar idea is employed in the operational profile-based software testing, which greatly improves software reliability (Musa, 1993).

© Yang-Ming Zhu 2016

Y.-M. Zhu, *Software Reading Techniques*, DOI 10.1007/978-1-4842-2346-8_6

6.2.1 Usage-Based Reading Technique

The main idea of usage-based reading is to use prioritized use cases to drive and focus an inspection effort on detecting critical defects that matter most to end users. Use cases are generally developed as specifications of a system. A use case is a set of scenarios together satisfying a common user goal, where a scenario is a sequence of steps describing an interaction between an agent (user) and the software system. Usage-based reading focuses on software quality from a user's perspective. It was suggested by Wohlin and Olsson, and further developed by Thelin et al. (2004).

The usage-based reading is schematically illustrated in Figure 6-1, where the artifact to be read and the list of prioritized use cases are input, and the list of defects are the outcome of the reading. Depending on how many use cases a system supports, prioritization can be done in an ad hoc way or systematically through, e.g., a pairwise comparison using the analytic hierarchy process (Saaty & Vargas, 2001). It is important to note that prioritization shall be done by perspective users or someone who is familiar with the usage of the system.

Figure 6-1. *Illustration of the usage-based reading technique*

Two schemes, time-boxed and rank-based, can be employed in usage-based reading. In the time-boxed reading, one allocates a fixed amount of time to the whole reading and divides the allocated time to individual use cases proportionally by their priority weights. Thus an important use case will be allotted more time for a close examination. Alternatively, a rank-based reading can be performed. That is, use cases are examined in their priority order and there is no forced time limit except the whole allocated time. The basic steps of a rank-based reading are listed in Panel 6-1: Instructions for Usage-Based Reading (Rank-Based).

PANEL 6-1: INSTRUCTIONS FOR USAGE-BASED READING (RANK-BASED)

1. Prioritize and sort use cases in order of importance from end users' point of view.
2. Select a use case with the highest priority.
3. Trace and manually execute the use case scenarios through the document under inspection.
4. Ensure the document fulfills the use case goals (the needed functionality is provided, the interfaces are correct, etc.). Identify and report issues found.
5. Select the next use case with the highest priority among the remaining ones and repeat steps 3 and 4, until allocated time is used up or use cases have been exhausted.

Use-case is the common instrument that helps to focus on users' needs. In usage-based reading, prioritized use cases are used to focus the defect detection effort. Use cases are used or involved in other reading techniques as well. For example:

- Perspective-based reading can have a user's perspective, where a reader, who takes the end user's perspective, creates a user's manual or use cases during reading. This is discussed in Chapter 4.
- A reader performs a vertical reading, which is a technique in the traceability-based reading family, and compares design diagrams with use cases to ensure the design is correct and complete. Traceability-based reading is discussed in the next section.
- Use cases are also used in object-oriented (OO) code reading to understand dynamic aspects of an OO system. Use-case-driven reading is discussed in Chapter 7.

The predominant application of usage-based reading is design reading, although it has been applied to requirements documents as well (see the empirical experiences below). It is our opinion that it is a bit awkward to apply usage-based reading to requirements, however. A textual description of requirements is typically developed first, followed by a use-case document. Upstream documents are frequently used to guide the development and review of downstream documents. When used as guidance, upstream documents are assumed to be correct. So employing use cases to guide requirements inspection has a limited value in practice.

Prioritized use cases can be used to direct code reading, e.g., which areas readers focus on and put more effort in. Once a piece of code is selected, a reader tends to have a local view of the code and its functionality implemented, not a global view such as the priority order of use cases. The sheer amount of code in a project is typically much more than an individual can deal with in a reasonable time. It is thus unproductive to apply usage-based reading to code documents. We therefore discuss usage-based reading as a specialized reading technique for design documents.

6.2.2 Variations of Usage-Based Reading

In the usage-based reading developed by Thelin et al. (2004), use case prioritization is performed by an expert of the end system. Winkler et al. (2005), however, suggested that use case prioritization can also be done by readers individually when reading software artifacts, based on one's own understanding of the application domain.

Cantone et al. (2003) reported a use-case-driven reading technique for analysis and design of Unified Modeling Language (UML) diagrams. This is not the same reading technique as that for OO code reading (see Chapter 7), although they have the same name; rather, it is closely related to usage-based reading with the exception that use cases are not prioritized. Analysis and design artifacts under review include a vision document, a use case diagram, a view of participating classes, and sequence diagrams. Four checklists were developed, one for each artifact type, that specify what issues to look for. In addition to checklists, the other idea behind the use-case-driven reading is to specify how to find defects in artifacts ranging from the informal specifications down to use-case realizations and sequence diagrams, using use cases as a guiding light. Their procedure guides a reader to examine the entire set of artifacts, following the order in which they are created using a development process, such as the Rational Unified Process. The authors compared the reading technique with a checklist-based one and concluded that checklist-based reading was more effective. In terms of defect detection rate, checklist-based reading peaked later but at a higher value. The authors indicated that the difference was not statistically significant, however.

6.2.3 Empirical Experiences

Usage-based reading has been tested on design documents in academic settings where subjects were software engineering students (Thelin et al., 2001; 2003; 2004). The effect of use case prioritization in the usage-based reading was assessed first (Thelin et al., 2001). Defects were classified as critical, major, and minor. Thelin et al. reported that readers found different defects using prioritized use cases compared to randomly ordered use cases. Readers who used prioritized use cases detected more critical and major defects than readers who used randomly ordered use cases, and the difference was significant. When all defect categories (i.e., critical, major, and minor) were considered, the difference was not significant anymore, which suggested that prioritized use cases enable readers to detect important defects from the users' point of view. Readers using prioritized use cases were also more efficient, i.e., they detected more defects per hour in areas of all defects, critical defects, and critical and major defects.

Usage-based reading (with prioritized use cases) was compared to checklist-based reading (Thelin et al., 2003). Researchers concluded that usage-based reading was more effective and efficient than a checklist-based one. Readers using usage-based reading detected 75% more critical defects than readers using checklist-based reading, 51% more critical and major defects, but were not significantly more effective in detecting all defects. Readers using usage-based reading detected 35% more total defects per hour than readers using checklist-based reading, 95% more critical defects, and 70% more critical and major defects. Readers using usage-based reading found different and more unique defects, and they started to find defects earlier as well. These findings were confirmed in a replicated experiment (Thelin et al., 2004a).

In their experiment, Thelin et al. (2003) didn't use any active guidance in checklist-based reading. As discussed in Chapter 3, it is possible to incorporate an active guidance in checklist-based reading. Winkler et al. (2005) extended the traditional checklist-based reading by providing an active guidance in design inspection. In particular, readers were instructed to prioritize use cases according to their knowledge of the application domain, and use the prioritized use cases during reading. The difference is that, in usage-based reading, use cases are prioritized by experts. They further compared usage-based reading with the traditional checklist-based reading and its variant with an active guidance and concluded that (1) checklist-based reading with an active guidance was significantly more effective and efficient than the traditional checklist-based reading in finding major defects, and (2) usage-based reading was more effective and efficient than both types of checklist-based reading.

It takes time to develop detailed use cases. Use cases themselves shall be reviewed for accuracy and consistency. Thus use cases shall just have enough details to serve the purpose of usage-based reading. Thelin et al. (2004) set forth to investigate if usage-based reading was still efficient and effective if use cases were prioritized, but without details except only their purposes. Readers were actively developing use cases to fill in more details during reading. Not surprisingly, the researchers reported that reading with full use cases was more efficient and more effective than reading with incomplete use cases. Reading with pre-developed use cases focused readers on detecting defects, while reading with incomplete use cases forced readers to develop a detailed understanding of the documents, enabling them to detect different kinds of defects. Should an organization develop detailed use cases for usage-based reading beforehand or on the fly during reading? Considering the total effort in developing use cases, the authors suggested not to spend time on developing use cases if these use cases are not used in the application development. The authors further suggested a hybrid approach, i.e., develop complete use cases if they are important and develop incomplete use cases if they are less important.

Erlansson et al. (2002) tested usage-based reading on detecting defects in requirements document written in English. They compared usage-based reading to checklist-based reading, using graduate students as subjects. They reported that readers using usage-based reading didn't detect more defects, even though they spent more time on the documents. However, when the severity of defects was considered, usage-based reading detected much fewer minor defects and more moderate defects than checklist-based reading, and to their surprise, checklist-based reading seemed more effective in detecting critical defects. The researchers added that their findings were not conclusive and they attributed the ineffectiveness of usage-based reading on requirements to several factors inherent to their experiments, including lack of training of the subjects. Information-overloading in usage-based reading could be another factor. Certainly readers had to read through more documents such as, in this case, use cases.

6.3 Traceability-Based Reading

Traceability-based reading is a family of reading techniques for inspecting high-level OO design in UML diagrams. The central feature of this family of reading techniques is to trace information between design documents (diagrams and textual descriptions) for design consistency and between design and requirements for design correctness and completeness, thus the name.

6.3.1 Types of Design Defects

Defects in software artifacts can generally be classified as omitted, ambiguous, inconsistent, incorrect, or extraneous information, which is consistent with the defect classification in requirements specification as discussed in defect-based reading. This classification can be tailored to design documents. The types of defects traceability-based reading intended to detect include (Travassos et al., 1999):

1. Omission: concept(s) from the requirements for a system are missing in the design document.
2. Incorrect fact: a design document has a misrepresentation of a concept described in the requirements for a system.
3. Inconsistency: a representation of a concept in one design document is inconsistent with a representation of the same concept in the same or another design document.
4. Ambiguity: a representation of a concept in the design is unclear and could cause a user of the design to misinterpret or misunderstand the meaning of the concept.
5. Extraneous information: the design includes unnecessary information.

6.3.2 High-Level OO Designs Using UML Diagrams

Software design is concerned with the description of real world concepts that are part of the solution for the system envisioned in a requirements specification. High-level design deals with the problem description without considering constraints. That is, it takes functional requirements and maps them to, e.g., UML design diagrams. This allows designers and developers to understand the problem first before attempting a solution. Low-level design deals with possible solutions. Low-level designs, which serve as a model for coding, depend on a high-level design and the non-functional requirements. We shall point out that what Travassos et al. called high-level designs are sometimes referred to as analysis models (Laitenberger et al., 2000).

Object-oriented development is widely practiced in industry nowadays and UML diagrams are frequently used to document OO designs. Multiple UML diagrams are used to capture different views, different levels of abstraction, or different aspects of a system such as static or dynamic behaviors. These diagrams might be drawn at different times by different designers using different viewpoints and abstractions. Consequently, it is hard to guarantee that these diagrams are consistent among themselves and that requirements are correctly and completely captured by these diagrams. It is important to review and inspect high-level designs to ensure developers fully understand the problem before defining a solution as low-level designs (e.g., an activity diagram). It is more difficult and expensive to fix problems in low-level designs or code than fixing them in high-level designs.

6.3.3 Traceability-Based Reading Techniques

Travassos and colleagues (1999; 2002) developed a family of software reading techniques to detect defects in high-level OO designs captured in UML diagrams. They assume that the requirements specification document is available, along with use cases. However, these documents are not subject to review but used to aid the review of the design. Designs are captured in UML diagrams and textual descriptions, including class diagrams, class descriptions, sequence diagrams, and state diagrams.

Traceability-based reading includes seven reading techniques, as shown in Figure 6-2. Each reading technique focuses readers on some aspects of the design. When all reading techniques are combined, a high degree of coverage of the design can be achieved. The seven reading techniques are organized into two groups, horizontal reading and vertical reading. Relevant artifacts (design and/or requirement) are connected by lines with or without arrows and the lines are labeled with numbers 1 through 7. The lines without arrows indicate a horizontal reading (readings 1-4) and the lines with arrows signal a vertical reading (readings 5-7). The dashed line separates lifecycle documents: above the dashed line are requirements description and use cases, which are generated in requirements definition phase; below the dashed line are high-level design artifacts created in the design phase and are subject to review. Each reading technique has a detailed instruction on what to do and where to look for issues. Detected issues are logged as "discrepancy" instead of defect and then the author of the design can decide if it is a defect, since the reader and author/designer may have different ideas about the design.

Figure 6-2. *Reading techniques 1 through 7 in the family of traceability-based reading*

To facilitate the understanding of the reading techniques, let's define some key terminologies (Travassos et al., 2002):

- *Functionality*: A functionality describes the behavior of a system from the user's point of view.
- *Service*: A service is an atomic action performed by a system. One or multiple services are used to compose a system functionality. A service can be used in one or more functionalities.

- *Message*: A message is a lowest level behavior. It represents a communication between objects. It is typically shown on sequence diagrams and must be associated with object behaviors. One or more messages are used to compose a service.
- *Condition*: A condition describes what must be true for a particular message to be executed.
- *Constraint*: A nonfunctional constraint such as performance restricts the way a certain system functionality has to be implemented.

Functional requirements describe concepts and services a system has to fulfill. A use case describes important concepts and services a system provides for a user to accomplish a particular task and it typically describes execution paths through the system functionality. A class diagram describes classes of a system and how they are associated (inheritance, uses, composition, etc). A set of class descriptions lists the classes of a system along with their attributes (properties, data members) and behaviors (methods). A state diagram describes internal states of an object and transitions between different states. State transitions are typically annotated with triggers. A sequence diagram describes classes and objects of a system and how they collaborate through sending and receiving messages to accomplish system services.

6.3.3.1 Horizontal Readings

Horizontal readings are designed to check if all of the design artifacts (class diagram, class description, state diagram, and sequence diagram) describe the same system. All diagrams are created in the same design phase, perhaps at different times. The consistency among them is the focus of horizontal reading. Different diagrams contain complementary views to a system. Class diagrams and state diagrams capture the static behavior of the system, while sequence diagrams capture the dynamic one. All views are important, and together they allow developers to understand system behaviors and figure out how to accomplish them. With reading techniques in this group, a reader mostly performs syntactic checking mechanically. There is not much application domain knowledge required. Instead, the reader's development expertise plays a role here.

6.3.3.1.1 Reading 1: Sequence Diagram vs Class Diagrams

The purpose of this horizontal reading is to verify that classes and their relationships captured in a class diagram are consistent with the behaviors captured in a sequence diagram. This reading is thus concerned with the static behavior of a system. To this end, a reader first verifies that classes and objects appearing in sequence diagrams also appear in class diagrams and then verifies that the relationships, behaviors, and conditions in class diagrams capture the services as described in sequence diagrams (Travassos et al., 2002).

Travassos et al. (2002) provided a detailed reading instruction. Here we sketch their main ideas:

- For every sequence diagram, read and understand services the system provides and how these services should be implemented. In particular,
    - identify objects, classes, and actors in the sequence diagram;
    - identify the information exchanged between objects and determine if the information exchange represents messages or services (services are composed by messages); and
    - identify constraints on these messages and services and conditions when a message will be sent.
- Identify and read class diagrams, and determine if the corresponding objects are documented correctly.
    - First verify for every object, class, and actor identified on sequence diagrams, there is a concrete class in a class diagram.
    - Then verify for every service or message identified on sequence diagrams, there is a class on a class diagram, which encapsulates the service or message, and the association between the sending and receiving object is also reflected on a class diagram. Note that in some cases, a reader may need to trace to the parent class of a class. The reader shall also determine if the series of messages is adequate to achieve the intended service.
    - Verify that constraints identified on sequence diagrams can be met based on the information on class diagrams. Those constraints can be the number of objects, permissible values of an attribute, dependencies between objects, timing constraints, etc.
    - Lastly, exercise some professional judgement if the design is sound, with respect to general design principles such as cohesion and decoupling as well as standards and guidelines.

If the reader cannot perform any steps or verify any points above, he or she shall raise a discrepancy. The author of the design will decide if a discrepancy is a defect or not.

6.3.3.1.2 Reading 2: State Diagrams vs Class Descriptions

The purpose of this horizontal reading is to verify that class descriptions capture the functionality specified in state diagrams (Travassos et al., 2002). Thus this reading is related to the static aspects of a system.

Travassos et al. (2002) provided a detailed reading instruction. Here we sketch their main ideas. To perform this reading, a reader starts with state diagrams, and for each state diagram, he or she performs the following steps and verifies the associated points. If any step cannot be performed or any verification fails, the reader raises a discrepancy that will be analyzed by the author of the design.

- Read and understand all possible and permissible states of an object, all actions that trigger state transitions, and how states and actions fit together.
    - Identify the class which the state diagram is for.
    - Start with the start state, trace all transitions and actions to trigger state transitions, and finish at the end state.
- Read and understand the class or class hierarchy for which the state diagram models.
    - On the class description, identify the class or class hierarchy associated with the state diagram.
    - Identify how the class describes and encapsulates states identified from the state diagram above. Encapsulation may be done as an explicit or implicit attribute, a combination of attributes, or a class. If encapsulated as a class type, read the class and class hierarchy. Make sure all states are encapsulated in the same manner.
    - For each transition action identified on the state diagram, verify that there are behaviors to achieve that state transition. Behaviors can be implemented by the class or its parent classes in the inheritance chain. The transition action can be an event or constraint; thus look for an appropriate behavior.
- Make sure the class as described in the class description captures the appropriate functionality, considering the system context and object behaviors and states.

6.3.3.1.3 Reading 3: Sequence Diagrams vs State Diagrams

State diagrams capture the static information and sequence diagrams capture the dynamic information. These different views allow developers to understand the system and objects from complementary viewpoints. However, these differences complicate reading and inspection, since a reader must combine them and identify possible discrepancies. The purpose of this horizontal reading is to verify that every state transition for any object (read from state diagrams) can be achieved by messages sent and received by that object (read from sequence diagrams). Thus this reading is related to dynamic aspects of a system.

Travassos et al. (2002) provided a detailed reading instruction. Here we sketch their main ideas. To perform this reading, a reader starts with state diagrams and reads them one by one. For each state diagram, the reader performs the following steps and verifies the associated points and raises a discrepancy if any step cannot be performed or any verification fails.

- Read and understand state diagrams. The relevant reading steps are the same as those for reading state diagrams in Reading 2.
- Read sequence diagrams and understand how state transitions are achieved via messages sent and received by an object.
    - Identify the relevant sequence diagram(s) in which the object (modeled in the above state diagram) is used.
    - For each of these sequence diagrams, identify services and messages this object receives. On the state diagram, identify the states related to the service and the states leading to and from these states.
    - Map the messages on the sequence diagram to the state transitions on the state diagram. Make sure the mapping makes semantic sense.
    - Look for conditions and constraints of the mapped messages and verify those conditions and constraints are captured consistently on the state diagram.
- Read through all state diagrams to ensure all state transitions are accounted for and have associated messages identified.
    - Identify any state transition on the state diagram that is not yet associated with object messages. If the transition is labeled with a constraint or event, check if there is a message, series of messages, or some action performed by an actor from outside of the system that can achieve the transition action.

When multiple messages are identified on the same sequence diagram for different state transitions of the same object, the reader shall also verify that the ordering of these messages reflects the topological order of the states (the state transition diagram is a directed graph).

6.3.3.1.4 Reading 4: Class Diagrams vs Class Descriptions

The purpose of this horizontal reading is to verify that class descriptions have all the information required by class diagrams and the class as described in class descriptions makes semantic sense (Travassos et al., 2002). Thus this reading is related to static aspects of a system.

Travassos et al. (2002) provided a detailed reading instruction. Here we sketch their main ideas. To perform this reading, a reader starts with class diagrams and reads them one by one. For each class in a class diagram, the reader verifies the following points and raises a discrepancy if any verification fails:

- The class has a class description.
- The name and textual description of the class are meaningful and the description is at the right level of abstraction.
- All attributes along with their types read from the class diagram are listed in and consistent with the class description. The class can encapsulate those attributes, and those attributes are implementable.
- All behaviors and constraints read from the class diagram are present in and consistent with the class description. The class can encapsulate those behaviors, the behaviors are implementable, the constraints make sense for the class and are satisfiable, and the class has a minimal dependencies on other classes.
- The inheritance relationship is included in the class description, if applicable, and the class hierarchy is reasonable (i.e., it is an "is-a" relationship).
- All class relationships (association, aggregation, or composition) are correctly annotated with regard to the multiplicity in the class diagram, the relationships make sense, the correct cardinalities are documented in the class description if important, and the relationship is represented by some class attribute, with a feasible type or data structure. Object roles and responsibilities are documented in the class description as well.

As the last step, the reader goes through all class descriptions and ensures that there is no class that is mentioned in the description but does not appear in any class diagrams. Otherwise this should be reported as a discrepancy, since it represents extraneous information.

6.3.3.2 Vertical Readings

Vertical readings are to check if design artifacts represent the right system as captured in the requirements specification and use cases, which are assumed to be correct for the design purpose. The artifacts involved here come from two different development phases, requirements definition and design. Vertical reading thus prompts readers to trace and compare documents to ensure designs are correct and complete. The artifacts represent different levels of abstraction and contain different levels of details. The design is considered a refinement of the requirements, and there is no single, direct mapping from requirements to the design. Vertical readings are thus complicated and a reader is required to abstract concepts from the design in order to match them in the requirements and use cases.

6.3.3.2.1 Reading 5: Class Descriptions vs Requirements Description

The purpose of this vertical reading is to verify concepts, and services described in the requirements are captured in class descriptions accurately and completely (Travassos et al., 2002). This reading is thus concerned with the static behavior of a system.

A reader starts with the requirements description to understand the functionality, as Travassos et al. (2002) suggested. The reader will then:

- Read through the requirements one by one.
- Identify nouns in the requirement that can be possible classes, objects, or attributes.
- Also identify verbs or descriptions of actions that can be possible services or behaviors in the design.
- Look for conditions, limitations, or constraints on these identified nouns and verbs.

With these identified nouns, verbs, conditions, and constraints, the reader next verifies these concepts are properly captured in class descriptions. For each and every action verb, find a related behavior or combination of behaviors in the class description. For each and every noun, find a related class in the class description. The noun may be used as class name, an instance name of the class, or an attribute name. If a class is matched, make sure its class description has sufficient information related to the concepts, the class encapsulates attributes and behaviors, and the conditions and constraints are described in the class description as well. If a class attribute is matched, make sure a feasible type is used for that attribute.

Lastly, make sure all nouns, verbs, conditions and constraints are reflected and not omitted in class descriptions.

6.3.3.2.2 Reading 6: Sequence Diagrams vs Use Cases

The purpose of this vertical reading is to verify that combinations of objects and messages sent among those objects as captured in sequence diagrams fulfill functionalities described by use cases (Travassos et al., 2002). This reading is related to both static and dynamic aspects of a system.

To achieve the reading purpose, a reader reads and understands use cases first. He or she goes through each use case, identifies the functionality the use case describes, and identifies important system concepts that are necessary to accomplish the functionality. Nouns in the use case describe concepts of the system. For those identified nouns, the reader further identifies verbs that describe actions applied to or by the nouns. Those identified verbs represent services the system provides. Constraints and conditions for those services, if any, shall also be identified. The reader also pays attention to the data or information exchanges in order to perform the actions.

Once having understood use cases, the reader identifies and reads related sequence diagrams, and identifies on these sequence diagrams the corresponding system objects, services, and data or information exchanges that were identified earlier in the use case. A correct set of sequence diagrams must be selected, which relies on deep understanding

of the system as well as traceability information. As sequence diagrams are refinement of the use case, the reader may need to identify messages and abstract out the services from these messages.

With these two sets of information identified on use cases and sequence diagrams, the reader next compares them semantically and verifies that they represent the same domain concepts.

- The reader first verifies for every noun identified on the use case, there is a corresponding noun represented on sequence diagrams. For every unmatched noun identified on the sequence diagrams, the reader makes sure it is an attribute of some class by searching through class descriptions; otherwise this could be an extraneous data.
- The reader compares the services identified on sequence diagrams and on the use case. The reader shall focus on the ordering of the messages/services and data exchanged between objects.
- Lastly the reader verifies that the constraints and conditions identified on the use case are observed on sequence diagrams.

While reading and performing the above steps, the reader shall raise a discrepancy if any steps cannot be performed or any point fails the verification.

6.3.3.2.3 Reading 7: State Diagrams vs Requirements Descriptions and Use Cases

The purpose of this vertical reading is to verify that object states and events that trigger state transitions are correct and complete as described in the requirements and use cases (Travassos et al., 2002). This reading is related to the dynamic behavior of a system.

To achieve the reading purpose, for each state diagram, a reader performs the following steps:

- Read the state diagram to understand the object whose states are modelled.
- Read the requirements description, compile a list of states for the object and complete the state adjacency matrix. To complete the task, the reader reads through the requirements description and focuses on where the concept related to the object is described, participates, or is affected. Among those places, the reader abstracts the different states. To identify a state, look for the situations where the object behaves differently as a result of the change of an attribute or a group of attributes. Among all the states identified, determine the start state and end state. Draw an adjacency matrix table, where the row signals a “from” state and the column signals a “to” state. For each pair of states (the order matters), mark if the state transition is allowed with the event and constraints if known or if the transition is forbidden.

- Read use cases and further complete the adjacency matrix. Choose the use case descriptions where the concept is involved. If the state transition is allowed, but the event is missing, complete the event. For the state transitions not determined from the requirements, use the use cases to determine if the transitions are allowed with events or forbidden.
- With those preparations, the reader is ready to examine the state diagram. The reader first checks if all states identified from requirements are represented on the state diagram. Be aware that those corresponding states may have different names. Also be mindful that two states might be merged. Conversely, make sure for every state shown on the state diagram, there is a state identified from the requirements. Once the states from the state diagram and from the requirements are paired, compare the state transitions and their triggers. Again the comparison is a two-way check. Finally, make sure constraints captured in the adjacency matrix are reflected on the state diagram. This is also a two-way check to guard against missing and extraneous constraints.

Like all other traceability-based reading techniques, the reader shall log any discrepancy if any above steps cannot be performed or any verification fails.

6.3.3.3 Semantic Checking

The traceability-based reading family includes seven reading techniques grouped into horizontal reading and vertical reading. In the horizontal reading, a reader compares design diagrams and a textual description. Since those documents represent the same level of abstraction, the comparison is relatively easy and syntactic checking is sufficient most of the time. With tools, the syntactic checking can be automated, which relieves the burdens on readers.

It is the semantic checking that proves to be challenging. This is apparent in the vertical reading group. For example, state names identified from the requirements may not be the same as those directly captured in the state diagrams. Even worse, two or more states can be merged in the design. The reader must understand the requirements and design in order to adequately deal with the situation. As another example, the requirements and use cases talk about system services, but sequence diagrams use messages between objects. In order to properly trace between them, the reader must abstract services out of a group of messages.

The semantic checking is performed or required in the horizontal readings as well, although not as pervasive as in the vertical readings. The aforementioned situation of messages versus services also exists when reading sequence diagrams and state diagrams (in Reading 3, state diagrams use services, but sequence diagrams use messages). In Reading 4, Class Diagrams vs Class Description, the reader is asked to decide if the class description makes semantic sense in terms of encapsulations of attributes and behaviors.

Because of this semantic checking, some level of subjectivity might be involved and a reader's domain experiences play a significant role. That is why Travassos et al. call the detected issues discrepancies, not defects, since the authors may have different ideas or opinions than the reviewer.

6.3.3.4 Practical Considerations

Traceability-based reading defines seven individual reading techniques. For an industrial-scale project, many diagrams may have been created. It is thus a huge effort to conduct a full traceability-based reading. Here we offer a few suggestions to tame the complexity.

If an incremental development process is used in your project, you can review artifacts gradually, as they are created incrementally. If this is not the case for your project, you can subdivide your system into units for inspection. Travassos et al. (1999) offered two alternatives. Each inspection can be focused on some subset of system functionality in requirements or subset of conceptual entities in designs. As the mapping between the functionality and OO concepts is not trivial, there are issues with either approach.

The reading responsibilities can be distributed among team members, to reduce the workload of and to provide focus to individuals. The responsibilities can be divided along the dimension of reading techniques. One possibility is to ask one or more readers to perform the horizontal reading for consistency checking, ask one or more readers to focus on the vertical reading for traceability, and discuss and resolve the discrepancies at the end. The other possibility is to ask one or more readers to focus on static views of a design (i.e., Readings 1, 2, 4, 5) and ask another one or more readers to focus on dynamic views (i.e., Readings 3, 7), with one in common for both static and dynamic views (i.e., Reading 6). The reading responsibilities can also be divided along the dimension of artifacts. For example, reader A can focus on artifacts a, b, and c, while reader B can focus on artifacts x, y, and z. All readers use the seven reading techniques. Of course, the responsibilities can be divided using both reading techniques and artifacts.

If all diagrams and descriptions are available, the development team can use all seven reading techniques defined by traceability-based reading. If some design artifacts do not exist, the team can just skip those reading techniques in which missing artifacts are used. Even if all artifacts exist, the development team can choose and apply a subset of reading techniques, depending on what attributes of the designs are important and what kinds of defects the team wants to detect.

In general, horizontal reading shall be performed before vertical reading, since we want to make sure the designs are consistent before validating them against the requirements. Specific situations may warrant the need to alter the reading order.

6.3.4 Empirical Experiences

Travassos et al. (1999) used undergraduate students to evaluate the feasibility of traceability-based reading. They reported that the reading techniques did allow subjects to detect defects, and a majority of the subjects thought the techniques were helpful. Vertical reading tended to find more defects in types of omission and incorrect facts, while horizontal reading tended to find more defects in types of ambiguity and inconsistency. The research team continued to improve and refine the reading techniques by conducting observational studies and case studies, and the results were summarized in Travassos et al. (2002). The reading techniques could be easily integrated into a development process, they indeed caught critical defects, and the effort was not prohibitively high when compared to other system tasks.

Traceability-based reading was successfully applied in industrial environments. Melo et al. (2001) reported a case study in which the reading techniques were applied in an industrial environment at Oracle Brazil, where a team of five people participated. Reading techniques allowed participants to find, on average, 35 defects (the most was 57 and the least 12). The number of false-positive defects was very low (0.8 on average). The average time spent (individual reading and meeting) was 5.8 hours and the average time per defect was 15 minutes. Conradi et al. (2003) conducted an industrial experiment at Ericsson in Norway. Traceability-based reading techniques were slightly tailored to their environment. They reported that traceability-based reading techniques fit well in an incremental development process, and engineers employing the techniques found more defects (39 vs 25) than their existing reading techniques, which were a mixture of checklists and guidelines, while the cost efficiency was almost the same (1.28 vs 1.37 defects per hour). Traceability-based reading seems to be complementary to their existing reading techniques as they found different types of defects, and engineers employing the new reading techniques found more defects of omission and inconsistency types. However, the new techniques need to be simplified with added questions and special guidelines for their project.

The ideas behind traceability-based reading have been used in other reading techniques. One can clearly see the influence of traceability-based reading in perspective-based reading of design from an analyst's viewpoint (Laitenberger et al., 2000). The architecture reading techniques presented in the next section was also motivated by traceability-based reading.

6.4 Architecture Reading

Software architecture is becoming a mature discipline. It is the blueprint for the software system development and has a far-reaching impact on organizations. Thus it is critical to review and get it right early on (and review it regularly afterward). The principal objectives of architecture reviews are to evaluate if the architecture can potentially deliver a system to fulfill the quality requirements and to identify potential risks. Babar and Gorton (2009) surveyed the current industry practice, and Knodel and Naab (2014) reported their own experiences on assessing industrial software architectures.

Architecting a system is to make fundamental, long-lasting, and hard-to-change decisions. These decisions can be implicit as made in an architect's head or explicit as found in documentation and manifested as implemented in the system (Knodel & Naab, 2014). Architecture review or evaluation is to assess all of these decisions. In this section we discuss architecture reading techniques, i.e., we are only concerned with reading and finding defects in software architecture documentation.

6.4.1 What Is Software Architecture?

Software architecture is a widely used term now. There are many definitions that capture different aspects of software architecture, and there is no commonly accepted definition. We like the definition used by Clements et al. (2011; Bass et al., 2013) : *The software architecture of a system is the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both.*

Software architecture is important. Bass et al. (2013) listed many reasons, including inhibiting or enabling a system's quality attributes (-ilities), reasoning about and managing changes, predicting system qualities even before it is built, enhancing communication among stakeholders, carrying early and fundamental design decisions, defining constraints on an implementation, influencing the organizational structure, enabling evolutionary prototyping, improving cost and schedule estimates, supplying a transferable and reusable model, allowing incorporation of independently developed components, restricting the vocabulary of design alternatives, and providing a basis for training.

A software architecture is shaped by many factors, and in turn, the architecture influences these factors. In addition to technique requirements, business, technical, project, and professional contexts influence the architecture (Bass et al., 2013). A software architecture is documented in a software architecture description that is both descriptive and prescriptive. An architecture description typically has textual description decorated with graphical diagrams, capturing module views, component-and-connector views, and allocation views using architecture patterns, styles, and frameworks (Clements et al., 2011).

We shall mention the differences between architecture and design. Architecture is design, but not all designs are architecture. The architecture is concerned with the fundamental structure of the system that delivers the quality attributes. These quality attributes are non-functional requirements and typically referred to as –ilities, such as availability, interoperability, modifiability, performance, security, testability, usability, and so on. High-level design as discussed in the traceability-based reading is a form of requirements analysis and expresses the requirements in analysis models, which are easy for designers to consume. Design is figuring out how to implement the functionalities within the software architecture constraints. Some authors dislike the term “detailed design,” as architecture design can occasionally be very detailed in some areas if the architects think they are important.

6.4.2 Traceability-Based Architecture Reading

The general concerns of architecture reading are thus to make sure different views are consistent in addition to complementary, and all views are correct with respect to and traceable to the requirements as well as other architecture drivers. The general ideas behind the traceability-based reading appear to be applicable to architecture reading. Carver and Lemon (2005) developed the architectural reading techniques, which were rightly motivated by the traceability-based reading.

One horizontal reading and three vertical reading techniques are available for traceability-based architecture reading:

- The horizontal reading is to compare the information in one part of the architecture description to the other parts of the architecture description and ensure they are consistent. More specifically, the following pairwise comparisons between views are performed:
    - Logical structures (module views) vs communications patterns (component-and-connector views)
    - Logical structures vs physical structures (allocation views)
    - Communication patterns vs physical structures

- The first vertical reading is to check if the logical decomposition captured in the module views is realistic, judged from the information in the requirements document.
- The second vertical reading is to check if the communication patterns captured in the component-and-connector views are accurate based on the requirements document.
- The third vertical reading is to compare the information about the stakeholders, architectural concerns, and rationale captured in the architecture description to the information in the requirements and ensure they are consistent.

Note that besides the requirements, other architectural drivers are not considered in the initial version of the architecture reading. An excerpt of the first vertical reading (module views vs requirements) is shown here (Carver & Lemon, 2005):

1. Read through the requirements and compile a list of modules that implement the system.
2. For each module view in the architecture document, read through all relevant sections of the architecture document.
    a. For each module in the module view, make sure there is a module identified from requirements in Step 1. If there is no match, log an issue as potentially extraneous information.
    b. Conversely, for each module identified in Step 1, make sure there is a module view in architecture that contains that module. If there is no match, log an issue on potential missing information. If the match is not found in the current module view, decide if that module should have been included in the view.
    c. Use the requirements as the guide to examine the relationship among modules. If the relationship does not make semantic sense (e.g., use relation, is-a relation, part-of relation), log an issue as potential incorrect fact. This also decides whether the existence of a module in the module view makes sense or not, taking into account the intentions of the module view.

6.4.3 Empirical Experiences

Carver and Lemon (2005) conducted a feasibility study of the traceability-based architecture reading and compared it with checklist-based reading, using students as subjects. The checklist was adopted from the first edition of the book by Clements et al. (2011). They concluded that, with checklist-based reading, subjects were more effective at finding defects of commission than defects of omission; with traceability-based architecture reading, subjects were more effective at finding defects of omission than defects of commission. Subjects reported that they preferred traceability-based architecture reading over checklist-based reading, as the former helps readers focus and

provides more detailed and structured guidance to read and inspect the whole document set. Traceability-based architecture reading appears to be feasible and easy to understand and use. The authors didn't publish a follow-up study of the reading technique and there was no independent replication yet.

We shall be cautious on the omission defects, however. As the architecture description is meant to document the architecturally significant decisions, the architect may choose to leave some details to downstream designers. In other words, the architect does not care whatsoever how a subpart of the system is designed and constructed. Thus a miss is not really a miss from the architect's point of view. For that reason, readers log potential issues for architects to address.

6.4.4 Other Architecture Reading Techniques

As mentioned earlier, architecture reading is only a small portion of architecture review or evaluation and is concerned with reading and finding defects in architecture documentation. Clements et al. (2011) devoted one chapter to software architecture document review. They presented a six-step procedure, and the fifth step is to perform the review, which is assisted with sets of questions. They defined a template for a question set. Which set of questions to use depends on the purpose of an architecture review. During the lifecycle of a system, the architecture document shall be reviewed often and for different purposes, such as to check if it has captured the right stakeholders and their concerns in the concept phase or to check if it can support development in the development phase. The questions for a purposed review are organized for different stakeholders. For example, software development managers, designers and implementers, integrators, testers, and quality assurance engineers are stakeholders when assessing whether the architecture document can support development. The concerns of these stakeholders are also provided to assist the architecture document review. The styles of the questions are sometimes similar to what one would expect in an active design review. One can clearly see some flavors of checklist-based and perspective-based readings in their approach.

We would be remiss if we didn't mention the state of the practice for software architecture evaluation. Broadly speaking, there are two approaches to software architecture assessment, scenario-based and model-based. The very first architecture review technique is software architecture analysis method (SAAM). The SAAM is a scenario-based method. Here a scenario is a brief description of some anticipated or desired use of a system. Essentially the SAAM develops, classifies, and prioritizes the scenarios and assesses how and if the scenarios can be mapped to the architecture. The method was originally developed to assess an architecture for modifiability and superseded by a new general method, architecture tradeoff analysis method (ATAM). ATAM is applicable to assess almost any quality attribute, particularly the tradeoff among conflicting ones, but it is a very expensive method – it involves a big team for a few days. A lightweight version has recently been proposed by the team at the Software Engineering Institute. Besides the general purpose methods, there are specific ones focusing on particular quality attributes, e.g., architecture-level modifiability analysis (ALMA), performance analysis of software architecture (PASA). On the other hand, the model-based approach uses some formal models that can be automatically analyzed with tools. Architecture description languages are formal languages for describing and analyzing architectures. Interested readers can find more information in books on architecture evaluation, such as Clements et al. (2002).

6.5 Scope-Based Reading

The objectives of software reading are for analysis or for construction. Almost all the reading techniques discussed in this book belong to the category of reading for analysis (defect detection). Here we discuss one reading technique, scope-based reading, of which the purpose is for construction (Shull, 1998; Shull et al. 2000). Reading for construction is useful for software maintenance or new system development from reusable components, frameworks, and architectures. Scope-based reading was developed to enable readers to reuse both the design and implementation of OO application frameworks.

6.5.1 What Is an Application Framework?

An application framework—particularly an OO framework—is a reusable design and implementation of all or part of a system for a specific class of software, represented as a set of cooperating classes, including abstract ones, and the way their instances interact. It provides a skeleton application for a particular domain and regulates interactions among various constituting elements. It typically has a set of hotspots where framework users can extent, customize, or plug in their own components. A framework serves to speed up application development by providing a foundation and a set of design and implementation elements that can be built upon and expanded as required.

From a software engineering perspective, it is desirable to develop an application framework to facilitate software development, since there are many advantages of framework-based development as claimed by proponents of this technology and supported by many successful industrial stories. Although the development of OO frameworks requires significant effort, the benefits generally justify the initial effort.

There are many different kinds of application frameworks. The development of OO application frameworks in general and domain-specific frameworks in particular has been a major industry thrust. There is big body of literature on framework construction, but literature on how to learn and use frameworks is scarce. The very steep learning curve of application framework is notoriously known among software practitioners.

6.5.2 Scope-Based Reading Techniques

Scope-based reading is meant to help developers learn and use white-box OO frameworks. Here white-box means that the internal part of the framework, particularly the class hierarchy, is exposed to the users who typically derive subclasses to implement new functionalities for new applications. As the application domain becomes better understood, the framework becomes mature and it evolves from white-box toward black-box.

There are two effective ways to learn an OO application framework: by studying the class hierarchy of the framework, and by studying example applications. Shull et al. (2000) proposed two corresponding reading techniques: hierarchy-based reading and example-based reading, together known as scope-based reading. These techniques give readers a different scope or breadth of knowledge about the framework, thus the name.

Scope-based reading assumes an OO application framework is given, the reader wants to construct an application making use of the framework, he or she already has the object model and dynamic model of the application to be built, and the object model and dynamic model are not adapted to the application framework yet.

6.5.2.1 Hierarchy-Based Reading

Hierarchy-based reading uses a stepwise approach. To help developers understand the functionality provided by the framework, it concentrates readers on the base or abstract classes first, then moves down the class hierarchy to the derived classes to find the most specific instantiation. It has the following steps (Shull et al., 2000):

1. Study the class hierarchy of the application framework. Determine which part of the framework architecture contains the functionalities that can be used or modified for the application under construction and identify base classes that can be used in the new application.
2. Identify classes related to the new application. For each class in the object model, find an appropriate class in the framework from which the class can be derived, and read and understand the framework class. If an appropriate class is not found, consider aggregating multiple framework classes to create a new class with the required attributes and operations.
3. Revise the object model and dynamic model for the application by using classes from the framework directly or classes composed from multiple framework classes, with details such as relevant attributes and operations.
4. Develop the application using the modified object models by deriving or aggregating framework classes wherever possible to reuse the attributes and operations provided by the framework. Write code as needed to extend the functionality and add new classes as needed to complete the new application.

Hierarchy-based reading has a steep learning curve. The reader has to understand how different pieces of the framework fit together before being productive. In practice, it is almost always supplemented with other reading techniques, such as the example-based reading technique discussed in the next section.

6.5.2.2 Example-Based Reading

The main idea behind the example-based reading is to learn from examples (Shull et al., 2000). It assumes there is a collection of examples, illustrating the range of functionality and behavior provided by the framework. The examples can be provided by framework developers as applets or developed by others as real-world applications. Further assume the reader understands the functionality of the application he or she is constructing. Example-based reading is an iterative process, since the new application will be built incrementally, feature by feature.

1. Decide which functionality should be added to the application under construction in the current increment. Search the collection of examples to find out which example provides the most complete coverage of that functionality. The reader can run each example in turn to check what features were implemented in the example.
2. Study the selected example in depth, but focus on the functionality being sought. By running the example, the reader constructs a set of use cases related to the sought-after functionality. For each use case, the reader traces the code, e.g., in the debug mode, to identify which objects/classes, attributes, and operations/methods are involved and how they cooperate to achieve the functionality. It is necessary to read, understand, and document the relevant code along the way, including the object models and dynamic models related to the use cases.
3. Revise the object model and dynamic model for the application, based on the newly discovered or learned information.
4. Add the functionality to the application. The reader adds new classes modified from the example and augments existing ones with new attributes or methods. If the example does not implement the complete functionality, the reader writes additional code as needed. He or she also provides code to integrate the newly implemented functionality to the application.
5. Go back to Step 1 for the remaining functionalities, if the application is not complete yet.

During the above reading process, it is a good practice to record what is done, what decision is made, and the rationale behind the decision. For example, when selecting one particular example to study in depth, the reader can note which functionality is being sought, what are the candidate examples, and why a particular example is chosen, etc.

Example-based reading can quickly make the reader more productive. However, it is unlikely that the framework originator will or can provide a big set of examples to cover the complete features the framework provides and enables. It is also a concern, at least for a non-trivial application, whether the functionalities borrowed from different examples can coherently coexist in the final, finished application.

6.5.3 Empirical Experiences

Shull (1998) and Shull et al. (2000) used software engineering students as subjects and conducted an exploratory experiment in an academic environment to test the effectiveness of hierarchy-based and example-based techniques. They concluded, in their environment, that example-based techniques are well-suited to use by beginning learners and hierarchy-based techniques are not well-suited to use by beginners under a tight schedule. Example-based learning is a natural way to approach learning of a

complicated system. Hierarchy-based technique is time-consuming and does not give developers an idea of where the starting place for implementation is or where they can look for certain functionality. The hierarchy-based technique would have been very effective if there has been sufficient documentation or the developers have been given more time. The example-based technique is not without problems, however. Developers experienced difficulty in finding and extracting a small functionality embedded in much larger examples. The effectiveness of the example-based technique depends on breadth of functionality and other characteristics of example applications. Quite often the rationale of the design choice is missing from examples, which makes it hard for developers to reason about their choice. The example-based technique may limit the developers to explore beyond what examples provide.

6.6 Summary

In addition to generic reading techniques such as ad hoc, checklist-based, and perspective-based reading discussed in earlier chapters, this chapter introduced specific reading techniques for design inspection. Usage-based reading employs prioritized use cases to focus readers' attention on what matters most to end users. Traceability-based reading is applicable to inspecting OO designs. The family includes seven techniques organized as horizontal reading and vertical reading, with the semantic checking as a theme across both. Horizontal reading ensures all design artifacts are consistent among themselves and vertical reading ensures designs are consistent with the requirements specification. Architecture reading is an emerging reading technique, motivated by the ideas behind traceability-based reading. While these reading techniques focus on defect detection in software artifacts such as design and architecture, scope-based reading is meant for developers to read the application framework and figure out how to construct software applications. With these ranges of reading techniques, you are ready to read and understand various software design artifacts to accomplish various tasks, be it for defect detection or for constructions.

6.7 References

(Bass, 2013) L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 3rd ed., Addison-Wesley, 2013.

(Babar, 2009) M.A. Babar and I. Gorton, Software architecture review: the state of practice, IEEE Computer, vol.42, no.7, pp.26-32, 2009.

(Cantone, 2003) G. Cantone, L. Colasanti, Z.A. Abdulnabi, A. Lomartire, and G. Calavaro, Evaluating checklist-based and use-case-driven reading techniques as applied to software analysis and design UML artifacts, LNCS 2765, pp.142-165, 2003.

(Carver, 2005) J. Carver and K. Lemon, Architecture reading techniques: a feasibility study, Proceedings of 4th International Symposium on Empirical Software Engineering (late breaking research track), pp.17-20, 2005.

(Clements, 2002) P. Clements, R. Kazman, and M. Klein, Evaluating Software Architectures: Methods and Case Studies, Addison-Wesley, 2002.

(Clements, 2011) P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. merson, R. Nord, and J. Stafford, Documenting Software Architectures: Views and Beyond, 2nd ed., Addison-Wesley, 2011.

(Conradi, 2003) R. Conradi, P. Mohagheghi, T. Arif, L.C. Hegde, G.A. Bunde, and A. Pedersen, Object-oriented reading techniques for inspection of UML models – An industrial experiment, Lecture Notes in Computer Science, vol.2743, pp.483-500, 2003.

(Erlansson, 2002) M. Erlansson, T. Thelin, and M. Host, Usage-based reading for inspections of software requirements, 2nd Conference on Software Engineering Research and Practice in Sweden, 2002.

(Knodel, 2014) J. Knodel and M. Naab, Software architecture evaluation in practice – retrospective on more than 50 architecture evaluations in industry, IEEE/IFIP Conference on Software Architecture (WICSA), pp.115-124, 2014.

(Laitenberger, 2000) O. Laitenberger, C. Atkinson, M. Schlich, K. El Emam, An experimental comparison of reading techniques for defect detection in UML design documents, Journal of Systems and Software, vol.53, no.2, pp.183-204, 2000.

(Melo, 2001) W. Melo, F. Shull, and G.H. Travassos, Software review guidelines, Technical Report ES-556/01, Systems Engineering and Computer Science Program, COPPE. Federal University of Rio de Janeiro, 2011 (http://www.cos.ufrj.br/uploadfile/es55601.pdf accessed on March 10, 2016).

(Musa, 1993) J.D. Musa, Operational profiles in software reliability engineering, IEEE Software, vol.10, no.2, pp.14-32, 1993.

(Saaty, 2001) T.L Saaty and L.G. Vargas, Models, Methods, Concepts and Applications of the Analytic Hierarchy Process, Kluwer Acdemic Publishers, 2001.

(Shull, 1998) F. Shull, Developing techniques for using software documents: A series of empirical studies, Ph.D Dissertation, University of Maryland at College Park, MD, 1998.

(Shull, 2000) F. Shull, F. Lanubile, and V.R. Basili, Investigating reading techniques for object-oriented framework learning, IEEE Transactions on Software Engineering, vol.26, no.11, pp.1101-1118, 2000.

(Thelin, 2001) T. Thelin, P. Runeson, and B. Regnell, Usage-based reading – an experiment to guide reviewers with use-cases, Information and Software Technology, vol.43, no.15, pp.925-938, 2001.

(Thelin, 2003) T. Thelin, P. Runeson, and C. Wohlin, An experimental comparison of usage-based ad checklist-based reading, IEEE Transactions on Software Engineering, vol.29, no.8, pp.687-704, 2003.

(Thelin, 2004a) T. Thelin, C. Andersson, P. Runeson, N. Dzamashvili-Fogelstrom, A replicated experiment of usage-based and checklist-based reading, 10th International Symposium on Software Metrics, pp.246-256, 2004.

(Thelin, 2004) T. Thelin, P. Runeson, C. Wohlin, T. Olsson, and C. Andersson, Evaluation of usage-based reading – conclusions after three experiments, Empirical Software Engineering, vol.9, pp.77-110, 2004.

(Travassos, 1999) G.H. Travassos, F. Shull, M. Fredericks, and V.R. Basili, Detecting defects in object-oriented designs: using reading techniques to increase software quality, Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp.47-56, 1999.

(Travassos, 2002) G.H. Travassos, F. Shull, J. Carver, and V.R. Basili, Reading techniques for OO design inspections, Technical Report CS-TR4353, University of Maryland, 2002 (http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/CS-TR4353.pdf accessed on March 10, 2016).

(Winkler, 2005) D. Winkler, S. Biffl, and B. Thurnher, Investigating the impact of active guidance on design inspection, LNCS 3547, pp.458-473, 2005.

CHAPTER 7

Code Reading Techniques

Most of the reading techniques discussed in Chapters 3 and 4 can be applied to code reading. This chapter first emphasizes the importance of code reading as a professional skill and discusses how people actually read code in practice. It then focuses on a few specific reading techniques for programmers to read and comprehend the programming source code. Reading by stepwise abstraction can be used for structured code. Object-oriented (OO) languages such as C++, Java, and C# are very popular nowadays. However, OO code reading is challenging. Stepwise abstraction is extended to support OO reading. To understand both static and dynamic aspects of the OO code, use-cases are used to direct the reading process. Object-oriented framework reading is even more challenging and functionality-based reading is an effective and efficient technique in finding defects in frameworks. Software developers spend most of their professional lives on reading other developers' code and there are many legacy codes that developers are tasked to maintain. Task-directed reading is developed to fill that need. To read and understand the programming code, the readers themselves shall be familiar with the program constructs. This is not covered here, however. We instead cover techniques that are applicable to any high-level programming languages. Lastly, factors that impact code readability are also examined before we conclude the chapter.

7.1 Code Reading As a Professional Skill

7.1.1 Importance of Code Reading

We read programming code for different purposes. As students, we read code in books, magazines, and journals or on the web to learn language constructs and how to master them. As professionals, we may still read code for learning, but most of the time it is for other reasons. We read code written by colleagues as part of the code inspection process; in this case, we read and analyze the code to verify its quality and detect possible defects. We also read code to identify reuse opportunities or figure out how to use it in our own projects. As software developers, a large part of our professional lives is spent maintaining, adapting, correcting, perfecting, and modifying existing code. Empirical data show that developers spend about half of their time on reading and comprehending programs during software maintenance (Maalej et al., 2014). We want to introduce minimal disruption to the existing functionality to maintain its original architectural and stylistic integrity. These purposes are consistent with the schema Basili et al. (1996) categorized, i.e., reading for analysis and reading for construction. To fulfill all these purposes, we must read and achieve a necessary level of understanding of the code. Successful software engineering professionals must learn, cultivate, and master code reading skills.

© Yang-Ming Zhu 2016

Y.-M. Zhu, *Software Reading Techniques*, DOI 10.1007/978-1-4842-2346-8_7

7.1.2 How Do People Read Code?

Program comprehension is an important activity in software engineering and a central activity during software maintenance, evolution and reuse. Comprehension is a process in which individuals build their own mental representation of the program. It has continued to be an active research topic since the 1970s for computer science researchers as well as cognitive psychologists, and there is a rich body of literature in the field. The IEEE Computer Society has been running an annual meeting, the IEEE International Conference on Program Comprehension, since 1992. Von Mayrhauser and Vans (1995) provided a survey of program understanding in 1995. Storey (2005) provided an updated review in 2005 and Schulte et al. (2010) surveyed the topic from the computer science education point of view.

Although program comprehension is a broad topic, much of the research has been focused on code reading and comprehension. Researchers are interested in how people, both experts and novices, read and understand code and what strategies they exploit to facilitate comprehension. Due to its practical importance, program comprehension is still an active and interesting topic today (Maalej et al., 2014).

To understand the empirical findings on how people read and understand programs, researchers put forth numerous, sometimes conflicting, models, which tend to have a set of common elements: an assimilation process, cognitive structures, and the knowledge base (Schulte et al., 2010). The assimilation process is the reading process or strategy the programmer uses to extract information from the code in order to build their mental representation of the code. Such strategies may include top-down and bottom-up. It is similar to the reading techniques we have discussed but less well-structured. The cognitive structure may include a programmer's existing knowledge base on the programming and application domain, and his/her mental representation of the model. Although the elements are common, the views on them are not. It is not our interest here to review and analyze cognitive models, for which interested readers can refer to the aforementioned surveys. We rather summarize the empirical findings related to code reading and comprehension.

Source code is not read like a novel, nor is its meaning determined by seeing how it behaves when run or traced using test data (Deimel & Naveda, 1990). To read and understand the code, one has to build layered abstractions of the code. There are various strategies to build the layered abstractions, one of which is top-down reading. Top-down reading is similar to how we write code. When we write code, we typically follow a divide and conquer approach to decompose a high-level function into multiple low-level ones, recursively as needed. In top-down reading, one gains the understanding of the code by appreciating the overall purpose first, followed by understanding how the function is implemented by constituent components. During this process, the reader repeatedly forms hypotheses about the code, which are subsequently verified, modified, or rejected. The reader scans the code and searches for familiar clues in the text, which are called beacons (Brooks, 1983) or program plans (Soloway & Ehrlich, 1984). For this reading strategy to be effective, one needs to know the overall programming purpose, and well-documented code can usually be read top-down (Linger et al., 1979). This top-down reading approach was theorized by Brooks (1983) and was observed by Soloway and Ehrlich (1984).

Bottom-up reading is the opposite of top-down reading. In bottom-up reading, understanding of the code is accumulated by understanding of small fragments of code. The reader recognizes the function of groups of statements as chunks and combines these chunks to explain increasingly larger program fragments (Shneiderman & Mayer, 1979). Deep knowledge of the programming language and constructs, and of the application domain, helps the reader read and understand the code. Identifying and understanding the control flow and data flow of the program greatly facilitates the global program comprehension (Pennington, 1987). Cross-referencing the program domain and the application domain tends to confirm and enhance the level of understanding, which points to the significant roles the reader's experience and knowledge play during reading and comprehension.

Empirical studies of professional programmers reveal that people do not employ pure top-down or bottom-up strategies but mix them freely (Letovsky, 1987; Littman et al., 1987). Code reading and comprehension is a hard and time-consuming task, and programmers often adopt "as-needed" (opportunistic) approaches to avoid deep understanding (Littman et al., 1987; Maalej et al., 2014). They focus on the task at hand and gain just enough knowledge to complete the task. They do a deep understanding of the code only when they have to. Littman et al. suggested that people adopting the as-needed reading strategy focus on local program behavior and fail to construct successful modifications to the program, since they fail to detect critical interactions among program components and don't have a complete and accurate understanding of the code. The pragmatic approach to code reading and comprehension has also been recently related to code reuse (Holmes & Walker, 2013). Which strategy to use is typically driven by what question the reader is seeking to answer. For example, to answer a "how" question (how a sorting algorithm is implemented), top-down reading is warranted in order to find out the low-level implementation.

Maalej et al. (2014) recently studied the state of the practice of software reading and comprehension. They were interested in the workflows followed by professional developers to read and understand programs, tools they used, and knowledge they needed, accessed, and shared. They observed and surveyed professional software developers at different companies. Their findings confirmed the results of previous studies and revealed some aspects of industry practices. The workflows including reading strategies vary among developers and depend on their skills, experience, personality, tasks at hand, and technology used. The tool usage was very low or limited and some developers were even not aware of the existence of certain features in tools they used daily.

7.2 Reading by Stepwise Abstraction

Reading by stepwise abstraction is a bottom-up reading technique that formulates an abstract description of what a fragment of code does from the fragment itself. It was first presented by Linger et al. (1979), which became the basis for what Basili and Mills (1982) did with greater formality. The reading technique was later integrated into the Cleanroom process, as the verification-based inspection to assert the implementation correctness (Selby et al., 1987; Dyer, 1992).

Linger et al. (1979) have argued that program writing is expanding the known function into a program and program reading is abstracting the known program into a function. When reading code for defect detection, one compares the known functions (design) to their expansions (code). Code reading is thus to recognize directly what the code does or mentally transform it into something that can be recognized directly. The result of this mental transformation is an abstraction, irrespective of all implementation details.

Reading by stepwise abstraction can be applied to reading any programming code. According to Linger et al., a structured program of any size can be read and understood in a completely systematic manner by reading and understanding its hierarchy of prime programs and their abstractions. A prime program is a fragment of code that has one entry and one exit and is irreducible in some sense. The purpose of reading the prime programs is to discover their program functions, and the program functions can be captured as comments in code. A well-structured and documented program can be read top-down, from overall design to lower levels of details. For poorly structured and documented code, however, bottom-up reading is a better strategy, which allows one to discover the intermediate abstractions, successively at higher levels. The process of bottom-up reading is called stepwise abstraction.

The description of reading by stepwise abstraction is not very clear in literature. We can, however, identify the reading instruction in Panel 7-1: Instruction for Stepwise Abstraction.

PANEL 7-1: INSTRUCTION FOR STEPWISE ABSTRACTION

1. Read code line by line to build up a conceptual understanding of code fragments.
2. Connect code fragments to form an overall picture.
3. Compare with specifications to detect defects.
4. Repeat until all code is abstracted and compared with specifications.

Reading by stepwise abstraction can be better understood with the example in Figure 7-1. From code in lines 23-25, one can conclude that `b` stores the maximum value of the array `samples` seen so far. Similarly, from code in lines 20-22, `a` stores the minimum value of the array seen so far. Lines 17 and 18 assign the first element of the array to both `a` and `b`, assuming the array is allocated with at least one element. The loop at line 19 loops over the remaining array elements. At the end of the code segment, `a` holds the minimum value of the array and `b` holds the maximum value of the array, i.e., `a`=max(`samples`()) and `b`=min(`samples`()), which is the abstraction one gains after reading this piece of code. Imagine this code segment is part of a larger chunk of code. With reading by stepwise abstraction, that small piece of code can be abstracted as such. Note that the conditional statements inside the loop have a gap (the element value is within `a` and `b`, inclusive), and it is appropriate to have an empty operation for the missing condition.

```
17      int a = samples[0];
18      int b = samples[0];
19      for (int i=1; i<samples.Length; i++) {
20          if (samples[i] < a) {
21              a = samples[i];
22          }
23          else if (samples[i] > b) {
24              b = samples[i];
25          }
26      }
```

Figure 7-1. *Sample code for reading by stepwise abstraction*

Since it was proposed in the 1970s, reading by stepwise abstraction was well studied and compared with other dynamic testing techniques such as functional testing and structural testing (Basili & Selby, 1987; Juristo & Vegas, 2003; Juristo et al., 2012). The general consensus is that the effectiveness of reading by stepwise abstraction varies significantly from code to code, and code reading shall be combined with other dynamic testing techniques in order to detect different kinds of coding defects.

7.3 Object-Oriented Code Reading

In early times, the code reading techniques were proposed mostly for procedural languages. As OO languages and programming techniques became popular, there was a growing collection of evidence suggesting that early code reading techniques couldn't deal with issues with OO programming (Dunsmore et al., 2001). In the following section, we first discuss the challenges raised in OO code reading and then introduce two reading techniques that address the challenges. We conclude our OO code reading with a summary of empirical findings.

7.3.1 Challenges of Object-Oriented Code Reading

OO programming has three hallmarks: encapsulation, inheritance, and polymorphism. These influence how the code is created, structured, and executed. The OO programming paradigm encourages the distribution of functionality related code elements across the system. Understanding of code frequently requires the understanding of code not in the same class, e.g., in its base class or in other composed class. Polymorphism and late binding make the dynamic behavior of the code hard to comprehend. To fully appreciate the code, one needs to understand its static and dynamic behaviors.

Soloway and Ehrlich (1984) introduced the concept of programming plan, which is a generic fragment of code that represents typical scenarios in programming. They observed that when a programming plan is distributed non-contiguously in a program, it becomes hard to comprehend since only a part of the code is seen at a time and the reader has to guess based on local information. They called this kind of plan delocalized. This delocalized nature is pervasive in OO programming, and Dunsmore et al. (2001) named the characteristic delocalization. Effective OO code reading has to address this delocalization.

In the following we discuss two reading techniques: abstraction-driven reading, which addresses the delocalization nature of the OO code, and use-case-driven reading, which is intended to address the difference between static and dynamic behaviors in OO systems.

7.3.2 Abstraction-Driven Reading

Dunsmore and colleagues extended the idea of stepwise abstraction to OO code reading, and their systematic reading technique is called abstraction-driven reading. In essence, the reading techniques have the following ingredients, as shown in Panel 7-2: Instruction for Abstraction-Driven Reading.

There are many kinds of dependencies and couplings, such as data dependencies and control dependencies. Dunemore et al. didn't provide details on how to quantify them. Skoglund and Kjellgren (2004) used coupling metrics (interaction coupling, component coupling, and inheritance coupling) to measure and rank the classes and methods so that the reading order can be objectively determined.

When developing the abstraction of a method, the reader should identify any changes of state and outputs in terms of inputs and prior states. The specifications should be brief and complete, describing what the method does but not how. A vigilant reader may have noticed that the abstraction development process is similar to the stepwise abstraction discussed earlier.

Abstraction-driven reading is a systematic approach. It encourages a deep understanding of the code and helps the readers stay focused and on track. The abstract specification generated during reading can be used in future code reading. It is a promising technique to address the delocalization nature of OO code. However, abstraction-driven reading has its shortcomings. It is often slow and time-consuming, and it is not designed to address the dynamic nature of OO software.

PANEL 7-2: INSTRUCTION FOR ABSTRACTION-DRIVEN READING

1. Determine the reading order.
   a. Analyze the interdependencies and couplings within the whole object-oriented system. Read the classes with the least amount of dependencies first.
   b. Analyze the methods within classes. Read the methods with the least amount of dependencies first.

2. Read using abstraction.
   a. For each method, reverse-engineer an abstract specification of the method. The method abstract specification may be used to compare with the class specification; it can also be used to support further reading and understanding of other methods (see tracing of referenced methods and classes below).
   b. Trace and understand all referenced classes during reading. This includes reading methods/classes, documentations, previously created abstractions, etc.

7.3.3 Use-Case-Driven Reading

The abstraction-driven reading technique has the potential to discover delocalized defects. To deal with the highly dynamic nature of the OO system behavior, however, additional reading techniques are needed. Use-cases play a significant role in OO system development. For example, they are used to capture the system requirements and play a driving role in the Rational Unified Process (Jacobson et al., 1999). It is natural to use use-cases to guide code reading. We describe use-case-driven reading as originally documented by Dunsmore et al. (2002).

The aim of use-case-driven reading is to check if each and every object behaves correctly in all the possible ways they are used. Specifically, we seek the answers to the following questions: Are correct methods called? Are decisions and state changes made within each method correct and consistent? The reading procedure is described in Panel 7-3: Instruction for Use-Case-Driven Reading.

PANEL 7-3: INSTRUCTION FOR USE-CASE-DRIVEN READING

1. For each use-case in turn, devise a set of scenarios that include preconditions, success or failure conditions, and exceptions.
2. For each scenario derived from a use-case:
   a. Document the expected outcome (e.g., state changes, outputs).
   b. Use a sequence diagram or other diagrams that capture the dynamic aspects of the system. Trace the interactions among participating objects that the scenario dictates by following the message calls.
   c. For the class whose code is under reading, verify that the correct methods of the object of that class are called to support the scenario.
   d. Note any decision and state changes in the method of the class under reading and verify that they are correct and consistent with respect to the scenario.

e. When reading the method code, follow the call to other methods if any. If the called method is in the class under reading, follow the method call, read the method and verify its correctness in a similar fashion; otherwise return and follow the sequence diagram.
f. At the end of the scenario tracing, make a note on the final outcomes and compare them to the expected ones. If there is any difference or anomaly, note the location of the difference and mark it as a defect.

In use-case reading, one devises a number of scenarios from a use-case and examines how the classes deal with those scenarios. It forces the readers to consider object behavior in the given concrete contexts, giving the readers a better idea of whether the code is operating as expected. The readers pay attention to missing/incorrect method calls, erroneous state changes, etc. The readers compare the sequence diagram and the implementing code to verify whether the correct method is called in the right context and whether a side effect of the method call is consistent between the code implementation and the sequence diagram, etc. The readers shall also trace other method calls and ensure their correctness. For defect detection, any difference, inconsistency and missing information, as well as its location in code, is noted and analyzed.

It is not feasible to exercise all scenarios and use-cases. Therefore readers take a dynamic slice of the system. In practice, it will detect fewer defects than other reading techniques. Hence, use-case reading is meant to be complementary to other reading methods.

7.3.4 Empirical Experiences

Dunsmore and colleagues (2002) introduced two new reading techniques, abstraction-driven reading and use-case-driven reading, for OO code and compared them to other reading methods, namely ad hoc reading and checklist-based reading. The details of ad hoc reading and checklist-based reading are covered in Chapter 3. To overcome the known shortcomings of the checklist approach, the authors designed their checklist carefully and based their questions off of historical defect. The final checklist includes 18 carefully ordered questions, covering “where to look” (class-level, method-level, and method-overriding issues) and “how to detect” components.

Dunsmore et al. (2001) reported that there is no significant difference between abstraction-based reading and ad hoc reading in terms of number of defects discovered. However, there is a small improvement using abstraction-based reading. Readers using ad hoc reading went through the code two or three times to build up their understanding, while readers using abstraction-based reading read through the code once, at most twice, albeit slowly. Some defects are completely undetected by all readers using ad hoc reading, but this was not the case for readers using abstraction-based reading. That is, abstraction-based reading has the potential to detect delocalized defects. Compared with ad hoc reading, abstraction-based reading also helps readers stay focused and on track.

Dunsmore et al. (2002; 2003) empirically compared the defect detection capabilities of abstraction-driven reading, use-case-driven reading, and checklist-based reading, using experienced students as subjects. They observed that readers using checklist-based reading found more defects and at a quicker rate. However, the detection performance dropped off sharply after the first 60 minutes. The defect detection of abstraction-driven reading and use-case-driven reading appeared to be similar to each other due to a higher initial overhead. Their defect detection performance leveled off at a later time, but not to the same degree. Readers using the use-case-driven method might have discovered more defects if they were given more time. In terms of the number of false-positive defects reported, checklist-based reading reported the most false positives and use-case-driven reading reported the least. These results are not totally unexpected. Abstraction-driven reading is slow and it aims at full understanding of the code. With use-case-driven reading, one has to generate scenarios before comparing the code and the sequence diagram. The researchers reported that although the performance of abstraction-driven reading is not as strong as that of checklist-based reading, abstraction-driven reading appears to be effective at detecting delocalized defects (but less effective at detecting other defects). Use-case-driven reading had the worst performance among the three studied. However, the method deals with the behaviors in the context of executing systems. Among the three reading techniques studied, no single method detected all defects, and there was not much overlap regarding the kinds of defects detected, suggesting a complementary reading approach would work best. This is in line with the underlining idea behind perspective-based reading (see Chapter 4). The combination of these three reading techniques would have the potential to detect recurring defect types (checklist-based), unusual defects that require deeper understanding (abstraction-driven), and particularly defects that are associated with OO programming (abstraction-driven reading and use-case-driven reading).

Skoglund and Kjellgren (2004) independently conducted two experiments to compare the performance of abstraction-driven reading and checklist-based reading. Their experimental results are inconsistent with those reported by Dunsmore et al. Further, Skoglund and Kjellgren reported that abstraction-driven reading gave more support in understanding the code.

7.4 Object-Oriented Framework Code Reading

Object-oriented framework is getting popular. We discussed scope-based reading for OO application construction in Chapter 6. Here we present an OO framework code reading technique for defect detection developed by Abdelnabi et al. (2004), functionality-based reading.

7.4.1 Why Yet Another Object-Oriented Code Reading Technique?

The OO code reading techniques discussed earlier are presumably applicable to OO framework code reading, so why do we need yet another reading technique? Application frameworks are generalized from existing applications in a specific domain. They have light requirements with no specific or fixed set of use-cases. It is not feasible to define all

possible use-cases the framework is going to support, since the concrete applications have not been instantiated yet when the application framework is being actively developed. The dynamic behavior of the framework is at least incomplete, since the hotspots will be extended by application developers. Therefore, the OO code reading techniques discussed earlier have only a limited use. Additionally, reading framework involves two aspects: code reading and design reading. The latter is crucial, otherwise a framework with very poor design will seriously limit its potential adoption.

Application framework has a steep learning curve, and understanding the framework remains a challenging task. It is important to understand the structure, both static and dynamic, of the framework when reading the framework code for defect detection. It is thus a wise approach to use the framework understanding to guide framework reading for defect detection.

7.4.2 Functionality-Based Approach to Framework Understanding

Chapter 6 discussed the hierarchy-based and example-based approaches. Similarly, an OO application framework can also be understood by first understanding its top-most framework constructs and then general OO constructs. General OO constructs typically include basic constructs (classes and their relationships, such as inheritance and composition) and advanced constructs (e.g., meta-classes and reflection). The top-most framework constructs include:

- *Components*: Here framework components are fully implemented functionalities that application developers can reuse directly.
- *Interfaces*: Framework interfaces are a collection of abstract operations called hotspots, which are customized and implemented by application developers without altering the structure and behavior of the basic framework.
- *Design patterns:* A design pattern is a reusable, proven solution to a commonly occurring problem within a given context.
- *Framelets:* Framelets are small frameworks that package components, interfaces, and design patterns. They are used to structure and document large and complex frameworks.

To understand a framework, one must extract and understand its functionalities, which can be traced to an operation or a set of operations the framework provides or supports. A functionality can expand to, be implemented by, or use another functionality. Functionalities supported by a framework can be categorized as:

- *Do-functionality:* A do-functionality is a fully implemented capability that every instantiation of the framework application must have.
- *Can-functionality:* A can-functionality is not fully implemented, and application developers must supply their own specific code at those hotspots.

- *Offer-functionality:* An offer-functionality is a fully implemented capability, but its use is not mandatory in an instantiated application.

In the functionality-based approach to framework understanding, one reads the framework code with the intent to extract and abstract framework functionalities, trace them to framework operations (methods), and relate them to other functionalities. In the end, the reader compiles "functionality rules." A functionality rule categorizes a functionality (do-functionality, can-functionality, or offer-functionality), documents the code locations where the functionality is implemented, provides a concise and precise description of the functionality, and lists other functionalities this functionality relates to and the relationship type (use, expand, implemented by). Generating the functionality rules is an additional documentation effort that should happen before code reading takes place.

To develop the functionality rules, one reads the framework documents in a top-down manner, from requirements to designs, and to code as needed. The class source code is read recursively, from the top-most classes in the inheritance hierarchy to the derived ones. The implementation class is read to abstract its function. Overriding methods are read to verify, refine, and update the abstractions established earlier. During reading and tracing, the relationships with other functionalities are established, particularly when an object sends a message to another. Lastly, the functionality is classified as a do-, can-, or offer-functionality.

7.4.3 Functionality-Based Reading

Functionality-based reading is motivated by the framework-based reading and understanding of OO frameworks. Its purpose is to trace the functionality to concrete framework constructs and their associated code. It is a hybrid reading technique: It uses the functionality rules as a guidance (top-down) and reads the code from bottom-up. It has steps as shown in Panel 7-4: Instruction for Functionality-Based Reading (Abdelnabi et al., 2004).

PANEL 7-4: INSTRUCTION FOR FUNCTIONALITY-BASED READING

1. Locate the functionality rules. Arrange for their development if they do not exist.
2. Read the functionality rules in order of categories Do-, Can-, and Offer-functionality.
3. For each functionality rule:
   a. Locate the associated method in the lowest level class; read the code with respect to the description of the functionality, and log any discrepancies as a defect.
   b. Locate the related functionalities. Read them for defect detection, if not already inspected.

7.4.4 Empirical Experiences

Abdelnabi et al. (2004) compared functionality-based reading with checklist-based reading and abstraction-driven reading for defect detection in OO frameworks, using students as subjects. The objects were real and professional C++ OO frameworks, with carefully seeded defects of different types. To make it manageable, subjects were asked to inspect about 1000 lines of code. The researchers concluded that functionality-based reading was significantly more effective (more positive total defects detected) and efficient (more positive defects detected per unit of time) than the other two reading techniques.

7.5 Task-Directed Inspection

There are a lot of legacy software applications around, and software developers are tasked to maintain them. Legacy software code may not be well documented, or the documentation is outdated or simply inaccurate. Quite often it is necessary to continuously improve the software quality of a legacy system, particularly for safety-related and mission-critical systems. Kelly and Shepard (2004) introduced so-called task-directed inspection for legacy code reading. Their main idea was to combine code inspection for defect detection with other software development tasks to reduce the potential resistance to the idea of code inspection, thus "task-directed." They also reported a lightweight process far removed from Fagan-style inspection, which is not discussed here.

Based on the particular circumstances when the reading technique was introduced, Kelly and Shepard defined three tasks, all aligned with the objective to produce useful documentation for the legacy software system:

- **Task 1**. Create a data dictionary for the module. A dictionary is simply a catalog of all variables in the module, including their definitions, units of measurement if appropriate, and the meaning of each discrete value if applicable. The roles of these variables in module calling sequences is also of interest. The reader is to confirm that each usage of the variables is consistent with its definition.
- **Task 2**. Document the logic of the module and add a description as comments in code files.
- **Task 3**. Compile a cross-reference between the code and specifications. Cross-reference tags are created and embedded in both code and specification to signal individual matches. Any mismatches or missing of materials in either the code or specification are recoded.

To accomplish these tasks, a reader will have to read and understand the code, trace the data flow and control flow, and cross-check the code and specification. Since the readers have clear objectives in mind, they are forced to scrutinize the code and related document closely.

Modules are assigned to readers deliberately, considering their background and expertise. All three tasks associated with the same module are assigned to a single reader, taking advantage of the potential synergies between tasks. The three different tasks give the inspectors different viewpoints on the source code. Each reader completes their assigned tasks in parallel, and task-directed reading doesn't dictate any interactions among individual readers, if there are any.

Kelly and Shepard also conducted a case study in the industry environment using professional developers as subjects. According to the authors, 50,000 lines-of-code scientific legacy software was read and 950 findings were recorded. Among all these findings, 6% were considered serious defects and received immediate attention for correction, 56% were related to style and maintenance issues, 33% identified inconsistencies between code and specifications, and the remaining 5% were related to enhanced functionalities. The code was inspected at a rate of 20 lines/hour. At completion, the amount of comments in the code increased from 20% to 60%, which is consistent with the code. The experiment was considered a success and the same technique was applied to other software systems.

7.6 Code Readability Factors

We all know that some articles or books are easy to read and understand, while others are not. We also know that the format of the page can affect reading speed. According to Wikipedia, readability is the ease with which a reader can understand written text, and it can be measured in many different ways.

When it comes to code reading, experience tells us that people read code at different speeds with different levels of understanding. However, we do not have a complete account of what impacts code readability. In the early days of computing programming, researchers kept track of eye motions and focus, trying to figure out how people read code. Based on the observational studies, concrete and sometimes radical changes were proposed regarding code display and formatting (Miara et al., 1983; Oman & Cook, 1990).

There are many factors that can affect code readability. Deimel and Naveda (1990) classify these into five categories. Their classification is still relevant today.

- *Reader characteristics*: A reader's experience and knowledge of programming, programming languages, and application domains play a significant role during code reading and comprehension.
- *Intrinsic factors*: Similar to the intrinsic and accidental complexity of a design, code can have its own intrinsic and accidental complexity, which affects its readability. As we learned earlier, object-oriented programming makes delocalization more prevalent, and delocalized code is harder to read and understand.
- *Representational factors*: Representation factors are broad and can include the programming language, whether the code has adequate and accurate comments, the complexity of the design, the naming conventions for variables and methods, etc.
- *Typographic factors*: Typographic factors include font, color-coding of keywords or other programming entities, usage of white space and indentation, etc.

- *Environmental factors*: Environmental factors are meant to contain anything else, e.g., the lighting in the reading spot, the integrated development environment (IDE), etc.

Numerous books, discussions, and postings on programming styles, standards, or conventions are available. We don't want to start another heated debate here. Rather, we make a few suggestions to improve code readability:

- Pick a coding standard, including formatting and indentation, for the team that most of you agree with. Uniformity and consistency will improve code readability.
- Add comments to the code and keep the comments up to date. Don't document facts obvious from the code. Instead capture the design rationale, assumptions, and decisions as comments.
- Choose your variables, function or method names, and other identifiers carefully and wisely. Make sure the names reflect their intentions. Also make sure the names are consistent with the usage in the application domain.
- Use simple programming structures. Be aware of the KISS (keep it simple, stupid) principle. Stay away from nonstandard language features.
- Add white spaces whenever feasible. Don't clutter the display. Logically group your code.

Different stakeholders will read the code you are writing today, including yourself at a later time. It might be true that the code logic is perfectly clear in your mind at the time of writing. However, you will appreciate your efforts to make the code easier to read if you come back to the code again in a few years, or even a few months.

7.7 Summary

Among all types of software artifacts, source code is probably read, reviewed, or inspected most by different people at different times. People read source code for analysis and for construction. While many of these reading techniques discussed earlier can be used for code reading, we focused on specific ones just for code reading in this chapter. Stepwise abstraction is a classic technique and was developed mostly for procedural programming. As the object-oriented programming paradigm gained popularity, the intrinsic complexity of OO code reading became evident. We discussed the abstraction-driven reading technique, which is meant to deal with the strong delocalization inherent to OO code. To cope with the unpredictable dynamic behavior of an OO system from its static code view, use-case-driven reading can be used as a complement. OO framework code reading is even more challenging, and functionality-based reading is meant to make the reading a bit easier. Legacy software applications are abundant and software developers are often tasked with maintaining and extending them. Task-directed reading can be used to improve code quality and complete the necessary documentation on the code and design. As the empirical and observational studies suggest, in the end, no single reading

technique performs best under all conditions. Software developers need to understand the pros and cons of individual reading techniques so that they can call upon different techniques or combine techniques in various ways to fit unique situations.

7.8 References

(Abdelnabi, 2004) Z. Abdelnabi, G. Cantone, M. Ciolkowski, and D. Rombach, Comparing code reading techniques applied to object-oriented software frameworks with regard to effectiveness and defect detection rate, Proceedings of the International Symposium on Empirical Software Engineering, pp.239-248, 2004.

(Basili, 1982) V.R. Basili and H.D. Mills, Understanding and documenting programs, IEEE Trans. Software Engineering, vol.8, no.3, pp.270-283, 1982.

(Basili, 1987) V.R. Basili and R.W. Selby, Comparing the effectiveness of software testing strategies, IEEE Transactions on Software Engineering, vol.13, no.2, pp.1278-1296, 1987.

(Basili, 1996) V. Basili, G. Caldiera, F. Lanubile, and F. Shull, Studies on reading techniques, In Proc. of the Twenty-First Annual Software Engineering Workshop, SEL-96-002, pp.59-65, 1996.

(Brooks, 1983) R. Brooks, Towards a Theory of the Comprehension of Computer Programs, Intl. J. Man-Machine Studies, vol.18, no.6, pp.543-554, 1983.

(Deimel, 1990) L.E. Deimel and J.F. Naveda, Reading computer programs: Instructor's guide and exercises, CMU/SEI-90-EM-3, Carnegie Mellon University, 1990.

(Dunsmore, 2001) A. Dunsmore, M. Roper, and M. Wood, Practical code inspection for object-oriented systems, Proc. of the 1st Workshop on Inspection in Software Engineering, pp.49-57, 2001.

(Dunsmore, 2002) A. Dunsmore, M. Roper, and M. Wood, Further investigations into the development and evaluation of reading techniques for object-oriented code inspection, Proc. of the 24th Int'l Conf. Software Engineering, pp.47-57, 2002.

(Dunsmore, 2003) A. Dunsmore, M. Roper, and M. Wood, Practical code inspection techniques for object-oriented systems: an experimental comparison, IEEE Software, vol.20, no.4, pp.21-29, 2003.

(Dyer, 1992) M. Dyer, Verification-based inspection, Proceedings of the 26th Annual Hawaii International Conference on System Sciences, pp.418-427, 1992.

(Fagan, 1976) M.E. Fagan, Design and code inspections to reduce errors in program development, IBM Systems Journal, vol.15, no.3, pp.182-211, 1976.

(Holmes, 2013) R. Holmes and R.J. Walker, Systematizing pragmatic software reuse, ACM Transactions on Software Engineering and Methodology, vol.21, no.4, Article 20, 2013.

(Jacobson, 1999) I. Jacobson, G. Booch, and J. Rumbaugh, The Unified Software Development Process, Addison-Wesley Professional, 1999.

(Juristo, 2003) N. Juristo and S. Vegas, Functional testing, structural testing, and code reading: what fault type do they each detect? In Empirical Methods and Studies in Software Engineering, vol.2765 of Lecture Notes in Computer Science, pp.208-232, 2003.

(Juristo, 2012) N. Juristo, S. Vegas, M. Solari, S. Abrahao, and I. Ramos, Comparing the effectiveness of equivalence partitioning, branch testing, and code reading by stepwise abstraction applied by subjects, IEEE 5th International Conference on Software Testing, Verification and Validation, pp.330-339, 2012.

(Kelly, 2004) D. Kelly and T. Shepard, Task-directed software inspection, Journal of Systems and Software, vol.73, pp.361-368, 2004.

(Letovsky, 1987) S. Letovsky, Cognitive processes in program comprehension, Journal of Systems and Software, vol.7, no.4, pp.325-339, 1987.

(Linger, 1979) R.C. Linger, H.D. Mills, and B.I. Witt, Structured Programming: Theory and Practice, Addison-Wesley, 1979, chap. 5.

(Littman, 1987) D.C. Littman, J. Pinto, S. Letovsky, and E. Soloway, Mental models and software maintenance, Journal of Systems and Software, vol.7, no.4, pp.341-355, 1987.

(Maalej, 2014) W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, On the comprehension of program comprehension, ACM Transactions on Software Engineering and Methodology, vol.23, no.4, Article 31, 2014.

(Mayrhauser, 1995) A. von Mayrhauser and A.M. Vans, Program comprehension during software maintenance and evolution, IEEE Computer, vol.28, no.8, pp.44-55, 1995.

(Miara, 1983) J.R. Miara, J.A. Musselman, J.A. Navarro, and B. Shneiderman, Program indentation and comprehensibility, Comm. ACM, vol.26, no.11, pp861-867, 1983.

(Oman, 1990) P.W. Oman and C.R. Cook, Typographic style is more than cosmetic, Comm. ACM, vol.33, no.5, pp.506-520, 1990.

(Pennington, 1987) N. Pennington, Stimulus structures and mental representations in expert comprehension of computer programs, Cognitive Psychology, vol.19, pp.295-341, 1987.

(Schulte, 2010) C. Schulte, T. Busjahn, T. Clear, J.H. Paterson, and A. Taherkhani, An introduction to program comprehension from computer science educators, Innovation and Technology in Computer Science Education Working Group Reports ITiCSE-WGR'10, pp.65-86, 2010.

(Selby, 1987) R.W. Selby, V.R. Basili, and F.T. Baker, Cleanroom software development: An empirical evaluation, IEEE Trans. Software Engineering, vol.13, no.9, pp.1027-1037, 1987.

(Shneiderman, 1979) B. Shneiderman and R. Mayer, Syntactic semantic interactions in programmer behavior: A model and experimental results, Intl. J. Comp. & Info. Sciences, vol.8, no.3, pp.219-238, 1979.

(Skoglund, 2004) M. Skoglund and V. Kjellgren, An experimental comparison of the effectiveness and usefulness of inspection techniques for object-oriented programs, Proceedings of the Conference on Empirical Assessment in Software Engineering (EASE'04), Edinburgh, Scotland, 2004.

(Soloway, 1984) E. Soloway and K. Ehrlich, Empirical studies of programming knowledge, IEEE Trans. Software Engineering, vol.10, no.5, pp.595-609, 1984.

(Storey, 2005) M-A. Storey, Theories, methods and tools in programming comprehension: Past, present and future, Proceedings of the 13th International Workshop on Program Comprehension, 2005.

CHAPTER 8

Conclusion

We have explored general and specific reading techniques for software artifacts such as requirements, design, code, and usability. Those techniques are summarized here, with the chapter number where it is introduced. It is a big list. There is no general consensus which reading technique is most effective. We doubt there will ever be for all types of software artifacts in all development contexts. The outcome of software reading is context- and situation-dependent and varies individually.

- *Ad hoc reading (Chapter 3)*. No reading instruction is provided in ad hoc reading and the results strongly depend on an individual's skill and experiences. It can be applied to any software artifacts and is used widely in practice.
- *Checklist-based reading (Chapter 3)*. It can be applied to any software artifacts and is used widely in practice. Checklists are usually artifact-specific and built from historical data or learned from others but customized to project or team's needs. Checklist-based reading enables readers to detect defects covered by the checklist and may limit readers to detect defects not covered by the checklist.
    - As a variant, an active guidance can be provided how to use the checklist. The guidance can also be implicitly embedded in the structure or ordering of checklist items.
- *Differential reading (Chapter 3)*. It can be applied to any artifacts that are changed. Two versions of the document are displayed side by side or in a single view, and changes are highlighted and examined in context.
- *Defect-based reading (Chapter 4)*. In defect-based reading, each reader is given specific steps to discover a particular class of defects. Multiple readers or a review team collectively detect more defects. Defect classes have to be understood and analyzed before specific reading steps can be devised to target specific defect classes. Defect classes are typically software artifact-dependent. In theory defect-based reading can be applied to any software artifact types, but only applications to software requirements were reported.

© Yang-Ming Zhu 2016
Y.-M. Zhu, *Software Reading Techniques*, DOI 10.1007/978-1-4842-2346-8_8

- *Perspective-based reading (Chapter 4)*. In perspective-based reading, each reader reads a software artifact using one specific operational scenario from a particular perspective, and the combination of different and complementary perspectives provides a better coverage of the artifact. It is adaptable to different artifacts and tailorable to organizational or project setting and assigns specific reading responsibilities to readers and provides well-defined reading instructions to focus reading analysis efforts. It has been gaining increased industry adoption in detecting defects in requirements, design, code, and usability. In fact, perspective-based reading includes a family of reading techniques. For a given artifact, key stakeholders shall be identified and then specific reading scenarios shall be developed. For the same stakeholders, different reading scenarios shall be prepared and applied for inspecting different artifact types, since the concerns vary as artifact types vary.
- *Function-point-based reading (Chapter 4)*. Function-point scenarios are alternative to partition software requirements and the scenarios augmented with checklists can be used to focus requirements reading.
- *Combined reading of requirements (Chapter 5)*. Checklist-based reading, defect-based reading, and perspective-based reading have different defect detection capabilities in detecting different classes of defects in requirements. Combined reading takes advantage of the strength of the aforementioned three reading techniques while compensating each other's weaknesses.
- *Test-case-driven reading (Chapter 5)*. Test-case-driven reading employs real testers to screen requirements by writing test cases. The purpose is to inspect the requirements effectively with minimal cost and amortizes the effort across different development phases. It is applicable to organizations with limited resources and can provide good-enough requirements for project decision making early in the software development lifecycle.
- *Usage-based reading (Chapter 6)*. The goal of software reading should not be to find as many defects as possible but to find the most critical defects that matter to users. Usage-based reading is just for that and has been applied to design reading. To do that, use cases prioritized by experts are used to drive and focus design reading. It has a couple variants, e.g., use cases can be prioritized by individual readers themselves.

- *Traceability-based reading (Chapter 6)*. This seven-member family of reading is applicable to high-level object-oriented design and documents, and the techniques are organized into horizontal reading and vertical reading. Which reading technique to use depends on the availability of design diagrams and descriptions. As documents may represent different levels of abstractions of the same system, special attention shall be paid to semantic checking, which proves to be challenging. It certainly takes coordination among all readers using different reading techniques on different artifacts.

  - The group of horizontal reading has four reading techniques that trace information between design documents, including diagrams and textual descriptions for design consistency.
  - The group of vertical reading has three reading techniques that trace information between design and requirements for design correctness and completeness.

- *Architecture reading (Chapter 6)*. The ideas behind traceability-based reading are applied to architecture reading. Horizontal reading compares module views and component-and-connector views and module views and allocation views to detect any inconsistency. Vertical reading checks the information captured in module views, component-and-connector views, and allocation views to be consistent with requirements.
- *Scope-based reading (Chapter 6)*. While all other reading techniques intend to detect defects in software artifacts, scope-based reading intends to aid software construction using object-oriented frameworks.

  - Hierarchy-based reading concentrates readers on class hierarchy in order to learn and use the framework functionalities.
  - Example-based reading is simply motivated by learning from examples. It guides readers through examples in order to reuse or extend functionalities exhibited in examples.

- *Stepwise abstraction (Chapter 7)*. This classical technique is for reading any programming code in a bottom-up fashion. It abstracts the programming code recursively and compares the abstraction to the intended specification in order to detect defects.

- *Abstraction-driven reading (Chapter 7)*. This reading technique is applicable to object-oriented code. It builds atop of stepwise abstraction and adds a step to determine the reading order, considering interdependencies among code modules. It copes with the delocalization nature of object-oriented code and focuses on the static code behavior.
- *Use-case-driven reading (Chapter 7)*. While abstraction-driven reading focuses on static behavior, use-case-driven reading emphasizes the dynamic behavior of an object-oriented system. It uses use-cases to slice the software and thus does not provide a complete coverage of code. It works best when use-case-driven reading is utilized as a complementary instrument to other reading techniques.
- *Functionality-based reading (Chapter 7)*. This is intended for object-oriented framework code and design reading. Framework code reading poses a unique challenge as there is no concrete requirement or fixed use case for it – it provides an application skeleton to enable development of other applications. A framework can be understood from the functionalities it provides. And the functionality-based reading combines top-down and bottom-up reading strategies.
- *Task-directed inspection (Chapter 7)*. This is designed to piggyback code review or inspection for defect detection as part of otherwise required tasks that are part of project effort and management mandate. Legacy code is abundant and there is not much documentation (comments and design rationale) for it. Task-directed inspection can be exploited to complete documentation, at the same time improving code quality.

Like design patterns, software reading techniques codify best practices for software review and inspection. Now those techniques are at your disposal. It is expected that those techniques will be tailored to your unique circumstances for their best effectiveness. You could customize, modify, subset, or extend a reading technique, or combine good ideas from multiple reading techniques to create your own. Agile software development has been widely practiced nowadays. Since agile practitioners favor working software over comprehensive documentation, there may be no complete or explicitly written documentation for requirements specification, architecture description, and design. It remains interesting to figure out how reading techniques can be effectively customized and applied in that context.

Software reading is a learned skill. You are encouraged to apply them in order to cultivate your own ability and improve personal effectiveness. You are also encouraged to share with others your experience and lessons learned with those reading techniques.

Index

A

Abstraction-driven reading, 5, 52, 108–111, 114, 116, 122
Active design review, 4, 7, 8, 13–15, 18, 35, 39, 41, 43, 46, 96
Active guidance, 26, 81, 119
Activity diagram, 48, 51, 82
Ad hoc partition, 64
Ad hoc reading, 4, 23–25, 28, 29, 32, 35, 37, 40, 41, 47, 57, 58, 64, 69, 75, 77, 110, 119
Analytical model for perspective-based reading, 4, 58, 60
Application framework, 5, 97, 98, 100, 111, 112
Architecture reading, 93–96, 100, 121
Architecture review, 5, 15, 93, 96
Architecture tradeoff analysis (ATAM), 15, 96
Artifact-based checklist, 24, 36
Assimilation process, 104

B

Bad fixes, 10
Base class, 27, 98, 107
Beacon, 104
Best practice, 8, 18, 23, 32, 50, 122
Black box, 97
Bottom-up, 52, 104–106, 113, 121, 122

C

Checklists
- based reading (CBR), 4, 23–29, 32, 35, 37, 39–42, 47, 51, 54, 58, 60–64, 69, 70, 72, 74, 75, 80, 81, 95, 110, 111, 114, 119, 120
- best practice of, 26, 28
- with guidance, 28

Class diagram, 48–51, 83–85, 87–88, 91
Class hierarchy, 86, 88, 97, 98, 121
Code
- inspection, 2, 8, 9, 12, 16, 17, 58, 103, 114
- reading, 5, 26, 27, 29, 52–54, 79, 80, 103–117, 122
- review, 3, 16, 23–25, 29, 122

Cognitive
- analysis, 58–60, 66
- bias, 17
- process, 4, 21
- structure, 104

Collaboration diagram, 48–51
Combined reading of requirements, 70–72, 120
Commission defect, 38, 95
Completeness, 24, 49, 50, 65, 73, 81, 121
Consistency, 4, 14, 24, 50, 55, 57, 58, 81, 84, 92, 116, 121
Control flow, 105, 114
Correctness, 15, 24, 48, 49, 51, 53, 58, 65, 81, 105, 110, 121

D

Data flow, 105, 114
Data-flow diagram, 74–75
Defect-based reading (DBR), 4, 35, 37–42, 64, 66, 69, 70, 72, 75, 82, 119, 120
Defects
- collection, 4, 7, 11, 16, 51, 54, 58, 60, 75
- containment, 1
- detection, 1, 3, 4, 7, 17, 23, 25, 35, 36, 40, 41, 45, 47, 51, 54, 60–65, 70, 72, 74, 75, 79, 80, 97, 100, 106, 110–114, 120, 122

© Yang-Ming Zhu 2016

Y.-M. Zhu, *Software Reading Techniques*, DOI 10.1007/978-1-4842-2346-8

Defects (*cont.*)
injection, 1, 3
prevention, 1
removal, 1
Delocalization, 108, 115, 116, 122
Design
inspection, 2, 77, 81, 100
pattern, 23, 112, 122
reading, 48–51, 53, 77–100, 112, 120, 122
Differential reading, 4, 29–32, 119
Discrepancy, 53, 83, 85–88, 90, 91
Distributed development, 1
Divide and conquer, 104
Dynamic behavior, 5, 48, 82, 90, 107, 108, 112, 116, 122
Dynamic testing, 107

E

Educational background, 75
Empirical evidence, 16, 21, 72
Empirical experience, 28–29, 41–42, 47–48, 51, 54, 57, 69, 72, 74, 79–81, 92–93, 95–96, 99–100, 110–111, 114
Encapsulation, 25, 86, 91, 107
Environmental factors, 116
Escaped defects, 1
Example-based reading, 97–99, 121

F

Fagan inspection, 2, 4, 7–13, 15–18, 22, 28
False positives, 4, 8, 10–12, 28, 74, 93, 111
Follow-up, 7, 8, 10, 74, 96
Framework, 5, 10, 21, 63, 94, 97–100, 103, 111–114, 116, 121, 122
Functionality-based reading, 103, 111–114, 116, 122
Function point analysis, 65
Function-point-based reading, 4, 66, 120
Function point scenarios, 65, 120

G

General reading techniques, 4, 69, 77

H

Hierarchy-based reading, 97, 98, 121
Horizontal reading, 83–88, 91, 92, 94, 100, 121
Hotspot, 97, 112
Human-computer interaction (HCI), 55

I, J

Individual factor, 18, 74–75
Individual performance, 16
Information overloading, 81
Information processing, 74
Inheritance, 25, 27, 84, 86, 88, 107, 108, 112, 113
Inspection
maxims, 17
roles, 9
Intrinsic factors, 115

K

Knowledge base, 104

L

Late binding, 107
Legacy software, 114–116
Lifecycle, 18, 75, 83, 96, 120

M

Management review, 2, 7, 15
Meeting-based review, 11
Meeting-less inspection, 12
Mental model, 36
Message, 49, 50, 84–87, 89–91, 109, 113
Metrics, 12, 16, 18, 108
Model-based, 96
Modern code review, 16

N

N-fold inspection, 12
Non-meeting-based review, 11

O

Object-oriented (OO), 5, 25–27, 29, 48, 52, 79, 82, 103, 107–116, 121, 122
Omission defects, 38, 42, 70, 72, 96
Operational scenarios, 35, 36, 39, 41–43, 46, 120
Orthogonal perspectives, 55

P

Pair programming, 15
Paradox of structure, 23, 26
Peer review, 2, 7
Personality type, 74, 75
Perspective-based
  code reading, 52–54
  design reading, 48–51
  reading, 4, 5, 23, 35, 37, 42–64, 66, 69, 70, 73, 77, 79, 93, 96, 100, 111, 120
  requirements reading, 45–48
  usability reading, 54–57
Phased inspection, 12
Polymorphism, 25, 107
Prime program, 106
Prioritized use cases, 4, 78–81, 100
Program comprehension, 104, 105
Program plan, 104
Property-based checklist, 24
Purpose of software reading, 4, 22–23

Q

Questionnaire, 13, 14

R

Rank-based, 78, 79
Reader characteristics, 115
Reading for analysis, 22, 97, 103
Reading for construction, 23, 97, 103
Reading scenarios, 35, 39, 40, 43–44, 46, 48–57, 60, 64, 73, 120
Reading techniques, 4–5, 10, 17, 18, 35–66, 69–75, 77–100, 103–117, 119–122
Representational factors, 115
Requirements reading, 41, 45, 65, 69–75, 120
Requirements specification, 2, 4, 5, 12, 22, 24, 25, 29, 37, 38, 42, 44–46, 48, 50, 58, 64–66, 69, 70, 72, 74, 75, 77, 82, 83, 88, 100, 122
Reuse, 23, 29, 45, 53, 73, 97, 98, 103–105, 112, 121
Root cause analysis, 3, 12

S

Sampling-driven inspection, 12
Scenario-based, 96
Scenario-based reading, 4, 15, 22, 32, 35–66, 75
Scenario template, 43
Scope-based reading, 5, 97–100, 111, 121
Semantic checking, 91, 100, 121
Semi-structured reading, 23, 26
Sequence diagram, 48, 49, 51, 80, 83–87, 89–91, 109–111
Service, 83–85, 87, 89–91
Software
  architecture, 15, 77, 93, 94, 96
  artifacts, 2–5, 8–12, 15, 16, 18, 21–24, 28–30, 32, 35–37, 39, 42, 43, 45, 46, 51, 52, 55, 58, 75, 77, 80, 82, 100, 116, 119–121
  audit, 15
  design, 21, 77, 82, 100
  engineering, 1, 2, 5, 16, 22, 28, 47, 74, 75, 80, 96, 97, 99, 103, 104
  implementation, 1, 77
  inspection, 1–3, 5, 7, 16, 18, 22, 24, 43, 58, 75
  lifecycle, 18
  product, 1, 7, 22, 46, 69, 71
  quality, 1–3, 5, 8, 78, 114
  quality insurance, 1, 5, 8
  reading, 4, 5, 16, 37, 59, 74, 75, 77, 83, 97, 105, 119, 120, 122
  reading techniques, 4, 10, 16, 18, 21–32, 35, 48, 83, 122
  requirements, 5, 37, 44, 46, 69, 70, 75, 119, 120
  review, 1–4, 21, 24, 32, 35, 43, 122
  review procedure, 4, 7–18
  specification, 5, 22, 44, 48, 69, 70, 75
Specific reading techniques, 4, 100, 103, 119
Stakeholders, 42–44, 46, 48, 49, 52, 64, 70, 72, 73, 94–96, 116, 120
State diagram, 48, 83–87, 90–91
Static analysis, 2
Static behavior, 5, 84, 89, 122
Static structure, 48
Stepwise abstraction, 5, 23, 52, 103, 105–108, 116, 121, 122
Structured program, 48, 106
Structured reading, 23

Subject matter experts, 11, 14, 44
Syntactic checking, 84, 91
Systematic reading, 23, 36, 108

T

Task-directed inspection, 114–115, 122
Taxonomy of defects, 38
Taxonomy of software reading techniques, 23
Team size, 12, 16
Technical review, 2, 7, 15
Testability, 51, 73, 94
Test case, 2, 3, 8, 11, 22–24, 29, 43–48, 50, 51, 53, 72–74
Test-case driven reading, 72–75, 120
Test-driven reading, 4
Testing, 1–3, 5, 29, 37, 38, 44–46, 50, 54, 73, 77, 107
Test plan, 2, 3, 8, 22, 44, 74
Textual description, 48, 79, 81, 83, 88, 91, 121
Time-boxed, 78
Top-down, 104–106, 113, 122
Traceability-based reading, 4, 5, 66, 77, 79, 81–96, 100, 121
Two-person inspection, 12, 52
Typographic factors, 115

U

UML diagrams, 48, 51, 80–83
Unified Modeling Language (UML), 48, 51, 80–83
Unstructured reading, 23
Unsystematic reading, 23, 36
Usability goal, 55, 56
Usability inspection, 5, 54–57, 66
Usability testing, 54
Usage-based reading, 4, 77–81, 100, 120
Use-based reading, 55, 57
Use-case diagram, 48, 80
Use-case-driven reading, 5, 52, 79, 80, 108–111, 116, 122
User documentation, 2, 8, 22
User interface, 4, 42, 44, 45, 54–58, 71

V

Vertical reading, 5, 79, 83, 88–92, 94, 95, 100, 121

W, X, Y, Z

Walkthrough, 2, 7, 15, 55
White box, 50, 97

SOFTWARE TESTING AND QUALITY ASSURANCE

THEORY AND PRACTICE

Kshirasagar Naik
Priyadarshi Tripathy

WILEY

SOFTWARE TESTING AND QUALITY ASSURANCE

Theory and Practice

KSHIRASAGAR NAIK

Department of Electrical and Computer Engineering
University of Waterloo, Waterloo

PRIYADARSHI TRIPATHY

NEC Laboratories America, Inc.

WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

SOFTWARE TESTING AND QUALITY ASSURANCE

SOFTWARE TESTING AND QUALITY ASSURANCE

Theory and Practice

KSHIRASAGAR NAIK

Department of Electrical and Computer Engineering
University of Waterloo, Waterloo

PRIYADARSHI TRIPATHY

NEC Laboratories America, Inc.

WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

Copyright © 2008 by John Wiley & Sons, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey
Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permission.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data:

Naik, Kshirasagar, 1959–
Software testing and quality assurance / Kshirasagar Naik and Priyadarshi Tripathy.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-471-78911-6 (cloth)
1. Computer software—Testing. 2. Computer software—Quality control. I. Tripathy, Piyu, 1958–II. Title.
QA76.76.T48N35 2008
005.14—dc22
2008008331

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To our parents
Sukru and Teva Naik
Kunjabihari and Surekha Tripathy

CONTENTS

Preface xvii

List of Figures xxi

List of Tables xxvii

CHAPTER 1 *BASIC CONCEPTS AND PRELIMINARIES* 1

1.1 Quality Revolution 1
1.2 Software Quality 5
1.3 Role of Testing 7
1.4 Verification and Validation 7
1.5 Failure, Error, Fault, and Defect 9
1.6 Notion of Software Reliability 10
1.7 Objectives of Testing 10
1.8 What Is a Test Case? 11
1.9 Expected Outcome 12
1.10 Concept of *Complete* Testing 13
1.11 Central Issue in Testing 13
1.12 Testing Activities 14
1.13 Test Levels 16
1.14 Sources of Information for Test Case Selection 18
1.15 White-Box and Black-Box Testing 20
1.16 Test Planning and Design 21
1.17 Monitoring and Measuring Test Execution 22
1.18 Test Tools and Automation 24
1.19 Test Team Organization and Management 26
1.20 Outline of Book 27
References 28
Exercises 30

CHAPTER 2 *THEORY OF PROGRAM TESTING* 31

2.1 Basic Concepts in Testing Theory 31
2.2 Theory of Goodenough and Gerhart 32
2.2.1 Fundamental Concepts 32
2.2.2 Theory of Testing 34
2.2.3 Program Errors 34
2.2.4 Conditions for Reliability 36
2.2.5 Drawbacks of Theory 37
2.3 Theory of Weyuker and Ostrand 37

2.4 Theory of Gourlay 39
2.4.1 Few Definitions 40
2.4.2 Power of Test Methods 42
2.5 Adequacy of Testing 42
2.6 Limitations of Testing 45
2.7 Summary 46
Literature Review 47
References 48
Exercises 49

CHAPTER 3 *UNIT TESTING* 51

3.1 Concept of Unit Testing 51
3.2 Static Unit Testing 53
3.3 Defect Prevention 60
3.4 Dynamic Unit Testing 62
3.5 Mutation Testing 65
3.6 Debugging 68
3.7 Unit Testing in eXtreme Programming 71
3.8 JUnit: Framework for Unit Testing 73
3.9 Tools for Unit Testing 76
3.10 Summary 81
Literature Review 82
References 84
Exercises 86

CHAPTER 4 *CONTROL FLOW TESTING* 88

4.1 Basic Idea 88
4.2 Outline of Control Flow Testing 89
4.3 Control Flow Graph 90
4.4 Paths in a Control Flow Graph 93
4.5 Path Selection Criteria 94
4.5.1 All-Path Coverage Criterion 96
4.5.2 Statement Coverage Criterion 97
4.5.3 Branch Coverage Criterion 98
4.5.4 Predicate Coverage Criterion 100
4.6 Generating Test Input 101
4.7 Examples of Test Data Selection 106
4.8 Containing Infeasible Paths 107
4.9 Summary 108
Literature Review 109
References 110
Exercises 111

CHAPTER 5 *DATA FLOW TESTING* 112

5.1 General Idea 112
5.2 Data Flow Anomaly 113
5.3 Overview of Dynamic Data Flow Testing 115
5.4 Data Flow Graph 116

5.5 Data Flow Terms 119
5.6 Data Flow Testing Criteria 121
5.7 Comparison of Data Flow Test Selection Criteria 124
5.8 Feasible Paths and Test Selection Criteria 125
5.9 Comparison of Testing Techniques 126
5.10 Summary 128
Literature Review 129
References 131
Exercises 132

CHAPTER 6 *DOMAIN TESTING* 135

6.1 Domain Error 135
6.2 Testing for Domain Errors 137
6.3 Sources of Domains 138
6.4 Types of Domain Errors 141
6.5 ON and OFF Points 144
6.6 Test Selection Criterion 146
6.7 Summary 154
Literature Review 155
References 156
Exercises 156

CHAPTER 7 *SYSTEM INTEGRATION TESTING* 158

7.1 Concept of Integration Testing 158
7.2 Different Types of Interfaces and Interface Errors 159
7.3 Granularity of System Integration Testing 163
7.4 System Integration Techniques 164
7.4.1 Incremental 164
7.4.2 Top Down 167
7.4.3 Bottom Up 171
7.4.4 Sandwich and Big Bang 173
7.5 Software and Hardware Integration 174
7.5.1 Hardware Design Verification Tests 174
7.5.2 Hardware and Software Compatibility Matrix 177
7.6 Test Plan for System Integration 180
7.7 Off-the-Shelf Component Integration 184
7.7.1 Off-the-Shelf Component Testing 185
7.7.2 Built-in Testing 186
7.8 Summary 187
Literature Review 188
References 189
Exercises 190

CHAPTER 8 *SYSTEM TEST CATEGORIES* 192

8.1 Taxonomy of System Tests 192
8.2 Basic Tests 194
8.2.1 Boot Tests 194
8.2.2 Upgrade/Downgrade Tests 195

8.2.3 Light Emitting Diode Tests **195**
8.2.4 Diagnostic Tests **195**
8.2.5 Command Line Interface Tests **196**
8.3 Functionality Tests **196**
8.3.1 Communication Systems Tests **196**
8.3.2 Module Tests **197**
8.3.3 Logging and Tracing Tests **198**
8.3.4 Element Management Systems Tests **198**
8.3.5 Management Information Base Tests **202**
8.3.6 Graphical User Interface Tests **202**
8.3.7 Security Tests **203**
8.3.8 Feature Tests **204**
8.4 Robustness Tests **204**
8.4.1 Boundary Value Tests **205**
8.4.2 Power Cycling Tests **206**
8.4.3 On-Line Insertion and Removal Tests **206**
8.4.4 High-Availability Tests **206**
8.4.5 Degraded Node Tests **207**
8.5 Interoperability Tests **208**
8.6 Performance Tests **209**
8.7 Scalability Tests **210**
8.8 Stress Tests **211**
8.9 Load and Stability Tests **213**
8.10 Reliability Tests **214**
8.11 Regression Tests **214**
8.12 Documentation Tests **215**
8.13 Regulatory Tests **216**
8.14 Summary **218**
Literature Review **219**
References **220**
Exercises **221**

CHAPTER 9 *FUNCTIONAL TESTING* **222**

9.1 Functional Testing Concepts of Howden **222**
9.1.1 Different Types of Variables **224**
9.1.2 Test Vector **230**
9.1.3 Testing a Function in Context **231**
9.2 Complexity of Applying Functional Testing **232**
9.3 Pairwise Testing **235**
9.3.1 Orthogonal Array **236**
9.3.2 In Parameter Order **240**
9.4 Equivalence Class Partitioning **244**
9.5 Boundary Value Analysis **246**
9.6 Decision Tables **248**
9.7 Random Testing **252**
9.8 Error Guessing **255**
9.9 Category Partition **256**
9.10 Summary **258**

Literature Review 260
References 261
Exercises 262

CHAPTER 10 *TEST GENERATION FROM FSM MODELS* 265

10.1 State-Oriented Model 265
10.2 Points of Control and Observation 269
10.3 Finite-State Machine 270
10.4 Test Generation from an FSM 273
10.5 Transition Tour Method 273
10.6 Testing with State Verification 277
10.7 Unique Input–Output Sequence 279
10.8 Distinguishing Sequence 284
10.9 Characterizing Sequence 287
10.10 Test Architectures 291
10.10.1 Local Architecture 292
10.10.2 Distributed Architecture 293
10.10.3 Coordinated Architecture 294
10.10.4 Remote Architecture 295
10.11 Testing and Test Control Notation Version 3 (TTCN-3) 295
10.11.1 Module 296
10.11.2 Data Declarations 296
10.11.3 Ports and Components 298
10.11.4 Test Case Verdicts 299
10.11.5 Test Case 300
10.12 Extended FSMs 302
10.13 Test Generation from EFSM Models 307
10.14 Additional Coverage Criteria for System Testing 313
10.15 Summary 315
Literature Review 316
References 317
Exercises 318

CHAPTER 11 *SYSTEM TEST DESIGN* 321

11.1 Test Design Factors 321
11.2 Requirement Identification 322
11.3 Characteristics of Testable Requirements 331
11.4 Test Objective Identification 334
11.5 Example 335
11.6 Modeling a Test Design Process 345
11.7 Modeling Test Results 347
11.8 Test Design Preparedness Metrics 349
11.9 Test Case Design Effectiveness 350
11.10 Summary 351
Literature Review 351
References 353
Exercises 353

CHAPTER 12 *SYSTEM TEST PLANNING AND AUTOMATION* **355**

12.1 Structure of a System Test Plan **355**
12.2 Introduction and Feature Description **356**
12.3 Assumptions **357**
12.4 Test Approach **357**
12.5 Test Suite Structure **358**
12.6 Test Environment **358**
12.7 Test Execution Strategy **361**
12.7.1 Multicycle System Test Strategy **362**
12.7.2 Characterization of Test Cycles **362**
12.7.3 Preparing for First Test Cycle **366**
12.7.4 Selecting Test Cases for Final Test Cycle **369**
12.7.5 Prioritization of Test Cases **371**
12.7.6 Details of Three Test Cycles **372**
12.8 Test Effort Estimation **377**
12.8.1 Number of Test Cases **378**
12.8.2 Test Case Creation Effort **384**
12.8.3 Test Case Execution Effort **385**
12.9 Scheduling and Test Milestones **387**
12.10 System Test Automation **391**
12.11 Evaluation and Selection of Test Automation Tools **392**
12.12 Test Selection Guidelines for Automation **395**
12.13 Characteristics of Automated Test Cases **397**
12.14 Structure of an Automated Test Case **399**
12.15 Test Automation Infrastructure **400**
12.16 Summary **402**
Literature Review **403**
References **405**
Exercises **406**

CHAPTER 13 *SYSTEM TEST EXECUTION* **408**

13.1 Basic Ideas **408**
13.2 Modeling Defects **409**
13.3 Preparedness to Start System Testing **415**
13.4 Metrics for Tracking System Test **419**
13.4.1 Metrics for Monitoring Test Execution **420**
13.4.2 Test Execution Metric Examples **420**
13.4.3 Metrics for Monitoring Defect Reports **423**
13.4.4 Defect Report Metric Examples **425**
13.5 Orthogonal Defect Classification **428**
13.6 Defect Causal Analysis **431**
13.7 Beta Testing **435**
13.8 First Customer Shipment **437**
13.9 System Test Report **438**
13.10 Product Sustaining **439**
13.11 Measuring Test Effectiveness **441**
13.12 Summary **445**
Literature Review **446**

References 447
Exercises 448

CHAPTER 14 *ACCEPTANCE TESTING* 450

14.1 Types of Acceptance Testing 450
14.2 Acceptance Criteria 451
14.3 Selection of Acceptance Criteria 461
14.4 Acceptance Test Plan 461
14.5 Acceptance Test Execution 463
14.6 Acceptance Test Report 464
14.7 Acceptance Testing in eXtreme Programming 466
14.8 Summary 467
Literature Review 468
References 468
Exercises 469

CHAPTER 15 *SOFTWARE RELIABILITY* 471

15.1 What Is Reliability? 471
15.1.1 Fault and Failure 472
15.1.2 Time 473
15.1.3 Time Interval between Failures 474
15.1.4 Counting Failures in Periodic Intervals 475
15.1.5 Failure Intensity 476
15.2 Definitions of Software Reliability 477
15.2.1 First Definition of Software Reliability 477
15.2.2 Second Definition of Software Reliability 478
15.2.3 Comparing the Definitions of Software Reliability 479
15.3 Factors Influencing Software Reliability 479
15.4 Applications of Software Reliability 481
15.4.1 Comparison of Software Engineering Technologies 481
15.4.2 Measuring the Progress of System Testing 481
15.4.3 Controlling the System in Operation 482
15.4.4 Better Insight into Software Development Process 482
15.5 Operational Profiles 482
15.5.1 Operation 483
15.5.2 Representation of Operational Profile 483
15.6 Reliability Models 486
15.7 Summary 491
Literature Review 492
References 494
Exercises 494

CHAPTER 16 *TEST TEAM ORGANIZATION* 496

16.1 Test Groups 496
16.1.1 Integration Test Group 496
16.1.2 System Test Group 497
16.2 Software Quality Assurance Group 499
16.3 System Test Team Hierarchy 500

16.4 Effective Staffing of Test Engineers 501
16.5 Recruiting Test Engineers 504
16.5.1 Job Requisition 504
16.5.2 Job Profiling 505
16.5.3 Screening Resumes 505
16.5.4 Coordinating an Interview Team 506
16.5.5 Interviewing 507
16.5.6 Making a Decision 511
16.6 Retaining Test Engineers 511
16.6.1 Career Path 511
16.6.2 Training 512
16.6.3 Reward System 513
16.7 Team Building 513
16.7.1 Expectations 513
16.7.2 Consistency 514
16.7.3 Information Sharing 514
16.7.4 Standardization 514
16.7.5 Test Environments 514
16.7.6 Recognitions 515
16.8 Summary 515
Literature Review 516
References 516
Exercises 517

CHAPTER 17 *SOFTWARE QUALITY* 519

17.1 Five Views of Software Quality 519
17.2 McCall's Quality Factors and Criteria 523
17.2.1 Quality Factors 523
17.2.2 Quality Criteria 527
17.2.3 Relationship between Quality Factors and Criteria 527
17.2.4 Quality Metrics 530
17.3 ISO 9126 Quality Characteristics 530
17.4 ISO 9000:2000 Software Quality Standard 534
17.4.1 ISO 9000:2000 Fundamentals 535
17.4.2 ISO 9001:2000 Requirements 537
17.5 Summary 542
Literature Review 544
References 544
Exercises 545

CHAPTER 18 *MATURITY MODELS* 546

18.1 Basic Idea in Software Process 546
18.2 Capability Maturity Model 548
18.2.1 CMM Architecture 549
18.2.2 Five Levels of Maturity and Key Process Areas 550
18.2.3 Common Features of Key Practices 553
18.2.4 Application of CMM 553
18.2.5 Capability Maturity Model Integration (CMMI) 554

18.3 Test Process Improvement **555**
18.4 Testing Maturity Model **568**
18.5 Summary **578**
Literature Review **578**
References **579**
Exercises **579**

GLOSSARY **581**

INDEX **600**

PREFACE

karmany eva dhikaras te; ma phalesu kadachana; ma karmaphalahetur bhur; ma te sango stv akarmani.
Your right is to work only; but never to the fruits thereof; may you not be motivated by the fruits of actions; nor let your attachment to be towards inaction.
— *Bhagavad Gita*

We have been witnessing tremendous growth in the software industry over the past 25 years. Software applications have proliferated from the original data processing and scientific computing domains into our daily lives in such a way that we do not realize that some kind of software executes when we do even something ordinary, such as making a phone call, starting a car, turning on a microwave oven, and making a debit card payment. The processes for producing software must meet two broad challenges. First, the processes must produce low-cost software in a short time so that corporations can stay competitive. Second, the processes must produce usable, dependable, and safe software; these attributes are commonly known as quality attributes. Software quality impacts a number of important factors in our daily lives, such as economy, personal and national security, health, and safety.

Twenty-five years ago, testing accounted for about 50% of the total time and more than 50% of the total money expended in a software development project—and, the same is still true today. Those days the software industry was a much smaller one, and academia offered a single, comprehensive course entitled *Software Engineering* to educate undergraduate students in the nuts and bolts of software development. Although software testing has been a part of the classical software engineering literature for decades, the subject is seldom incorporated into the mainstream undergraduate curriculum. A few universities have started offering an *option* in software engineering comprising three specialized courses, namely, *Requirements Specification*, *Software Design*, and *Testing and Quality Assurance*. In addition, some universities have introduced full undergraduate and graduate degree programs in software engineering.

Considering the impact of software quality, or the lack thereof, we observe that software testing education has not received its due place. Ideally, research should lead to the development of tools and methodologies to produce low-cost, high-quality software, and students should be educated in the testing fundamentals. In other words, software testing research should not be solely academic in nature but must strive to be practical for industry consumers. However, in practice, there

is a large gap between the testing skills needed in the industry and what are taught and researched in the universities.

Our goal is to provide the students and the teachers with a set of well-rounded educational materials covering the fundamental developments in testing theory and common testing practices in the industry. We intend to provide the students with the "big picture" of testing and quality assurance, because software quality concepts are quite broad. There are different kinds of software systems with their own intricate characteristics. We have not tried to specifically address their testing challenges. Instead, we have presented testing theory and practice as broad stepping stones which will enable the students to understand and develop testing practices for more complex systems.

We decided to write this book based on our teaching and industrial experiences in software testing and quality assurance. For the past 15 years, Sagar has been teaching software engineering and software testing on a regular basis, whereas Piyu has been performing hands-on testing and managing test groups for testing routers, switches, wireless data networks, storage networks, and intrusion prevention appliances. Our experiences have helped us in selecting and structuring the contents of this book to make it suitable as a textbook.

Who Should Read This Book?

We have written this book to introduce students and software professionals to the fundamental ideas in testing theory, testing techniques, testing practices, and quality assurance. Undergraduate students in software engineering, computer science, and computer engineering with no prior experience in the software industry will be introduced to the subject matter in a step-by-step manner. Practitioners too will benefit from the structured presentation and comprehensive nature of the materials. Graduate students can use the book as a reference resource. After reading the whole book, the reader will have a thorough understanding of the following topics:

- Fundamentals of testing theory and concepts
- Practices that support the production of quality software
- Software testing techniques
- Life-cycle models of requirements, defects, test cases, and test results
- Process models for unit, integration, system, and acceptance testing
- Building test teams, including recruiting and retaining test engineers
- Quality models, capability maturity model, testing maturity model, and test process improvement model

How Should This Book be Read?

The purpose of this book is to teach how to *do* software testing. We present some essential background material in Chapter 1 and save the enunciation of software

quality questions to a later part of the book. It is difficult to intelligently discuss for beginners what software quality *means* until one has a firm sense of what software testing *does*. However, practitioners with much testing experience can jump to Chapter 17, entitled "Software Quality," immediately after Chapter 1.

There are three different ways to read this book depending upon someone's interest. First, those who are exclusively interested in software testing concepts and want to apply the ideas should read Chapter 1 ("Basic Concepts and Preliminaries"), Chapter 3 ("Unit Testing"), Chapter 7 ("System Integration Testing"), and Chapters 8–14, related to system-level testing. Second, test managers interested in improving the test effectiveness of their teams can read Chapters 1, 3, 7, 8–14, 16 ("Test Team Organization"), 17 ("Software Quality"), and 18 ("Maturity Models"). Third, beginners should read the book from cover to cover.

Notes for Instructors

The book can be used as a text in an introductory course in software testing and quality assurance. One of the authors used the contents of this book in an undergraduate course entitled Software Testing and Quality Assurance for several years at the University of Waterloo. An introductory course in software testing can cover selected sections from most of the chapters except Chapter 16. For a course with more emphasis on testing techniques than on processes, we recommend to choose Chapters 1 ("Basic Concepts and Preliminaries") to 15 ("Software Reliability"). When used as a supplementary text in a software engineering course, selected portions from the following chapters can help students imbibe the essential concepts in software testing:

- Chapter 1: Basic Concepts and Preliminaries
- Chapter 3: Unit Testing
- Chapter 7: System Integration Testing
- Chapter 8: System Test Category
- Chapter 14: Acceptance Testing

Supplementary materials for instructors are available at the following Wiley website: http:/www.wiley.com/sagar.

Acknowledgments

In preparing this book, we received much support from many people, including the publisher, our family members, and our friends and colleagues. The support has been in many different forms. First, we would like to thank our editors, namely, Anastasia Wasko, Val Moliere, Whitney A. Lesch, Paul Petralia, and Danielle Lacourciere who gave us much professional guidance and patiently answered our various queries. Our friend Dr. Alok Patnaik read the whole draft and made numerous suggestions to improve the presentation quality of the book; we thank him for

all his effort and encouragement. The second author, Piyu Tripathy, would like to thank his former colleagues at Nortel Networks, Cisco Systems, and Airvana Inc., and present colleagues at NEC Laboratories America.

Finally, the support of our parents, parents-in-law, and partners deserve a special mention. I, Piyu Tripathy, would like to thank my dear wife Leena, who has taken many household and family duties off my hands to give me time that I needed to write this book. And I, Sagar Naik, would like to thank my loving wife Alaka for her invaluable support and for always being there for me. I would also like to thank my charming daughters, Monisha and Sameeksha, and exciting son, Siddharth, for their understanding while I am writing this book. I am grateful to my elder brother, Gajapati Naik, for all his support. We are very pleased that now we have more time for our families and friends.

Kshirasagar Naik
University of Waterloo
Waterloo

Priyadarshi Tripathy
NEC Laboratories America, Inc.
Princeton

LIST OF FIGURES

1.1 Shewhart cycle 2
1.2 Ishikawa diagram 4
1.3 Examples of basic test cases 11
1.4 Example of a test case with a sequence of < input, expected outcome > 12
1.5 Subset of the input domain exercising a subset of the program behavior 14
1.6 Different activities in program testing 14
1.7 Development and testing phases in the V model 16
1.8 Regression testing at different software testing levels. (From ref. 41. © 2005 John Wiley & Sons.) 17

2.1 Executing a program with a subset of the input domain 32
2.2 Example of inappropriate path selection 35
2.3 Different ways of comparing power of test methods: (*a*) produces all test cases produced by another method; (*b*) test sets have common elements. 43
2.4 Context of applying test adequacy 44

3.1 Steps in the code review process 55
3.2 Dynamic unit test environment 63
3.3 Test-first process in XP. (From ref. 24. © 2005 IEEE.) 72
3.4 Sample pseudocode for performing unit testing 73
3.5 The assertTrue() assertion throws an exception 75
3.6 Example test suite 76

4.1 Process of generating test input data for control flow testing 90
4.2 Symbols in a CFG 91
4.3 Function to open three files 91
4.4 High-level CFG representation of openfiles(). The three nodes are numbered 1, 2, and 3. 92
4.5 Detailed CFG representation of openfiles(). The numbers 1–21 are the nodes 93
4.6 Function to compute average of selected integers in an array. This program is an adaptation of "Figure 2. A sample program" in ref. 10. (With permission from the Australian Computer Society.) 94
4.7 A CFG representation of ReturnAverage(). Numbers 1–13 are the nodes. 95
4.8 Dashed arrows represent the branches not covered by statement covering in Table 4.4 99
4.9 Partial CFG with (*a*) OR operation and (*b*) AND operations 100
4.10 Example of a path from Figure 4.7 102
4.11 Path predicate for path in Figure 4.10 102
4.12 Method in Java to explain symbolic substitution [11] 103
4.13 Path predicate expression for path in Figure 4.10 105
4.14 Another example of path from Figure 4.7 105
4.15 Path predicate expression for path shown in Figure 4.14 106
4.16 Input data satisfying constraints of Figure 4.13 106

4.17 Binary search routine 111

5.1 Sequence of computations showing data flow anomaly 113
5.2 State transition diagram of a program variable. (From ref. 2. © 1979 IEEE.) 115
5.3 Definition and uses of variables 117
5.4 Data flow graph of ReturnAverage() example 118
5.5 Relationship among DF (data flow) testing criteria. (From ref. 4. © 1988 IEEE.) 125
5.6 Relationship among FDF (feasible data flow) testing criteria. (From ref. 4. © 1988 IEEE.) 127
5.7 Limitation of different fault detection techniques 128
5.8 Binary search routine 133
5.9 Modified binary search routine 133

6.1 Illustration of the concept of program domains 137
6.2 A function to explain program domains 139
6.3 Control flow graph representation of the function in Figure 6.2 139
6.4 Domains obtained from interpreted predicates in Figure 6.3 140
6.5 Predicates defining the TT domain in Figure 6.4 141
6.6 ON and OFF points 146
6.7 Boundary shift resulting in reduced domain (closed inequality) 147
6.8 Boundary shift resulting in enlarged domain (closed inequality) 149
6.9 Tilted boundary (closed inequality) 149
6.10 Closure error (closed inequality) 150
6.11 Boundary shift resulting in reduced domain (open inequality) 151
6.12 Boundary shift resulting in enlarged domain (open inequality) 152
6.13 Tilted boundary (open inequality) 153
6.14 Closure error (open inequality) 153
6.15 Equality border 154
6.16 Domains D_1, D_2 and D_3 157

7.1 Module hierarchy with three levels and seven modules 168
7.2 Top-down integration of modules A and B 169
7.3 Top-down integration of modules A, B, and D 169
7.4 Top-down integration of modules A, B, D, and C 169
7.5 Top-down integration of modules A, B, C, D, and E 170
7.6 Top-down integration of modules A, B, C, D, E, and F 170
7.7 Top-down integration of modules A, B, C, D, E, F and G 170
7.8 Bottom-up integration of modules E, F, and G 171
7.9 Bottom-up integration of modules B, C, and D with E, F, and G 172
7.10 Bottom-up integration of module A with all others 172
7.11 Hardware ECO process 179
7.12 Software ECO process 180
7.13 Module hierarchy of software system 190

8.1 Types of system tests 193
8.2 Types of basic tests 194
8.3 Types of functionality tests 197
8.4 Types of robustness tests 205
8.5 Typical 1xEV-DO radio access network. (Courtesy of Airvana, Inc.) 206

9.1 Frequency selection box of Bluetooth specification 224
9.2 Part of form ON479 of T1 general—2001, published by the CCRA 227

9.3 Functionally related variables 231
9.4 Function in context 232
9.5 (*a*) Obtaining output values from an input vector and (*b*) obtaining an input vector from an output value in functional testing 233
9.6 Functional testing in general 234
9.7 System S with three input variables 235
9.8 (a) Too many test inputs; (b) one input selected from each subdomain 244
9.9 Gold standard oracle 253
9.10 Parametric oracle 253
9.11 Statistical oracle 254

10.1 Spectrum of software systems 266
10.2 Data-dominated systems 266
10.3 Control-dominated systems 267
10.4 FSM model of dual-boot laptop computer 267
10.5 Interactions between system and its environment modeled as FSM 268
10.6 PCOs on a telephone 269
10.7 FSM model of a PBX 270
10.8 FSM model of PBX 271
10.9 Interaction of test sequence with SUT 274
10.10 Derived test case from transition tour 275
10.11 Conceptual model of test case with state verification 278
10.12 Finite-state machine G_1 (From ref. 5. © 1997 IEEE.) 281
10.13 UIO tree for G_1 in Figure 10.12. (From ref. 5. © 1997 IEEE.) 282
10.14 Identification of UIO sequences on UIO tree of Figure 10.13 283
10.15 Finite-state machine G_2 286
10.16 Distinguishing sequence tree for G_2 in Figure 10.15 286
10.17 FSM that does not possess distinguishing sequence. (From ref. 11. © 1994 IEEE.) 287
10.18 DS tree for FSM (Figure 10.17) 288
10.19 Abstraction of N-entity in OSI reference architecture 291
10.20 Abstract local test architecture 292
10.21 Abstract external test architecture 292
10.22 Local architecture 293
10.23 Distributed architecture 293
10.24 Coordinated architecture 294
10.25 Remote architecture 295
10.26 Structure of module in TTCN-3 297
10.27 Definitions of two subtypes 297
10.28 Parameterized template for constructing message to be sent 298
10.29 Parameterized template for constructing message to be received 298
10.30 Testing (a) square-root function (SRF) calculator and (*b*) port between tester and SRF calculator 299
10.31 Defining port type 300
10.32 Associating port with component 300
10.33 Test case for testing SRF calculator 301
10.34 Executing test case 302
10.35 Comparison of state transitions of FSM and EFSM 303
10.36 Controlled access to a door 304
10.37 SDL/GR door control system 305

10.38 Door control behavior specification 306
10.39 Door control behavior specification 307
10.40 Transition tour from door control system of Figures 10.38 and 10.39 309
10.41 Testing door control system 309
10.42 Output and input behavior obtained from transition tour of Figure 10.40 310
10.43 Test behavior obtained by refining if part in Figure 10.42 310
10.44 Test behavior that can receive unexpected events (derived from Figure 10.43) 311
10.45 Core behavior of test case for testing door control system (derived from Figure 10.44) 312
10.46 User interface of ATM 314
10.47 Binding of buttons with user options 314
10.48 Binding of buttons with cash amount 315
10.49 FSM G 318
10.50 FSM H 318
10.51 FSM K 319
10.52 Nondeterministic FSM 319

11.1 State transition diagram of requirement 323
11.2 Test suite structure 336
11.3 Service interworking between FR and ATM services 337
11.4 Transformation of FR to ATM cell 338
11.5 FrAtm test suite structure 342
11.6 State transition diagram of a test case 345
11.7 State transition diagram of test case result 349

12.1 Concept of cycle-based test execution strategy 363
12.2 Gantt chart for FR–ATM service interworking test project 390
12.3 Broad criteria of test automation tool evaluation 393
12.4 Test selection guideline for automation 396
12.5 Characteristics of automated test cases 397
12.6 Six major steps in automated test case 399
12.7 Components of a automation infrastructure 401

13.1 State transition diagram representation of life cycle of defect 409
13.2 Projected execution of test cases on weekly basis in cumulative chart form 417
13.3 PAE metric of Bazooka (PE: projected execution; AE: actually executed) project 421
13.4 Pareto diagram for defect distribution shown in Table 13.12 431
13.5 Cause–effect diagram for DCA 434

15.1 Relationship between MTTR, MTTF, and MTBF 475
15.2 Graphical representation of operational profile of library information system 484
15.3 Failure intensity λ as function of cumulative failure μ ($\lambda_0 = 9$ failures per unit time, $\nu_0 = 500$ failures, $\theta = 0.0075$) 488
15.4 Failure intensity λ as function of execution time τ ($\lambda_0 = 9$ failures per unit time, $\nu_0 = 500$ failures, $\theta = 0.0075$) 490
15.5 Cumulative failure μ as function of execution time τ ($\lambda_0 = 9$ failures per unit time, $\nu_0 = 500$ failures, $\theta = 0.0075$) 490

16.1 Structure of test groups 498
16.2 Structure of software quality assurance group 499
16.3 System test team hierarchy 500
16.4 Six phases of effective recruiting process 505

16.5 System test organization as part of development **518**

17.1 Relation between quality factors and quality criteria [6] **528**

17.2 ISO 9126 sample quality model refines standard's features into subcharacteristics. (From ref. 4. © 1996 IEEE.) **532**

18.1 CMM structure. (From ref. 3. © 2005 John Wiley & Sons.) **549**

18.2 SW-CMM maturity levels. (From ref. 3 © 2005 John Wiley & Sons.) **550**

18.3 Five-level structure of TMM. (From ref. 5. © 2003 Springer.) **568**

LIST OF TABLES

3.1 Hierarchy of System Documents 56
3.2 Code Review Checklist 58
3.3 McCabe Complexity Measure 79

4.1 Examples of Path in CFG of Figure 4.7 95
4.2 Input Domain of openfiles() 97
4.3 Inputs and Paths in openfiles() 97
4.4 Paths for Statement Coverage of CFG of Figure 4.7 98
4.5 Paths for Branch Coverage of CFG of Figure 4.7 99
4.6 Two Cases for Complete Statement and Branch Coverage of CFG of Figure 4.9*a* 101
4.7 Interpretation of Path Predicate of Path in Figure 4.10 104
4.8 Interpretation of Path Predicate of Path in Figure 4.14 105
4.9 Test Data for Statement and Branch Coverage 106

5.1 Def() and c-use() Sets of Nodes in Figure 5.4 120
5.2 Predicates and p-use() Set of Edges in Figure 5.4 121

6.1 Two Interpretations of Second if() Statement in Figure 6.2 140
6.2 Detection of Boundary Shift Resulting in Reduced Domain (Closed Inequality) 148
6.3 Detection of Boundary Shift Resulting in Enlarged Domain (Closed Inequality) 149
6.4 Detection of Boundary Tilt (Closed Inequality) 150
6.5 Detection of Closure Error (Closed Inequality) 151
6.6 Detection of Boundary Shift Resulting in Reduced Domain (Open Inequality) 151
6.7 Detection of Boundary Shift Resulting in Enlarged Domain (Open Inequality) 152
6.8 Detection of Boundary Tilt (Open Inequality) 153
6.9 Detection of Closure Error (Open Inequality) 154

7.1 Check-in Request Form 166
7.2 Example Software/Hardware Compatibility Matrix 178
7.3 Framework for SIT Plan 181
7.4 Framework for Entry Criteria to Start System Integration 182
7.5 Framework for System Integration Exit Criteria 182

8.1 EMS Functionalities 199
8.2 Regulatory Approval Bodies of Different Countries 217

9.1 Number of Special Values of Inputs to FBS Module of Figure 9.1 230
9.2 Input and Output Domains of Functions of P in Figure 9.6 234
9.3 Pairwise Test Cases for System S 236
9.4 $L_4(2^3)$ Orthogonal Array 236
9.5 Commonly Used Orthogonal Arrays 237
9.6 Various Values That Need to Be Tested in Combinations 238

9.7 $L_9(3^4)$ Orthogonal Array 239
9.8 $L_9(3^4)$ Orthogonal Array after Mapping Factors 239
9.9 Generated Test Cases after Mapping Left-Over Levels 240
9.10 Generated Test Cases to Cover Each Equivalence Class 246
9.11 Decision Table Comprising Set of Conditions and Effects 248
9.12 Pay Calculation Decision Table with Values for Each Rule 250
9.13 Pay Calculation Decision Table after Column Reduction 251
9.14 Decision Table for Payment Calculation 252

10.1 PCOs for Testing Telephone PBX 270
10.2 Set of States in FSM of Figure 10.8 272
10.3 Input and Output Sets in FSM of Figure 10.8 272
10.4 Transition Tours Covering All States in Figure 10.8 276
10.5 State Transitions Not Covered by Transition Tours of Table 10.4 277
10.6 Transition Tours Covering All State Transitions in Figure 10.8 277
10.7 UIO Sequences of Minimal Lengths Obtained from Figure 10.14 284
10.8 Examples of State Blocks 284
10.9 Outputs of FSM G_2 in Response to Input Sequence 11 in Different States 287
10.10 Output Sequences Generated by FSM of Figure 10.17 as Response to W_1 289
10.11 Output Sequences Generated by FSM of Figure 10.17 as Response to W_2 289
10.12 Test Sequences for State Transition $(D, A, a/x)$ of FSM in Figure 10.17 290

11.1 Coverage Matrix $[A_{ij}]$ 322
11.2 Requirement Schema Field Summary 324
11.3 Engineering Change Document Information 329
11.4 Characteristics of Testable Functional Specifications 333
11.5 Mapping of FR QoS Parameters to ATM QoS Parameters 340
11.6 Test Case Schema Summary 346
11.7 Test Suite Schema Summary 348
11.8 Test Result Schema Summary 348

12.1 Outline of System Test Plan 356
12.2 Equipment Needed to be Procured 360
12.3 Entry Criteria for First System Test Cycle 368
12.4 Test Case Failure Counts to Initiate RCA in Test Cycle 1 374
12.5 Test Case Failure Counts to Initiate RCA in Test Cycle 2 375
12.6 Test Effort Estimation for FR–ATM PVC Service Interworking 379
12.7 Form for Computing Unadjusted Function Point 382
12.8 Factors Affecting Development Effort 382
12.9 Empirical Relationship between Function Points and LOC 383
12.10 Guidelines for Manual Test Case Creation Effort 384
12.11 Guidelines for Manual Test Case Execution Effort 386
12.12 Guidelines for Estimation of Effort to Manually Execute Regression Test Cases 386
12.13 Benefits of Automated Testing 391

13.1 States of Defect Modeled in Figure 13.1 410
13.2 Defect Schema Summary Fields 412
13.3 State Transitions to Five Possible Next States from Open State 413
13.4 Outline of Test Execution Working Document 416
13.5 EST Metric in Week 4 of Bazooka Project 422
13.6 EST Metric in Bazooka Monitored on Weekly Basis 423

13.7 DAR Metric for Stinger Project 425
13.8 Weekly DRR Status for Stinger Test Project 426
13.9 Weekly OD on Priority Basis for Stinger Test Project 427
13.10 Weekly CD Observed by Different Groups for Stinger Test Project 427
13.11 ARD Metric for *Bayonet* 428
13.12 Sample Test Data of Chainsaw Test Project 430
13.13 Framework for Beta Release Criteria 436
13.14 Structure of Final System Test Report 438
13.15 Scale for Defect Age 443
13.16 Defect Injection versus Discovery on Project Boomerang 443
13.17 Number of Defects Weighted by Defect Age on Project Boomerang 444
13.18 ARD Metric for Test Project 448
13.19 Scale for PhAge 449

14.1 Outline of ATP 462
14.2 ACC Document Information 464
14.3 Structure of Acceptance Test Status Report 465
14.4 Structure of Acceptance Test Summary Report 466

15.1 Example of Operational Profile of Library Information System 484

17.1 McCall's Quality Factors 524
17.2 Categorization of McCall's Quality Factors 527
17.3 McCall's Quality Criteria 529

18.1 Requirements for Different Maturity Levels 564
18.2 Test Maturity Matrix 566

CHAPTER 1

Basic Concepts and Preliminaries

Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the second law of thermodynamics, i.e., it always increases.
— *Norman Ralph Augustine*

1.1 QUALITY REVOLUTION

People seek quality in every man-made artifact. Certainly, the concept of quality did not originate with software systems. Rather, the quality concept is likely to be as old as human endeavor to mass produce artifacts and objects of large size. In the past couple of decades a quality revolution, has been spreading fast throughout the world with the explosion of the Internet. Global competition, outsourcing, off-shoring, and increasing customer expectations have brought the concept of quality to the forefront. Developing quality products on tighter schedules is critical for a company to be successful in the new global economy. Traditionally, efforts to improve quality have centered around the end of the product development cycle by emphasizing the detection and correction of defects. On the contrary, the new approach to enhancing quality encompasses all phases of a product development process—from a requirements analysis to the final delivery of the product to the customer. Every step in the development process must be performed to the highest possible standard. An effective quality process must focus on [1]:

- Paying much attention to customer's requirements
- Making efforts to continuously improve quality
- Integrating measurement processes with product design and development
- Pushing the quality concept down to the lowest level of the organization
- Developing a system-level perspective with an emphasis on methodology and process
- Eliminating waste through continuous improvement

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

A quality movement started in Japan during the 1940s and the 1950s by William Edwards Deming, Joseph M. Juran, and Kaoru Ishikawa. In circa 1947, W. Edwards Deming "visited India as well, then continued on to Japan, where he had been asked to join a statistical mission responsible for planning the 1951 Japanese census" [2], p. 8. During his said visit to Japan, Deming invited statisticians for a dinner meeting and told them how important they were and what they could do for Japan [3]. In March 1950, he returned to Japan at the invitation of Managing Director Kenichi Koyanagi of the Union of Japanese Scientists and Engineers (JUSE) to teach a course to Japanese researchers, workers, executives, and engineers on statistical quality control (SQC) methods. Statistical quality control is a discipline based on measurements and statistics. Decisions are made and plans are developed based on the collection and evaluation of actual data in the form of metrics, rather than intuition and experience. The SQC methods use seven basic quality management tools: Pareto analysis, cause-and-effect diagram, flow chart, trend chart, histogram, scatter diagram, and control chart [2].

In July 1950, Deming gave an eight-day seminar based on the Shewhart methods of statistical quality control [4, 5] for Japanese engineers and executives. He introduced the *plan–do–check–act* (PDCA) cycle in the seminar, which he called the Shewhart cycle (Figure 1.1). The Shewhart cycle illustrates the following activity sequence: setting goals, assigning them to measurable milestones, and assessing the progress against those milestones. Deming's 1950 lecture notes formed the basis for a series of seminars on SQC methods sponsored by the JUSE and provided the criteria for Japan's famed Deming Prize. Deming's work has stimulated several different kinds of industries, such as those for radios, transistors, cameras, binoculars, sewing machines, and automobiles.

Between circa 1950 and circa 1970, automobile industries in Japan, in particular Toyota Motor Corporation, came up with an innovative principle to compress the time period from customer order to banking payment, known as the "lean principle." The objective was to minimize the consumption of resources that added no value to a product. The lean principle has been defined by the National Institute of Standards and Technology (NIST) Manufacturing Extension Partnership program [61] as "a systematic approach to identifying and eliminating waste through continuous improvement, flowing the product at the pull of the customer in pursuit of perfection," p.1. It is commonly believed that lean principles were started in Japan by Taiichi Ohno of Toyota [7], but Henry Ford

Figure 1.1 Shewhart cycle.

had been using parts of lean as early as circa 1920, as evidenced by the following quote (Henry Ford, 1926) [61], p.1:

> One of the noteworthy accomplishments in keeping the price of Ford products low is the gradual shortening of the production cycle. The longer an article is in the process of manufacture and the more it is moved about, the greater is its ultimate cost.

This concept was popularized in the United States by a Massachusetts Institute of Technology (MIT) study of the movement from mass production toward production, as described in *The Machine That Changed the World*, by James P. Womack, Daniel T. Jones, and Daniel Roos, New York: Rawson and Associates, 1990. Lean thinking continues to spread to every country in the world, and leaders are adapting the principles beyond automobile manufacturing, to logistics and distribution, services, retail, health care, construction, maintenance, and software development [8].

Remark: Walter Andrew Shewhart was an American physicist, engineer, and statistician and is known as the father of statistical quality control. Shewhart worked at Bell Telephone Laboratories from its foundation in 1925 until his retirement in 1956 [9]. His work was summarized in his book *Economic Control of Quality of Manufactured Product*, published by McGraw-Hill in 1931. In 1938, his work came to the attention of physicist W. Edwards Deming, who developed some of Shewhart's methodological proposals in Japan from 1950 onward and named his synthesis the Shewhart cycle.

In 1954, Joseph M. Juran of the United States proposed raising the level of quality management from the manufacturing units to the entire organization. He stressed the importance of systems thinking that begins with product requirement, design, prototype testing, proper equipment operations, and accurate process feedback. Juran's seminar also became a part of the JUSE's educational programs [10]. Juran spurred the move from SQC to TQC (total quality control) in Japan. This included companywide activities and education in quality control (QC), audits, quality circle, and promotion of quality management principles. The term TQC was coined by an American, Armand V. Feigenbaum, in his 1951 book *Quality Control Principles, Practice and Administration*. It was republished in 2004 [11]. By 1968, Kaoru Ishikawa, one of the fathers of TQC in Japan, had outlined, as shown in the following, the key elements of TQC management [12]:

- Quality comes first, not short-term profits.
- The customer comes first, not the producer.
- Decisions are based on facts and data.
- Management is participatory and respectful of all employees.
- Management is driven by cross-functional committees covering product planning, product design, purchasing, manufacturing, sales, marketing, and distribution.

Remark: A quality circle is a volunteer group of workers, usually members of the same department, who meet regularly to discuss the problems and make presentations to management with their ideas to overcome them. Quality circles were started in Japan in 1962 by Kaoru Ishikawa as another method of improving quality. The movement in Japan was coordinated by the JUSE.

One of the innovative TQC methodologies developed in Japan is referred to as the *Ishikawa* or *cause-and-effect* diagram. Kaoru Ishikawa found from statistical data that dispersion in product quality came from four common causes, namely *materials*, *machines*, *methods*, and *measurements*, known as the 4 Ms (Figure 1.2). The bold horizontal arrow points to quality, whereas the diagonal arrows in Figure 1.2 are probable causes having an effect on the quality. Materials often differ when sources of supply or size requirements vary. Machines, or equipment, also function differently depending on variations in their parts, and they operate optimally for only part of the time. Methods, or processes, cause even greater variations due to lack of training and poor handwritten instructions. Finally, measurements also vary due to outdated equipment and improper calibration. Variations in the 4 Ms parameters have an effect on the quality of a product. The Ishikawa diagram has influenced Japanese firms to focus their quality control attention on the improvement of materials, machines, methods, and measurements.

The total-quality movement in Japan has led to pervasive top-management involvement. Many companies in Japan have extensive documentation of their quality activities. Senior executives in the United States either did not believe quality mattered or did not know where to begin until the National Broadcasting Corporation (NBC), an America television network, broadcast the documentary "If Japan Can ... Why Can't We?" at 9:30 P.M. on June 24, 1980 [2]. The documentary was produced by Clare Crawford-Mason and was narrated by Lloyd Dobyns. Fifteen minutes of the broadcast was devoted to Dr. Deming and his work. After the

Figure 1.2 Ishikawa diagram.

broadcast, many executives and government leaders realized that a renewed emphasis on quality was no longer an option for American companies but a necessity for doing business in an ever-expanding and more demanding competitive world market. Ford Motor Company and General Motors immediately adopted Deming's SQC methodology into their manufacturing process. Other companies such as Dow Chemical and the Hughes Aircraft followed suit. Ishikawa's TQC management philosophy gained popularity in the United States. Further, the spurred emphasis on quality in American manufacturing companies led the U.S. Congress to establish the Malcolm Baldrige National Quality Award—similar to the Deming Prize in Japan—in 1987 to recognize organizations for their achievements in quality and to raise awareness about the importance of quality excellence as a competitive edge [6]. In the Baldrige National Award, quality is viewed as something defined by the customer and thus the focus is on *customer-driven quality*. On the other hand, in the Deming Prize, quality is viewed as something defined by the producers by conforming to specifications and thus the focus is on *conformance to specifications*.

Remark: Malcolm Baldrige was U.S. Secretary of Commerce from 1981 until his death in a rodeo accident in July 1987. Baldrige was a proponent of quality management as a key to his country's prosperity and long-term strength. He took a personal interest in the quality improvement act, which was eventually named after him, and helped draft one of its early versions. In recognition of his contributions, Congress named the award in his honor.

Traditionally, the TQC and lean concepts are applied in the manufacturing process. The software development process uses these concepts as another tool to guide the production of quality software [13]. These concepts provides a framework to discuss software production issues. The software capability maturity model (CMM) [14] architecture developed at the Software Engineering Institute is based on the principles of product quality that have been developed by W. Edwards Deming [15], Joseph M. Juran [16], Kaoru Ishikawa [12], and Philip Crosby [17].

1.2 SOFTWARE QUALITY

The question "What is software quality?" evokes many different answers. Quality is a complex concept—it means different things to different people, and it is highly context dependent. Garvin [18] has analyzed how software quality is perceived in different ways in different domains, such as philosophy, economics, marketing, and management. Kitchenham and Pfleeger's article [60] on software quality gives a succinct exposition of software quality. They discuss five views of quality in a comprehensive manner as follows:

1. *Transcendental View*: It envisages quality as something that can be recognized but is difficult to define. The transcendental view is not specific to software quality alone but has been applied in other complex areas

of everyday life. For example, In 1964, Justice Potter Stewart of the U.S. Supreme Court, while ruling on the case *Jacobellis v. Ohio*, 378 U.S. 184 (1964), which involved the state of Ohio banning the French film *Les Amants* ("The Lovers") on the ground of pornography, wrote "I shall not today attempt further to define the kinds of material I understand to be embraced within that shorthand description; and perhaps I could never succeed in intelligibly doing so. But *I know it when I see it*, and the motion picture involved in this case is not that" (emphasis added).

2. *User View*: It perceives quality as fitness for purpose. According to this view, while evaluating the quality of a product, one must ask the key question: "Does the product satisfy user needs and expectations?"
3. *Manufacturing View*: Here quality is understood as conformance to the specification. The quality level of a product is determined by the extent to which the product meets its specifications.
4. *Product View*: In this case, quality is viewed as tied to the inherent characteristics of the product. A product's inherent characteristics, that is, internal qualities, determine its external qualities.
5. *Value-Based View*: Quality, in this perspective, depends on the amount a customer is willing to pay for it.

The concept of software quality and the efforts to understand it in terms of measurable quantities date back to the mid-1970s. McCall, Richards, and Walters [19] were the first to study the concept of software quality in terms of *quality factors* and *quality criteria*. A quality factor represents a behavioral characteristic of a system. Some examples of high-level quality factors are *correctness*, *reliability*, *efficiency*, *testability*, *maintainability*, and *reusability*. A quality criterion is an attribute of a quality factor that is related to software development. For example, modularity is an attribute of the architecture of a software system. A highly modular software allows designers to put cohesive components in one module, thereby improving the maintainability of the system.

Various software quality models have been proposed to define quality and its related attributes. The most influential ones are the ISO 9126 [20–22] and the CMM [14]. The ISO 9126 quality model was developed by an expert group under the aegis of the International Organization for Standardization (ISO). The document ISO 9126 defines six broad, independent categories of quality characteristics: *functionality, reliability, usability, efficiency, maintainability*, and *portability*. The CMM was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University. In the CMM framework, a development process is evaluated on a scale of 1–5, commonly known as level 1 through level 5. For example, level 1 is called the initial level, whereas level 5—optimized—is the highest level of process maturity.

In the field of software testing, there are two well-known process models, namely, the test process improvement (TPI) model [23] and the test maturity Model (TMM) [24]. These two models allow an organization to assess the current state

of their software testing processes, identify the next logical area for improvement, and recommend an action plan for test process improvement.

1.3 ROLE OF TESTING

Testing plays an important role in achieving and assessing the quality of a software product [25]. On the one hand, we improve the quality of the products as we repeat a *test–find defects–fix* cycle during development. On the other hand, we assess how good our system is when we perform system-level tests before releasing a product. Thus, as Friedman and Voas [26] have succinctly described, software testing is a verification process for software quality assessment and improvement. Generally speaking, the activities for software quality assessment can be divided into two broad categories, namely, *static analysis* and *dynamic analysis*.

- **Static Analysis:** As the term "static" suggests, it is based on the examination of a number of documents, namely requirements documents, software models, design documents, and source code. Traditional static analysis includes code review, inspection, walk-through, algorithm analysis, and proof of correctness. It does not involve actual execution of the code under development. Instead, it examines code and reasons over all possible behaviors that might arise during run time. Compiler optimizations are standard static analysis.
- **Dynamic Analysis:** Dynamic analysis of a software system involves actual program execution in order to expose possible program failures. The behavioral and performance properties of the program are also observed. Programs are executed with both typical and carefully chosen input values. Often, the input set of a program can be impractically large. However, for practical considerations, a finite subset of the input set can be selected. Therefore, in testing, we observe some representative program behaviors and reach a conclusion about the quality of the system. Careful selection of a finite test set is crucial to reaching a reliable conclusion.

By performing static and dynamic analyses, practitioners want to identify as many faults as possible so that those faults are fixed at an early stage of the software development. Static analysis and dynamic analysis are complementary in nature, and for better effectiveness, both must be performed repeatedly and alternated. Practitioners and researchers need to remove the boundaries between static and dynamic analysis and create a hybrid analysis that combines the strengths of both approaches [27].

1.4 VERIFICATION AND VALIDATION

Two similar concepts related to software testing frequently used by practitioners are *verification* and *validation*. Both concepts are abstract in nature, and each can be

realized by a set of concrete, executable activities. The two concepts are explained as follows:

- **Verification:** This kind of activity helps us in evaluating a software system by determining whether the product of a given development phase satisfies the requirements established before the start of that phase. One may note that a product can be an intermediate product, such as requirement specification, design specification, code, user manual, or even the final product. Activities that check the correctness of a development phase are called *verification activities*.
- **Validation:** Activities of this kind help us in confirming that a product meets its intended *use*. Validation activities aim at confirming that a product meets its customer's expectations. In other words, validation activities focus on the final product, which is extensively tested from the customer point of view. Validation establishes whether the product meets overall expectations of the users.

  Late execution of validation activities is often risky by leading to higher development cost. Validation activities may be executed at early stages of the software development cycle [28]. An example of early execution of validation activities can be found in the eXtreme Programming (XP) software development methodology. In the XP methodology, the customer closely interacts with the software development group and conducts acceptance tests during each development iteration [29].

The verification process establishes the correspondence of an implementation phase of the software development process with its specification, whereas validation establishes the correspondence between a system and users' expectations. One can compare verification and validation as follows:

- Verification activities aim at confirming that one is *building the product correctly*, whereas validation activities aim at confirming that one is *building the correct product* [30].
- Verification activities review interim work products, such as requirements specification, design, code, and user manual, during a project life cycle to ensure their quality. The quality attributes sought by verification activities are consistency, completeness, and correctness at each major stage of system development. On the other hand, validation is performed toward the end of system development to determine if the entire system meets the customer's needs and expectations.
- Verification activities are performed on interim products by applying mostly static analysis techniques, such as inspection, walkthrough, and reviews, and using standards and checklists. Verification can also include dynamic analysis, such as actual program execution. On the other hand, validation is performed on the entire system by actually running the system in its real environment and using a variety of tests.

1.5 FAILURE, ERROR, FAULT, AND DEFECT

In the literature on software testing, one can find references to the terms *failure*, *error*, *fault*, and *defect*. Although their meanings are related, there are important distinctions between these four concepts. In the following, we present first three terms as they are understood in the fault-tolerant computing community:

- **Failure:** A failure is said to occur whenever the external behavior of a system does not conform to that prescribed in the system specification.
- **Error:** An error is a *state* of the system. In the absence of any corrective action by the system, an error state could lead to a failure which would not be attributed to any event subsequent to the error.
- **Fault:** A fault is the adjudged cause of an error.

A fault may remain undetected for a long time, until some event activates it. When an event activates a fault, it first brings the program into an intermediate error state. If computation is allowed to proceed from an error state without any corrective action, the program eventually causes a failure. As an aside, in fault-tolerant computing, corrective actions can be taken to take a program out of an error state into a desirable state such that subsequent computation does not eventually lead to a failure. The process of failure manifestation can therefore be succinctly represented as a behavior chain [31] as follows: fault $\rightarrow$ error $\rightarrow$ failure. The behavior chain can iterate for a while, that is, failure of one component can lead to a failure of another interacting component.

The above definition of failure assumes that the given specification is acceptable to the customer. However, if the specification does not meet the expectations of the customer, then, of course, even a fault-free implementation fails to satisfy the customer. It is a difficult task to give a precise definition of fault, error, or failure of software, because of the "human factor" involved in the overall acceptance of a system. In an article titled "What Is Software Failure" [32], Ram Chillarege commented that in modern software business software failure means "the customer's expectation has not been met and/or the customer is unable to do useful work with product," p. 354.

Roderick Rees [33] extended Chillarege's comments of software failure by pointing out that "failure is a matter of function only [and is thus] related to purpose, not to whether an item is physically intact or not" (p. 163). To substantiate this, Behrooz Parhami [34] provided three interesting examples to show the relevance of such a view point in wider context. One of the examples is quoted here (p. 451):

> Consider a small organization. *Defects* in the organization's staff promotion policies can cause improper promotions, viewed as *faults*. The resulting ineptitudes & dissatisfactions are *errors* in the organization's state. The organization's personnel or departments probably begin to *malfunction* as result of the errors, in turn causing an overall *degradation* of performance. The end result can be the organization's *failure* to achieve its goal.

There is a fine difference between defects and faults in the above example, that is, execution of a defective policy may lead to a faulty promotion. In a software

context, a software system may be defective due to design issues; certain system states will expose a defect, resulting in the development of faults defined as incorrect signal values or decisions within the system. In industry, the term defect is widely used, whereas among researchers the term fault is more prevalent. For all practical purpose, the two terms are synonymous. In this book, we use the two terms interchangeably as required.

1.6 NOTION OF SOFTWARE RELIABILITY

No matter how many times we run the test–find faults–fix cycle during software development, some faults are likely to escape our attention, and these will eventually surface at the customer site. Therefore, a quantitative measure that is useful in assessing the quality of a software is its *reliability* [35]. *Software reliability* is defined as the probability of failure-free operation of a software system for a specified time in a specified environment. The level of reliability of a system depends on those inputs that cause failures to be observed by the end users. Software reliability can be estimated via *random testing*, as suggested by Hamlet [36]. Since the notion of reliability is specific to a "specified environment," test data must be drawn from the input distribution to closely resemble the future usage of the system. Capturing the future usage pattern of a system in a general sense is described in a form called the *operational profile*. The concept of operational profile of a system was pioneered by John D. Musa at AT&T Bell Laboratories between the 1970s and the 1990s [37, 38].

1.7 OBJECTIVES OF TESTING

The stakeholders in a test process are the programmers, the test engineers, the project managers, and the customers. A stakeholder is a person or an organization who influences a system's behaviors or who is impacted by that system [39]. Different stakeholders view a test process from different perspectives as explained below:

- **It does work:** While implementing a program unit, the programmer may want to test whether or not the unit works in normal circumstances. The programmer gets much confidence if the unit works to his or her satisfaction. The same idea applies to an entire system as well—once a system has been integrated, the developers may want to test whether or not the system performs the basic functions. Here, for the psychological reason, the objective of testing is to show that the system works, rather than it does not work.
- **It does not work:** Once the programmer (or the development team) is satisfied that a unit (or the system) works to a certain degree, more tests are conducted with the objective of finding faults in the unit (or the system). Here, the idea is to try to make the unit (or the system) fail.

- **Reduce the risk of failure:** Most of the complex software systems contain faults, which cause the system to fail from time to time. This concept of "failing from time to time" gives rise to the notion of *failure rate*. As faults are discovered and fixed while performing more and more tests, the failure rate of a system generally decreases. Thus, a higher level objective of performing tests is to bring down the risk of failing to an acceptable level.
- **Reduce the cost of testing:** The different kinds of costs associated with a test process include

  the cost of designing, maintaining, and executing test cases,

  the cost of analyzing the result of executing each test case,

  the cost of documenting the test cases, and

  the cost of actually executing the system and documenting it.

  Therefore, the less the number of test cases designed, the less will be the associated cost of testing. However, producing a small number of arbitrary test cases is not a good way of saving cost. The highest level of objective of performing tests is to produce low-risk software with fewer number of test cases. This idea leads us to the concept of *effectiveness of test cases*. Test engineers must therefore judiciously select fewer, effective test cases.

1.8 WHAT IS A TEST CASE?

In its most basic form, a *test case* is a simple pair of $<$ input, expected outcome $>$. If a program under test is expected to compute the square root of nonnegative numbers, then four examples of test cases are as shown in Figure 1.3.

In stateless systems, where the outcome depends solely on the current input, test cases are very simple in structure, as shown in Figure 1.3. A program to compute the square root of nonnegative numbers is an example of a stateless system. A compiler for the C programming language is another example of a stateless system. A compiler is a stateless system because to compile a program it does not need to know about the programs it compiled previously.

In state-oriented systems, where the program outcome depends both on the current state of the system and the current input, a test case may consist of a

TB_1: $< 0, 0 >$,
TB_2: $< 25, 5 >$,
TB_3: $< 40, 6.3245553 >$,
TB_4: $< 100.5, 10.024968 >$.

Figure 1.3 Examples of basic test cases.

TS_1:	< check balance, \$500.00 >, < withdraw, ''amount?'' >, < \$200.00, ''\$200.00'' >, < check balance, \$300.00 > .

Figure 1.4 Example of a test case with a sequence of < input, expected outcome >.

sequence of < input, expected outcome > pairs. A telephone switching system and an automated teller machine (ATM) are examples of state-oriented systems. For an ATM machine, a test case for testing the *withdraw* function is shown in Figure 1.4. Here, we assume that the user has already entered validated inputs, such as the cash card and the personal identification number (PIN).

In the test case TS_1, "check balance" and "withdraw" in the first, second, and fourth tuples represent the pressing of the appropriate keys on the ATM keypad. It is assumed that the user account has \$500.00 on it, and the user wants to withdraw an amount of \$200.00. The expected outcome "\$200.00" in the third tuple represents the cash dispensed by the ATM. After the withdrawal operation, the user makes sure that the remaining balance is \$300.00.

For state-oriented systems, most of the test cases include some form of decision and timing in providing input to the system. A test case may include loops and timers, which we do not show at this moment.

1.9 EXPECTED OUTCOME

An *outcome* of program execution is a complex entity that may include the following:

- Values produced by the program:
  - Outputs for local observation (integer, text, audio, image)
  - Outputs (messages) for remote storage, manipulation, or observation
- State change:
  - State change of the program
  - State change of the database (due to add, delete, and update operations)
- A sequence or set of values which must be interpreted together for the outcome to be valid

An important concept in test design is the concept of an *oracle*. An oracle is any entity—program, process, human expert, or body of data—that tells us the expected outcome of a particular test or set of tests [40]. A test case is meaningful only if it is possible to decide on the acceptability of the result produced by the program under test.

Ideally, the expected outcome of a test should be computed while designing the test case. In other words, the test outcome is computed before the program is

executed with the selected test input. The idea here is that one should be able to compute the expected outcome from an *understanding* of the program's requirements. Precomputation of the expected outcome will eliminate any implementation bias in case the test case is designed by the developer.

In exceptional cases, where it is extremely difficult, impossible, or even undesirable to compute a single expected outcome, one should identify expected outcomes by examining the actual test outcomes, as explained in the following:

1. Execute the program with the selected input.
2. Observe the actual outcome of program execution.
3. Verify that the actual outcome is the expected outcome.
4. Use the verified actual outcome as the expected outcome in subsequent runs of the test case.

1.10 CONCEPT OF *COMPLETE* TESTING

It is not unusual to find people making claims such as "I have exhaustively tested the program." Complete, or exhaustive, testing means *there are no undiscovered faults at the end of the test phase*. All problems must be known at the end of complete testing. For most of the systems, complete testing is near impossible because of the following reasons:

- The domain of possible inputs of a program is too large to be completely used in testing a system. There are both valid inputs and invalid inputs. The program may have a large number of states. There may be timing constraints on the inputs, that is, an input may be valid at a certain time and invalid at other times. An input value which is valid but is not properly timed is called an *inopportune* input. The input domain of a system can be very large to be completely used in testing a program.
- The design issues may be too complex to completely test. The design may have included implicit design decisions and assumptions. For example, a programmer may use a global variable or a *static* variable to control program execution.
- It may not be possible to create all possible execution environments of the system. This becomes more significant when the behavior of the software system depends on the real, outside world, such as weather, temperature, altitude, pressure, and so on.

1.11 CENTRAL ISSUE IN TESTING

We must realize that though the outcome of complete testing, that is, discovering all faults, is highly desirable, it is a near-impossible task, and it may not be attempted. The next best thing is to select a subset of the input domain to test a program.

Figure 1.5 Subset of the input domain exercising a subset of the program behavior.

Referring to Figure 1.5, let D be the input domain of a program P. Suppose that we select a subset D_1 of D, that is, $D_1 \subset D$, to test program P. It is possible that D_1 exercises only a part P_1, that is, $P_1 \subset P$, of the *execution behavior* of P, in which case faults with the other part, P_2, will go undetected.

By selecting a subset of the input domain D_1, the test engineer attempts to deduce properties of an entire program P by observing the behavior of a part P_1 of the entire behavior of P on selected inputs D_1. Therefore, *selection* of the subset of the input domain must be done in a systematic and careful manner so that the deduction is as accurate and complete as possible. For example, the idea of *coverage* is considered while selecting test cases.

1.12 TESTING ACTIVITIES

In order to test a program, a test engineer must perform a sequence of testing activities. Most of these activities have been shown in Figure 1.6 and are explained in the following. These explanations focus on a single test case.

- **Identify an objective to be tested:** The first activity is to identify an *objective* to be tested. The objective defines the intention, or *purpose*, of designing one or more test cases to ensure that the program supports the objective. A clear purpose must be associated with every test case.

Figure 1.6 Different activities in program testing.

- **Select inputs:** The second activity is to select test inputs. Selection of test inputs can be based on the requirements specification, the source code, or our expectations. Test inputs are selected by keeping the test objective in mind.
- **Compute the expected outcome:** The third activity is to compute the expected outcome of the program with the selected inputs. In most cases, this can be done from an overall, high-level understanding of the test objective and the specification of the program under test.
- **Set up the execution environment of the program:** The fourth step is to prepare the right execution environment of the program. In this step all the assumptions external to the program must be satisfied. A few examples of assumptions external to a program are as follows:

  Initialize the local system, external to the program. This may include making a network connection available, making the right database system available, and so on.

  Initialize any remote, external system (e.g., remote partner process in a distributed application.) For example, to test the client code, we may need to start the server at a remote site.

- **Execute the program:** In the fifth step, the test engineer executes the program with the selected inputs and observes the actual outcome of the program. To execute a test case, inputs may be provided to the program at different physical locations at different times. The concept of *test coordination* is used in synchronizing different components of a test case.
- **Analyze the test result:** The final test activity is to analyze the result of test execution. Here, the main task is to compare the actual outcome of program execution with the expected outcome. The complexity of comparison depends on the complexity of the data to be observed. The observed data type can be as simple as an integer or a string of characters or as complex as an image, a video, or an audio clip. At the end of the analysis step, a test verdict is assigned to the program. There are three major kinds of test verdicts, namely, *pass*, *fail*, and *inconclusive*, as explained below.

  If the program produces the expected outcome and the purpose of the test case is satisfied, then a pass verdict is assigned.

  If the program does not produce the expected outcome, then a fail verdict is assigned.

  However, in some cases it may not be possible to assign a clear pass or fail verdict. For example, if a timeout occurs while executing a test case on a distributed application, we may not be in a position to assign a clear pass or fail verdict. In those cases, an inconclusive test verdict is assigned. An inconclusive test verdict means that further tests are needed to be done to refine the inconclusive verdict into a clear pass or fail verdict.

A *test report* must be written after analyzing the test result. The motivation for writing a test report is to get the fault fixed if the test revealed a fault. A test report contains the following items to be informative:

Explain how to reproduce the failure.

Analyze the failure to be able to describe it.

A pointer to the actual outcome and the test case, complete with the input, the expected outcome, and the execution environment.

1.13 TEST LEVELS

Testing is performed at different levels involving the complete system or parts of it throughout the life cycle of a software product. A software system goes through four stages of testing before it is actually deployed. These four stages are known as *unit*, *integration*, *system*, and *acceptance* level testing. The first three levels of testing are performed by a number of different stakeholders in the development organization, where as acceptance testing is performed by the customers. The four stages of testing have been illustrated in the form of what is called the classical V model in Figure 1.7.

In unit testing, programmers test individual program units, such as a procedures, functions, methods, or classes, in isolation. After ensuring that individual units work to a satisfactory extent, modules are assembled to construct larger subsystems by following integration testing techniques. Integration testing is jointly performed by software developers and integration test engineers. The objective of

Figure 1.7 Development and testing phases in the V model.

integration testing is to construct a reasonably stable system that can withstand the rigor of system-level testing. System-level testing includes a wide spectrum of testing, such as functionality testing, security testing, robustness testing, load testing, stability testing, stress testing, performance testing, and reliability testing. System testing is a critical phase in a software development process because of the need to meet a tight schedule close to delivery date, to discover most of the faults, and to verify that fixes are working and have not resulted in new faults. System testing comprises a number of distinct activities: creating a test plan, designing a test suite, preparing test environments, executing the tests by following a clear strategy, and monitoring the process of test execution.

Regression testing is another level of testing that is performed throughout the life cycle of a system. Regression testing is performed whenever a component of the system is modified. The key idea in regression testing is to ascertain that the modification has not introduced any new faults in the portion that was not subject to modification. To be precise, regression testing is not a distinct level of testing. Rather, it is considered as a subphase of unit, integration, and system-level testing, as illustrated in Figure 1.8 [41].

In regression testing, new tests are not designed. Instead, tests are selected, prioritized, and executed from the existing pool of test cases to ensure that nothing is broken in the new version of the software. Regression testing is an expensive process and accounts for a predominant portion of testing effort in the industry. It is desirable to select a subset of the test cases from the existing pool to reduce the cost. A key question is how many and which test cases should be selected so that the selected test cases are more likely to uncover new faults [42–44].

After the completion of system-level testing, the product is delivered to the customer. The customer performs their own series of tests, commonly known as *acceptance testing*. The objective of acceptance testing is to measure the quality of the product, rather than searching for the defects, which is objective of system testing. A key notion in acceptance testing is the customer's *expectations* from the system. By the time of acceptance testing, the customer should have developed their acceptance criteria based on their own expectations from the system. There are two kinds of acceptance testing as explained in the following:

- User acceptance testing (UAT)
- Business acceptance testing (BAT)

Figure 1.8 Regression testing at different software testing levels. (From ref. 41. © 2005 John Wiley & Sons.)

User acceptance testing is conducted by the customer to ensure that the system satisfies the contractual acceptance criteria before being signed off as meeting user needs. On the other hand, BAT is undertaken within the supplier's development organization. The idea in having a BAT is to ensure that the system will eventually pass the user acceptance test. It is a rehearsal of UAT at the supplier's premises.

1.14 SOURCES OF INFORMATION FOR TEST CASE SELECTION

Designing test cases has continued to stay in the foci of the research community and the practitioners. A software development process generates a large body of information, such as requirements specification, design document, and source code. In order to generate effective tests at a lower cost, test designers analyze the following sources of information:

- Requirements and functional specifications
- Source code
- Input and output domains
- Operational profile
- Fault model

Requirements and Functional Specifications The process of software development begins by capturing user needs. The nature and amount of user needs identified at the beginning of system development will vary depending on the specific life-cycle model to be followed. Let us consider a few examples. In the Waterfall model [45] of software development, a requirements engineer tries to capture most of the requirements. On the other hand, in an agile software development model, such as XP [29] or the Scrum [46–48], only a few requirements are identified in the beginning. A test engineer considers all the requirements the program is expected to meet whichever life-cycle model is chosen to test a program.

The requirements might have been specified in an informal manner, such as a combination of plaintext, equations, figures, and flowcharts. Though this form of requirements specification may be ambiguous, it is easily understood by customers. For example, the Bluetooth specification consists of about 1100 pages of descriptions explaining how various subsystems of a Bluetooth interface is expected to work. The specification is written in plaintext form supplemented with mathematical equations, state diagrams, tables, and figures. For some systems, requirements may have been captured in the form of *use cases*, *entity–relationship diagrams*, and *class diagrams*. Sometimes the requirements of a system may have been specified in a formal language or notation, such as Z, SDL, Estelle, or finite-state machine. Both the informal and formal specifications are prime sources of test cases [49].

Source Code Whereas a requirements specification describes the *intended behavior* of a system, the source code describes the *actual behavior* of the system. High-level assumptions and constraints take concrete form in an implementation. Though a software designer may produce a detailed design, programmers may introduce additional details into the system. For example, a step in the detailed design can be "sort array A." To sort an array, there are many sorting algorithms with different characteristics, such as iteration, recursion, and temporarily using another array. Therefore, test cases must be designed based on the program [50].

Input and Output Domains Some values in the input domain of a program have special meanings, and hence must be treated separately [5]. To illustrate this point, let us consider the *factorial* function. The factorial of a nonnegative integer n is computed as follows:

```
factorial(0) = 1;
factorial(1) = 1;
factorial(n) = n * factorial(n-1);
```

A programmer may wrongly implement the factorial function as

```
factorial(n) = 1 * 2 * ... * n;
```

without considering the special case of $n = 0$. The above wrong implementation will produce the correct result for all positive values of n, but will fail for $n = 0$.

Sometimes even some output values have special meanings, and a program must be tested to ensure that it produces the special values for all possible causes. In the above example, the output value 1 has special significance: (i) it is the minimum value computed by the factorial function and (ii) it is the only value produced for two different inputs.

In the integer domain, the values 0 and 1 exhibit special characteristics if arithmetic operations are performed. These characteristics are $0 \times x = 0$ and $1 \times x = x$ for all values of x. Therefore, all the special values in the input and output domains of a program must be considered while testing the program.

Operational Profile As the term suggests, an *operational profile* is a quantitative characterization of how a system will be used. It was created to guide test engineers in selecting test cases (inputs) using samples of system usage. The notion of operational profiles, or *usage profiles*, was developed by Mills et al. [52] at IBM in the context of Cleanroom Software Engineering and by Musa [37] at AT&T Bell Laboratories to help develop software systems with better reliability. The idea is to infer, from the observed test results, the future reliability of the software when it is in actual use. To do this, test inputs are assigned a probability distribution, or profile, according to their occurrences in actual operation. The ways test engineers assign probability and select test cases to operate a system may significantly differ from the ways actual users operate a system. However, for accurate estimation of the reliability of a system it is important to test a system by considering the ways it will actually be used in the field. This concept is being used to test web

applications, where the user session data are collected from the web servers to select test cases [53, 54].

Fault Model Previously encountered faults are an excellent source of information in designing new test cases. The known faults are classified into different classes, such as initialization faults, logic faults, and interface faults, and stored in a repository [55, 56]. Test engineers can use these data in designing tests to ensure that a particular class of faults is not resident in the program.

There are three types of fault-based testing: error guessing, fault seeding, and mutation analysis. In error guessing, a test engineer applies his experience to (i) assess the situation and guess where and what kinds of faults might exist, and (ii) design tests to specifically expose those kinds of faults. In fault seeding, known faults are injected into a program, and the test suite is executed to assess the effectiveness of the test suite. Fault seeding makes an assumption that a test suite that finds seeded faults is also likely to find other faults. Mutation analysis is similar to fault seeding, except that mutations to program statements are made in order to determine the fault detection capability of the test suite. If the test cases are not capable of revealing such faults, the test engineer may specify additional test cases to reveal the faults. Mutation testing is based on the idea of fault simulation, whereas fault seeding is based on the idea of fault injection. In the fault injection approach, a fault is inserted into a program, and an oracle is available to assert that the inserted fault indeed made the program incorrect. On the other hand, in fault simulation, a program modification is not guaranteed to lead to a faulty program. In fault simulation, one may modify an incorrect program and turn it into a correct program.

1.15 WHITE-BOX AND BLACK-BOX TESTING

A key idea in Section 1.14 was that test cases need to be designed by considering information from several sources, such as the specification, source code, and special properties of the program's input and output domains. This is because all those sources provide complementary information to test designers. Two broad concepts in testing, based on the sources of information for test design, are *white-box* and *black-box* testing. White-box testing techniques are also called *structural testing* techniques, whereas black-box testing techniques are called *functional testing* techniques.

In structural testing, one primarily examines *source code* with a focus on control flow and data flow. Control flow refers to flow of control from one instruction to another. Control passes from one instruction to another instruction in a number of ways, such as one instruction appearing after another, function call, message passing, and interrupts. Conditional statements alter the normal, sequential flow of control in a program. Data flow refers to the propagation of values from one variable or constant to another variable. Definitions and uses of variables determine the data flow aspect in a program.

In functional testing, one does not have access to the internal details of a program and the program is treated as a black box. A test engineer is concerned only with the part that is accessible outside the program, that is, just the input and the externally visible outcome. A test engineer applies input to a program, observes the externally visible outcome of the program, and determines whether or not the program outcome is the expected outcome. Inputs are selected from the program's requirements specification and properties of the program's input and output domains. A test engineer is concerned only with the functionality and the features found in the program's specification.

At this point it is useful to identify a distinction between the scopes of structural testing and functional testing. One applies structural testing techniques to individual units of a program, whereas functional testing techniques can be applied to both an entire system and the individual program units. Since individual programmers know the details of the source code they write, they themselves perform structural testing on the individual program units they write. On the other hand, functional testing is performed at the external interface level of a system, and it is conducted by a separate software quality assurance group.

Let us consider a program unit U which is a part of a larger program P. A program unit is just a piece of source code with a well-defined objective and well-defined input and output domains. Now, if a programmer derives test cases for testing U from a knowledge of the internal details of U, then the programmer is said to be performing structural testing. On the other hand, if the programmer designs test cases from the stated objective of the unit U and from his or her knowledge of the special properties of the input and output domains of U, then he or she is said to be performing functional testing on the same unit U.

The ideas of structural testing and functional testing do not give programmers and test engineers a choice of whether to design test cases from the source code or from the requirements specification of a program. However, these strategies are used by different groups of people at different times during a software's life cycle. For example, individual programmers use both the structural and functional testing techniques to test their own code, whereas quality assurance engineers apply the idea of functional testing.

Neither structural testing nor functional testing is by itself good enough to detect most of the faults. Even if one selects all possible inputs, a structural testing technique cannot detect all faults if there are *missing paths* in a program. Intuitively, a path is said to be missing if there is no code to handle a possible condition. Similarly, without knowledge of the structural details of a program, many faults will go undetected. Therefore, a combination of both structural and functional testing techniques must be used in program testing.

1.16 TEST PLANNING AND DESIGN

The purpose of system test planning, or simply test planning, is to get ready and organized for test execution. A test plan provides a framework, scope, details of resource needed, effort required, schedule of activities, and a budget. A framework

is a set of ideas, facts, or circumstances within which the tests will be conducted. The stated scope outlines the domain, or extent, of the test activities. The scope covers the managerial aspects of testing, rather than the detailed techniques and specific test cases.

Test design is a critical phase of software testing. During the test design phase, the system requirements are critically studied, system features to be tested are thoroughly identified, and the objectives of test cases and the detailed behavior of test cases are defined. Test objectives are identified from different sources, namely, the requirement specification and the functional specification, and one or more test cases are designed for each test objective. Each test case is designed as a combination of modular test components called *test steps*. These test steps can be combined together to create more complex, multistep tests. A test case is clearly specified so that others can easily borrow, understand, and reuse it.

It is interesting to note that a new test-centric approach to system development is gradually emerging. This approach is called test-driven development (TDD) [57]. In test-driven development, programmers design and implement test cases before the production code is written. This approach is a key practice in modern agile software development processes such as XP. The main characteristics of agile software development processes are (i) incremental development, (ii) coding of unit and acceptance tests conducted by the programmers along with customers, (iii) frequent regression testing, and (iv) writing test code, one test case at a time, before the production code.

1.17 MONITORING AND MEASURING TEST EXECUTION

Monitoring and measurement are two key principles followed in every scientific and engineering endeavor. The same principles are also applicable to the testing phases of software development. It is important to monitor certain metrics which truly represent the progress of testing and reveal the quality level of the system. Based on those metrics, the management can trigger corrective and preventive actions. By putting a small but critical set of metrics in place the executive management will be able to know whether they are on the right track [58]. Test execution metrics can be broadly categorized into two classes as follows:

- Metrics for monitoring test execution
- Metrics for monitoring defects

The first class of metrics concerns the process of executing test cases, whereas the second class concerns the defects found as a result of test execution. These metrics need to be tracked and analyzed on a periodic basis, say, daily or weekly. In order to effectively control a test project, it is important to gather valid and accurate information about the project. One such example is to precisely know when to trigger revert criteria for a test cycle and initiate root cause analysis of

the problems before more tests can be performed. By triggering such a revert criteria, a test manager can effectively utilize the time of test engineers, and possibly money, by suspending a test cycle on a product with too many defects to carry out a meaningful system test. A management team must identify and monitor metrics while testing is in progress so that important decisions can be made [59]. It is important to analyze and understand the test metrics, rather than just collect data and make decisions based on those raw data. Metrics are meaningful only if they enable the management to make decisions which result in lower cost of production, reduced delay in delivery, and improved quality of software systems.

Quantitative evaluation is important in every scientific and engineering field. Quantitative evaluation is carried out through measurement. Measurement lets one evaluate parameters of interest in a quantitative manner as follows:

- Evaluate the effectiveness of a technique used in performing a task. One can evaluate the effectiveness of a test generation technique by counting the number of defects detected by test cases generated by following the technique and those detected by test cases generated by other means.
- Evaluate the productivity of the development activities. One can keep track of productivity by counting the number of test cases designed per day, the number of test cases executed per day, and so on.
- Evaluate the quality of the product. By monitoring the number of defects detected per week of testing, one can observe the quality level of the system.
- Evaluate the product testing. For evaluating a product testing process, the following two measurements are critical:

  Test case effectiveness metric: The objective of this metric is twofold as explained in what follows: (1) measure the "defect revealing ability" of the test suite and (2) use the metric to improve the test design process. During the unit, integration, and system testing phases, faults are revealed by executing the planned test cases. In addition to these faults, new faults are also found during a testing phase for which no test cases had been designed. For these new faults, new test cases are added to the test suite. Those new test cases are called test case escaped (TCE). Test escapes occur because of deficiencies in test design. The need for more testing occurs as test engineers get new ideas while executing the planned test cases.

  Test effort effectiveness metric: It is important to evaluate the effectiveness of the testing effort in the development of a product. After a product is deployed at the customer's site, one is interested to know the effectiveness of testing that was performed. A common measure of test effectiveness is the number of defects found by the customers that were not found by the test engineers prior to the release of the product. These defects had escaped our test effort.

1.18 TEST TOOLS AND AUTOMATION

In general, software testing is a highly labor intensive task. This is because test cases are to a great extent manually generated and often manually executed. Moreover, the results of test executions are manually analyzed. The durations of those tasks can be shortened by using appropriate tools. A test engineer can use a variety of tools, such as a *static code analyzer*, a *test data generator*, and a *network analyzer*, if a network-based application or protocol is under test. Those tools are useful in increasing the efficiency and effectiveness of testing.

Test automation is essential for any testing and quality assurance division of an organization to move forward to become more efficient. The benefits of test automation are as follows:

- Increased productivity of the testers
- Better coverage of regression testing
- Reduced durations of the testing phases
- Reduced cost of software maintenance
- Increased effectiveness of test cases

Test automation provides an opportunity to improve the skills of the test engineers by writing programs, and hence their morale. They will be more focused on developing automated test cases to avoid being a bottleneck in product delivery to the market. Consequently, software testing becomes less of a tedious job.

Test automation improves the coverage of regression testing because of accumulation of automated test cases over time. Automation allows an organization to create a rich library of reusable test cases and facilitates the execution of a consistent set of test cases. Here consistency means our ability to produce repeated results for the same set of tests. It may be very difficult to reproduce test results in manual testing, because exact conditions at the time and point of failure may not be precisely known. In automated testing it is easier to set up the initial conditions of a system, thereby making it easier to reproduce test results. Test automation simplifies the debugging work by providing a detailed, unambiguous log of activities and intermediate test steps. This leads to a more organized, structured, and reproducible testing approach.

Automated execution of test cases reduces the elapsed time for testing, and, thus, it leads to a shorter time to market. The same automated test cases can be executed in an unsupervised manner at night, thereby efficiently utilizing the different platforms, such as hardware and configuration. In short, automation increases test execution efficiency. However, at the end of test execution, it is important to analyze the test results to determine the number of test cases that passed or failed. And, if a test case failed, one analyzes the reasons for its failure.

In the long run, test automation is cost-effective. It drastically reduces the software maintenance cost. In the sustaining phase of a software system, the regression tests required after each change to the system are too many. As a result, regression testing becomes too time and labor intensive without automation.

A repetitive type of testing is very cumbersome and expensive to perform manually, but it can be automated easily using software tools. A simple repetitive type of application can reveal memory leaks in a software. However, the application has to be run for a significantly long duration, say, for weeks, to reveal memory leaks. Therefore, manual testing may not be justified, whereas with automation it is easy to reveal memory leaks. For example, stress testing is a prime candidate for automation. Stress testing requires a worst-case load for an extended period of time, which is very difficult to realize by manual means. Scalability testing is another area that can be automated. Instead of creating a large test bed with hundreds of equipment, one can develop a simulator to verify the scalability of the system.

Test automation is very attractive, but it comes with a price tag. Sufficient time and resources need to be allocated for the development of an automated test suite. Development of automated test cases need to be managed like a programming project. That is, it should be done in an organized manner; otherwise it is highly likely to fail. An automated test suite may take longer to develop because the test suite needs to be debugged before it can be used for testing. Sufficient time and resources need to be allocated for maintaining an automated test suite and setting up a test environment. Moreover, every time the system is modified, the modification must be reflected in the automated test suite. Therefore, an automated test suite should be designed as a modular system, coordinated into reusable libraries, and cross-referenced and traceable back to the feature being tested.

It is important to remember that test automation cannot replace manual testing. Human creativity, variability, and observability cannot be mimicked through automation. Automation cannot detect some problems that can be easily observed by a human being. Automated testing does not introduce minor variations the way a human can. Certain categories of tests, such as usability, interoperability, robustness, and compatibility, are often not suited for automation. It is too difficult to automate all the test cases; usually 50% of all the system-level test cases can be automated. There will always be a need for some manual testing, even if all the system-level test cases are automated.

The objective of test automation is not to reduce the head counts in the testing department of an organization, but to improve the productivity, quality, and efficiency of test execution. In fact, test automation requires a larger head count in the testing department in the first year, because the department needs to automate the test cases and simultaneously continue the execution of manual tests. Even after the completion of the development of a test automation framework and test case libraries, the head count in the testing department does not drop below its original level. The test organization needs to retain the original team members in order to improve the quality by adding more test cases to the automated test case repository.

Before a test automation project can proceed, the organization must assess and address a number of considerations. The following list of prerequisites must be considered for an assessment of whether the organization is ready for test automation:

- The test cases to be automated are well defined.
- Test tools and an infrastructure are in place.

- The test automation professionals have prior successful experience in automation.
- Adequate budget should have been allocated for the procurement of software tools.

1.19 TEST TEAM ORGANIZATION AND MANAGEMENT

Testing is a distributed activity conducted at different levels throughout the life cycle of a software. These different levels are unit testing, integration testing, system testing, and acceptance testing. It is logical to have different testing groups in an organization for each level of testing. However, it is more logical—and is the case in reality—that unit-level tests be developed and executed by the programmers themselves rather than an independent group of unit test engineers. The programmer who develops a software unit should take the ownership and responsibility of producing good-quality software to his or her satisfaction. System integration testing is performed by the system integration test engineers. The integration test engineers involved need to know the software modules very well. This means that all development engineers who collectively built all the units being integrated need to be involved in integration testing. Also, the integration test engineers should thoroughly know the build mechanism, which is key to integrating large systems.

A team for performing system-level testing is truly separated from the development team, and it usually has a separate head count and a separate budget. The mandate of this group is to ensure that the system requirements have been met and the system is acceptable. Members of the system test group conduct different categories of tests, such as functionality, robustness, stress, load, scalability, reliability, and performance. They also execute business acceptance tests identified in the user acceptance test plan to ensure that the system will eventually pass user acceptance testing at the customer site. However, the real user acceptance testing is executed by the client's special user group. The user group consists of people from different backgrounds, such as software quality assurance engineers, business associates, and customer support engineers. It is a common practice to create a temporary user acceptance test group consisting of people with different backgrounds, such as integration test engineers, system test engineers, customer support engineers, and marketing engineers. Once the user acceptance is completed, the group is dismantled. It is recommended to have at least two test groups in an organization: integration test group and system test group.

Hiring and retaining test engineers are challenging tasks. Interview is the primary mechanism for evaluating applicants. Interviewing is a skill that improves with practice. It is necessary to have a recruiting process in place in order to be effective in hiring excellent test engineers. In order to retain test engineers, the management must recognize the importance of testing efforts at par with development efforts. The management should treat the test engineers as professionals and as a part of the overall team that delivers quality products.

1.20 OUTLINE OF BOOK

With the above high-level introduction to quality and software testing, we are now in a position to outline the remaining chapters. Each chapter in the book covers technical, process, and/or managerial topics related to software testing. The topics have been designed and organized to facilitate the reader to become a software test specialist. In Chapter 2 we provide a self-contained introduction to the theory and limitations of software testing.

Chapters 3–6 treat unit testing techniques one by one, as quantitatively as possible. These chapters describe both static and dynamic unit testing. Static unit testing has been presented within a general framework called *code review*, rather than individual techniques called *inspection* and *walkthrough*. Dynamic unit testing, or execution-based unit testing, focuses on control flow, data flow, and domain testing. The JUnit framework, which is used to create and execute dynamic unit tests, is introduced. We discuss some tools for effectively performing unit testing.

Chapter 7 discusses the concept of integration testing. Specifically, five kinds of integration techniques, namely, top down, bottom up, sandwich, big bang, and incremental, are explained. Next, we discuss the integration of hardware and software components to form a complete system. We introduce a framework to develop a plan for system integration testing. The chapter is completed with a brief discussion of integration testing of off-the-shelf components.

Chapters 8–13 discuss various aspects of system-level testing. These six chapters introduce the reader to the technical details of system testing that is the practice in industry. These chapters promote both qualitative and quantitative evaluation of a system testing process. The chapters emphasize the need for having an independent system testing group. A process for monitoring and controlling system testing is clearly explained. Chapter 14 is devoted to acceptance testing, which includes acceptance testing criteria, planning for acceptance testing, and acceptance test execution.

Chapter 15 contains the fundamental concepts of software reliability and their application to software testing. We discuss the notion of *operation profile* and its application in system testing. We conclude the chapter with the description of an example and the time of releasing a system by determining the additional length of system testing. The additional testing time is calculated by using the idea of software reliability.

In Chapter 16, we present the structure of test groups and how these groups can be organized in a software company. Next, we discuss how to hire and retain test engineers by providing training, instituting a reward system, and establishing an attractive career path for them within the testing organization. We conclude this chapter with the description of how to build and manage a test team with a focus on teamwork rather than individual gain.

Chapters 17 and 18 explain the concepts of software quality and different maturity models. Chapter 17 focuses on quality factors and criteria and describes the ISO 9126 and ISO 9000:2000 standards. Chapter 18 covers the CMM, which

was developed by the SEI at Carnegie Mellon University. Two test-related models, namely the TPI model and the TMM, are explained at the end of Chapter 18.

We define the key words used in the book in a glossary at the end of the book. The reader will find about 10 practice exercises at the end of each chapter. A list of references is included at the end of each chapter for a reader who would like to find more detailed discussions of some of the topics. Finally, each chapter, except this one, contains a literature review section that, essentially, provides pointers to more advanced material related to the topics. The more advanced materials are based on current research and alternate viewpoints.

REFERENCES

1. B. Davis, C. Skube, L. Hellervik, S. Gebelein, and J. Sheard. *Successful Manager's Handbook*. Personnel Decisions International, Minneapolis, 1996.
2. M. Walton. *The Deming Management Method*. The Berkley Publishing Group, New York, 1986.
3. W. E. Deming. Transcript of Speech to GAO Roundtable on Product Quality—Japan vs. the United States. *Quality Progress*, March 1994, pp. 39–44.
4. W. A. Shewhart. *Economic Control of Quality of Manufactured Product*. Van Nostrand, New York, 1931.
5. W. A. Shewhart. The Application of Statistics as an Aid in Maintaining Quality of a Manufactured Product. *Journal of American Statistical Association*, December 1925, pp. 546–548.
6. National Institute of Standards and Technology, *Baldridge National Quality Program*, 2008. Available: http://www.quality.nist.gov/.
7. J. Liker and D. Meier. *The Toyota Way Fieldbook*. McGraw-Hill, New York, 2005.
8. M. Poppendieck and T. Poppendieck. *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley, Reading, MA, 2006.
9. A. B. Godfrey and A. I. C. Endres. The Evolution of Quality Management Within Telecommunications. *IEEE Communications Magazine*, October 1994, pp. 26–34.
10. M. Pecht and W. R. Boulton. *Quality Assurance and Reliability in the Japanese Electronics Industry*. Japanses Technology Evaluation Center (JTEC), Report on Electronic Manufacturing and Packaging in Japan, W. R. Boulton, Ed. International Technology Research Institute at Loyola College, February 1995, pp. 115–126.
11. A. V. Feigenbaum. *Total Quality Control*, 4th ed. McGraw-Hill, New York, 2004.
12. K. Ishikawa. *What Is Total Quality Control*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
13. A. Cockburn. What Engineering Has in Common With Manufacturing and Why It Matters. *Crosstalk, the Journal of Defense Software Engineering*, April 2007, pp. 4–7.
14. S. Land. *Jumpstart CMM/CMMI Software Process Improvement*. Wiley, Hoboken, NJ, 2005.
15. W. E. Deming. *Out of the Crisis*. MIT, Cambridge, MA, 1986.
16. J. M. Juran and A. B. Godfrey. *Juran's Quality Handbook*, 5th ed. McGraw-Hill, New York, 1998.
17. P. Crosby. *Quality Is Free*. New American Library, New York, 1979.
18. D. A. Garvin. What Does "Product Quality" Really Mean? *Sloan Management Review*, Fall 1984, pp. 25–43.
19. J. A. McCall, P. K. Richards, and G. F. Walters. Factors in Software Quality, Technical Report RADC-TR-77-369. U.S. Department of Commerce, Washington, DC, 1977.
20. International Organization for Standardization (ISO). Quality Management Systems—Fundamentals and Vocabulary, ISO 9000:2000. ISO, Geneva, December 2000.
21. International Organization for Standardization (ISO). Quality Management Systems—Guidelines for Performance Improvements, ISO 9004:2000. ISO, Geneva, December 2000.
22. International Organization for Standardization (ISO). Quality Management Systems—Requirements, ISO 9001:2000. ISO, Geneva, December 2000.
23. T. Koomen and M. Pol. *Test Process Improvement*. Addison-Wesley, Reading, MA, 1999.

24. I. Burnstein. *Practical Software Testing*. Springer, New York, 2003.
25. L. Osterweil et al. Strategic Directions in Software Quality. *ACM Computing Surveys*, December 1996, pp. 738–750.
26. M. A. Friedman and J. M. Voas. *Software Assessment: Reliability, Safety, Testability*. Wiley, New York, 1995.
27. Michael D. Ernst. Static and Dynamic Analysis: Synergy and Duality. Paper presented at ICSE Workshop on Dynamic Analysis, Portland, OR, May 2003, pp. 24–27.
28. L. Baresi and M. Pezzè. *An Introduction to Software Testing*, Electronic Notes in Theoretical Computer Science. Elsevier, Vol. 148, Feb. 2006, pp. 89–111.
29. K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley, Reading, MA, 2004.
30. B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
31. J. C. Laprie. Dependability—Its Attributes, Impairments and Means. In *Predictably Dependable Computing Systems*, B. Randall, J. C. Laprie, H. Kopetz, and B. Littlewood, Eds. Springer-Verlag, New York, 1995.
32. R. Chillarege. What Is Software Failure. *IEEE Transactions on Reliability*, September 1996, pp. 354–355.
33. R. Rees. What Is a Failure. *IEEE Transactions on Reliability*, June 1997, p. 163.
34. B. Parhami. Defect, Fault, Error, . . . , or Failure. *IEEE Transactions on Reliability*, December 1997, pp. 450–451.
35. M. R. Lyu. *Handbook of Software Reliability Engineering*. McGraw-Hill, New York, 1995.
36. R. Hamlet. Random Testing. In *Encyclopedia of Software Engineering*, J. Marciniak, Ed. Wiley, New York, 1994, pp. 970–978.
37. J. D. Musa. Software Reliability Engineering. *IEEE Software*, March 1993, pp. 14–32.
38. J. D. Musa. A Theory of Software Reliability and Its Application. *IEEE Transactions on Software Engineering*, September 1975, pp. 312–327.
39. M. Glinz and R. J. Wieringa. Stakeholders in Requirements Engineering. *IEEE Software*, March–April 2007, pp. 18–20.
40. A. Bertolino and L. Strigini. On the Use of Testability Measures for Dependability Assessment. *IEEE Transactions on Software Engineering*, February 1996, pp. 97–108.
41. A. Bertolino and E Marchelli. *A Brief Essay on Software Testing*. In *Software Engineering, Vol. 1, The Development Process*, 3rd ed., R. H. Thayer and M. J. Christensen, Eds. Wiley–IEEE Computer Society Press, Hoboken, NJ, 2005.
42. D. Jeffrey and N. Gupta. Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction. *IEEE Transactions on Software Engineering*, February 2007, pp. 108–123.
43. Z. Li, M. Harman, and R. M. Hierons. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering*, April 2007, pp. 225–237.
44. W. Masri, A. Podgurski, and D. Leon. An Empirical Study of Test Case Filtering Techniques Based on Exercising Information Flows. *IEEE Transactions on Software Engineering*, July 2007, pp. 454–477.
45. W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of IEEE WESCON*, August 1970, pp. 1–9. Republished in ICSE, Monterey, 1987, pp. 328–338.
46. L. Rising and N. S. Janoff. The Scrum Software Development Process for Small Teams. *IEEE Software*, July/August 2000, pp. 2–8.
47. K. Schwaber. *Agile Project Management with Scrum*. Microsoft Press, Redmond, WA, 2004.
48. H. Takeuchi and I. Nonaka. The New Product Development Game. *Harvard Business Review*, Boston, January-February 1986, pp. 1–11.
49. A. P. Mathur. *Foundation of Software Testing*. Pearson Education, New Delhi, 2007.
50. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
51. M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, Hoboken, NJ, 2007.
52. H. D. Mills, M. Dyer, and R. C. Linger. Cleanroom Software Engineering. *IEEE Software*, September 1987, pp. 19–24.

53. S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging User Session Data to Support Web Application Testing. *IEEE Transactions on Software Engineering*, March 2005, pp. 187–202.
54. S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. S. Greenwald. Applying Concept Analysis to User-Session-Based Testing of Web Applications. *IEEE Transactions on Software Engineering*, October 2007, pp. 643–657.
55. A. Endress. An Analysis of Errors and Their Causes in System Programs. *IEEE Transactions on Software Engineering*, June 1975, pp. 140–149.
56. T. J. Ostrand and E. J. Weyuker. Collecting and Categorizing Software Error Data in an Industrial Environment. *Journal of Systems and Software*, November 1984, pp. 289–300.
57. K. Beck. *Test-Driven Development*. Addison-Wesley, Reading, MA, 2003.
58. D. Lemont. *CEO Discussion—From Start-up to Market Leader—Breakthrough Milestones*. Ernst and Young Milestones, Boston, May 2004, pp. 9–11.
59. G. Stark, R. C. Durst, and C. W. Vowell. Using Metrics in Management Decision Making. *IEEE Computer*, September 1994, pp. 42–48.
60. B. Kitchenham and S. L. Pfleeger. Software Quality: The Elusive Target. *IEEE Software*, January 1996, pp. 12–21.
61. J. Kilpatrick. Lean Principles. http://www.mep.org/textfiles/LeanPrinciples.pdf, 2003, pp. 1–5.

Exercises

1. Explain the principles of statistical quality control. What are the tools used for this purpose? Explain the principle of a control chart.
2. Explain the concept of lean principles.
3. What is an "Ishikawa" diagram? When should the Ishikawa diagram be used? Provide a procedure to construct an Ishikawa diagram.
4. What is total quality management (TQM)? What is the difference between TQM and TQC?
5. Explain the differences between *validation* and *verification*.
6. Explain the differences between *failure, error*, and *fault*.
7. What is a test case? What are the objectives of testing?
8. Explain the concepts of *unit, integration, system, acceptance*, and *regression* testing.
9. What are the different sources from which test cases can be selected?
10. What is the difference between *fault injection* and *fault simulation*?
11. Explain the differences between *structural* and *functional* testing.
12. What are the strengths and weaknesses of automated testing and manual testing?

CHAPTER 2

Theory of Program Testing

He who loves practice without theory is like the sailor who boards [a] ship without a rudder and compass and never knows where he may cast.
— *Leonardo da Vinci*

2.1 BASIC CONCEPTS IN TESTING THEORY

The idea of program testing is as old as computer programming. As computer programs got larger and larger since their early days in the 1960s, the need for eliminating defects from them in a systematic manner received more attention. Both the research community and the practitioners became more deeply involved in software testing. Thus, in the 1970s, a new field of research called *testing theory* emerged. Testing theory puts emphasis on the following:

- Detecting defects through execution-based testing
- Designing test cases from different sources, namely, requirement specification, source code, and the input and output domains of programs
- Selecting a subset of test cases from the set of all possible test cases [1, 2]
- Effectiveness of the test case selection strategy [3–5]
- Test oracles used during testing [6, 7]
- Prioritizing the execution of the selected test cases [8]
- Adequacy analysis of test cases [9–15]

A theoretical foundation of testing gives testers and developers valuable insight into software systems and the development processes. As a consequence, testers design more effective test cases at a lower cost. While considering testing theory, there may be a heightened expectation that it lets us detect all the defects in a computer program. Any testing theory must inherit the fundamental limitation of testing. The limitation of testing has been best articulated by Dijkstra: *Testing can only reveal the presence of errors, never their absence* [16]. In spite of the

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

said limitation, testing remains as the most practical and reliable method for defect detection and quality improvement.

In this chapter, three well-known testing theories are discussed. These are Goodenough and Gerhart's theory [17], Weyuker and Ostrand's theory [18], and Gourlay's theory [19]. Goodenough and Gerhart introduced some key concepts such as *an ideal test*, *reliability* and *validity* of a test, *test selection criteria*, *thorough test*, and five categories of program errors. Weyuker and Ostrand refined some of the above ideas in the form of *uniformly reliable criterion*, *uniformly valid criterion*, and *uniformly ideal test*. Gourlay introduced the concept of a *test system* and a general method for comparing different test methods.

2.2 THEORY OF GOODENOUGH AND GERHART

Goodenough and Gerhart published a seminal paper [17] in 1975 on test data selection. This paper gave a fundamental testing concept, identified a few types of program errors, and gave a theory for selecting test data from the input domain of a program. Though this theory is not without critiques, it is widely quoted and appreciated in the research community of software testing.

2.2.1 Fundamental Concepts

Let D be the input domain of a program P. Let $T \subseteq D$. The result of executing P with input $d \in D$ is denoted by $P(d)$ (Figure 2.1):

OK(d): Define a predicate OK(d) which expresses the acceptability of result $P(d)$. Thus, OK(d) = *true* if and only if $P(d)$ is an acceptable outcome.

SUCCESSFUL(T): For a given $T \subseteq D$, T is a *successful* test, denoted by SUCCESSFUL(T), if and only if, $\forall t \in$ T, OK(t). Thus, SUCCESSFUL(T) = true if and only if, $\forall t \in$ T, OK(t).

Ideal Test: T constitutes an *ideal test* if

$$\text{OK}(t)\ \forall t \in T \Rightarrow \text{OK}(d)\ \forall d \in D$$

An ideal test is interpreted as follows. If from the successful execution of a sample of the input domain we can conclude that the program contains no errors, then the sample constitutes an ideal test. Practitioners may

Figure 2.1 Executing a program with a subset of the input domain.

loosely interpret "no error" as "not many errors of severe consequences." The validity of the above definition of an ideal test depends on how "thoroughly" T exercises P. Some people equate *thorough test* with exhaustive or complete test, in which case $T = D$.

COMPLETE(T, C): A *thorough test* T is defined to be one satisfying COMPLETE(T,C), where COMPLETE is a predicate that defines how some test *selection criteria* C is used in selecting a particular set of test data T from D. COMPLETE(T, C) will be defined in a later part of this section. Essentially, C defines the properties of a program that must be exercised to constitute a thorough test.

Reliable Criterion: A selection criterion C is *reliable* if and only if either every test selected by C is successful or no test selected is successful. Thus, reliability refers to consistency.

Valid Criterion: A selection criterion C is *valid* if and only if whenever P is incorrect C selects at least one test set T which is not successful for P. Thus, validity refers to the ability to produce meaningful results.

Fundamental Theorem. $(\exists T \subseteq D)(\text{COMPLETE}(T, C) \wedge \text{RELIABLE}(C) \wedge \text{VALID}(C) \wedge \text{SUCCESSFUL}(T)) \Rightarrow (\forall d \in D)\text{OK}(d)$

Proof. Let P be a program and D be the set of inputs for P. Let d be a member of D. We assume that P fails on input d. In other words, the actual outcome of executing P with input d is not the same as the expected outcome. In the form of our notation, $\neg\text{OK}(d)$ is true. VALID(C) implies that there exists a complete set of test data T such that $\neg$SUCCESSFUL(T). RELIABLE(C) implies that if one complete test fails, all tests fail. However, this leads to a contradiction that there exists a complete test that is successfully executed.

One may be tempted to find a reliable and valid criterion, if it exists, so that all faults can be detected with a small set of test cases. However, there are several difficulties in applying the above theory, as explained in the following:

- Since faults in a program are unknown, it is impossible to prove the reliability and validity of a criterion. A criterion is guaranteed to be both reliable and valid if it selects the entire input domain D. However, this is undesirable and impractical.
- Neither reliability nor validity is preserved during the debugging process, where faults keep disappearing.
- If the program P is correct, then any test will be successful and every selection criterion is reliable and valid.
- If P is not correct, there is in general no way of knowing whether a criterion is ideal without knowing the errors in P.

2.2.2 Theory of Testing

Let D be the input domain of a program P. Let C denote a set of test predicates. If $d \in D$ satisfies test predicate $c \in C$, then $c(d)$ is said to be true. Selecting data to satisfy a test predicate means selecting data to exercise the condition combination in the course of executing P.

With the above idea in mind, COMPLETE(T, C), where $T \subseteq D$, is defined as follows:

$$\text{COMPLETE}(T, C) \equiv (\forall c \in C)(\exists t \in T)c(t) \wedge (\forall t \in T)(\exists c \in C)c(t)$$

The above theory means that, for every test predicate, we select a test such that the test predicate is satisfied. Also, for every test selected, there exists a test predicate which is satisfied by the selected test. The definitions of an ideal test and thoroughness of a test do not reveal any relationship between them. However, we can establish a relationship between the two in the following way.

Let B be the set of faults (or bugs) in a program P revealed by an ideal test T_I. Let a test engineer identify a set of test predicates C_1 and design a set of test cases T_1, such that COMPLETE(T_1, C_1) is satisfied. Let B_1 represent the set of faults revealed by T_1. There is no guarantee that T_1 reveals all the faults. Later, the test engineer identifies a larger set of test predicates C_2 such that $C_2 \supset C_1$ and designs a new set of test cases T_2 such that $T_2 \supset T_1$ and COMPLETE(T_2, C_2) is satisfied. Let B_2 be the set of faults revealed by T_2. Assuming that the additional test cases selected reveal more faults, we have $B_2 \supset B_1$. If the test engineer repeats this process, he may ultimately identify a set of test predicates C_I and design a set of test cases T_I such that COMPLETE(T_I, C_I) is satisfied and T_I reveals the entire set of faults B. In this case, T_I is a thorough test satisfying COMPLETE(T_I, C_I) and represents an ideal test set.

2.2.3 Program Errors

Any approach to testing is based on assumptions about the way program faults occur. Faults are due to two main reasons:

- Faults occur due to our inadequate understanding of all conditions with which a program must deal.
- Faults occur due to our failure to realize that certain combinations of conditions require special treatments.

Goodenough and Gerhart classify program faults as follows:

- **Logic Fault:** This class of faults means a program produces incorrect results independent of resources required. That is, the program fails because of the faults present in the program and not because of a lack of resources. Logic faults can be further split into three categories:

  Requirements fault: This means our failure to capture the real requirements of the customer.

Design fault: This represents our failure to satisfy an understood requirement.

Construction fault: This represents our failure to satisfy a design. Suppose that a design step says "Sort array *A*." To sort the array with *N* elements, one may choose one of several sorting algorithms. Let

```
for (i = 0; i < N; i++) {
          :
}
```

be the desired for loop construct to sort the array. If a programmer writes the for loop in the form

```
for (i = 0; i <= N; i++){
          :
}
```

then there is a construction error in the implementation.

- **Performance Fault:** This class of faults leads to a failure of the program to produce expected results within specified or desired resource limitations.

A thorough test must be able to detect faults arising from any of the above reasons. Test data selection criteria must reflect information derived from each stage of software development. Since each type of fault is manifested as an improper effect produced by an implementation, it is useful to categorize the sources of faults in implementation terms as follows:

Missing Control Flow Paths: Intuitively, a control flow path, or simply a path, is a feasible sequence of instructions in a program. A path may be missing from a program if we fail to identify a condition and specify a path to handle that condition. An example of a missing path is our failure to test for a zero divisor before executing a division. If we fail to recognize that a divisor can take a zero value, then we will not include a piece of code to handle the special case. Thus, a certain desirable computation will be missing from the program.

Inappropriate Path Selection: A program executes an inappropriate path if a condition is expressed incorrectly. In Figure 2.2, we show a desired behavior and an implemented behavior. Both the behaviors are identical except in the condition part of the if statement. The if part of the implemented behavior contains an additional condition *B*. It is easy to see that

Desired behavior	**Implemented behavior**
if (A) proc1();	if (A && B) proc1();
else proc2();	else proc2();

Figure 2.2 Example of inappropriate path selection.

both the desired part and the implemented part behave in the same way for all combinations of values of A and B except when $A = 1$ and $B = 0$.

Inappropriate or Missing Action: There are three instances of this class of fault:

- One may calculate a value using a method that does not necessarily give the correct result. For example, a desired expression is $x = x \times w$, whereas it is wrongly written as $x = x + w$. These two expressions produce identical results for several combinations of x and w, such as $x = 1.5$ and $w = 3$, for example.
- Failing to assign a value to a variable is an example of a missing action.
- Calling a function with the wrong argument list is a kind of inappropriate action.

The main danger due to an inappropriate or missing action is that the action is incorrect only under certain combinations of conditions. Therefore, one must do the following to find test data that reliably reveal errors:

- Identify all the conditions relevant to the correct operation of a program.
- Select test data to exercise all possible combinations of these conditions.

The above idea of selecting test data leads us to define the following terms:

Test Data: Test data are actual values from the input domain of a program that collectively satisfy some test selection criteria.

Test Predicate: A test predicate is a description of conditions and combinations of conditions relevant to correct operation of the program:

- Test predicates describe the *aspects* of a program that are to be tested. Test data cause these aspects to be tested.
- Test predicates are the motivating force for test data selection.
- Components of test predicates arise first and primarily from the specifications for a program.
- Further conditions and predicates may be added as implementations are considered.

2.2.4 Conditions for Reliability

A set of test predicates must at least satisfy the following conditions to have any chance of being reliable. These conditions are key to meaningful testing:

- Every individual branching condition in a program must be represented by an equivalent condition in C.
- Every potential termination condition in the program, for example, an overflow, must be represented by a condition in C.
- Every condition relevant to the correct operation of the program that is implied by the specification and knowledge of the data structure of the program must be represented as a condition in C.

2.2.5 Drawbacks of Theory

Several difficulties prevent us from applying Goodenough and Gerhart's theory of an *ideal test* as follows [18]:

- The concepts of reliability and validity have been defined with respect to the entire input domain of a program. A criterion is guaranteed to be both reliable and valid if and only if it selects the entire domain as a single test. Since such exhaustive testing is impractical, one will have much difficulty in assessing the reliability and validity of a criterion.
- The concepts of reliability and validity have been defined with respect to a program. A test selection criterion that is reliable and valid for one program may not be so for another program. The goodness of a test set should be independent of individual programs and the faults therein.
- Neither validity nor reliability is preserved throughout the debugging process. In practice, as program failures are observed, the program is debugged to locate the faults, and the faults are generally fixed as soon as they are found. During this debugging phase, as the program changes, so does the idealness of a test set. This is because a fault that was revealed before debugging is no more revealed after debugging and fault fixing. Thus, properties of test selection criteria are not even "monotonic" in the sense of being either always gained or preserved or always lost or preserved.

2.3 THEORY OF WEYUKER AND OSTRAND

A key problem in the theory of Goodenough and Gerhart is that the reliability and validity of a criterion depend upon the presence of faults in a program and their types. Weyuker and Ostrand [18] provide a modified theory in which the validity and reliability of test selection criteria are dependent only on the program *specification*, rather than a program. They propose the concept of a *uniformly ideal* test selection criterion for a given output specification. In the theory of Goodenough and Gerhart, implicit in the definitions of the predicates OK(d) and SUCCESSFUL(T) is a program P. By abbreviating SUCCESSFUL() as SUCC(), the two predicates are rewritten as follows:

OK(P, d): Define a predicate $\mathrm{OK}(P, d)$ which expresses the acceptability of result $P(d)$. Thus, $\mathrm{OK}(P, d) =$ true if and only if $P(d)$ is an acceptable outcome of program P.

SUCC(P, T): For a given $T \subseteq D$, T is a *successful* test for a program P, denoted by $\mathrm{SUCC}(P, T)$, if and only if, $\forall t \in T$, $\mathrm{OK}(P, t)$. Thus, $\mathrm{SUCC}(T) =$ true if and only if, $\forall t \in T$, $\mathrm{OK}(P, t)$.

With the above definitions of $\mathrm{OK}(P, d)$ and $\mathrm{SUCC}(P, T)$, the concepts of *uniformly valid* criterion, *uniformly reliable* criterion, and *uniformly ideal* test selection are defined as follows.

Uniformly Valid Criterion C: Criterion C is uniformly valid iff

$$(\forall P)[(\exists d \in D)(\neg \mathrm{OK}(P, d)) \Rightarrow (\exists T \subseteq D)(C(T) \,\&\, \neg \mathrm{SUCC}(P, T))]$$

Uniformly Reliable Criterion C: Criterion C is uniformly reliable iff

$$(\forall P)(\forall T_1, \forall T_2 \subseteq D)[(C(T_1) \,\&\, C(T_2)) \Rightarrow (\mathrm{SUCC}(P, T_1) \Leftrightarrow \mathrm{SUCC}(P, T_2))]$$

Uniformly Ideal Test Selection: A uniformly ideal test selection criterion for a given specification is both uniformly valid and uniformly reliable.

The external quantifier $(\forall P)$ binding the free variable P in the definition of uniformly valid criterion C essentially means that the rest of the predicate holds for all programs P for a given output specification. Similarly, the external quantifier $(\forall P)$ binding the free variable P in the definition of uniformly reliable criterion C means that the rest of the predicate holds for all programs P for a given output specification.

Since a uniformly ideal test selection criterion is defined over *all* programs for a given specification, it was intended to solve all the program-dependent difficulties in the definitions given by Goodenough and Gerhart. However, the concept of uniformly ideal test selection also has several flaws. For example, for any significant program there can be no uniformly ideal criterion that is not trivial in the sense of selecting the entire input domain D. A criterion C is said to be *trivially valid* if the union of all tests selected by C is D. Hence, the following theorems.

Theorem. A criterion C is uniformly valid if and only if C is trivially valid.

Proof. Obviously a trivially valid criterion is valid. Now we need to show that a criterion C which is not trivially valid cannot be uniformly valid for a given output specification. For any element d not included in any test of C, one can write a program which is incorrect for d and correct for $D - \{d\}$.

Theorem. A criterion C is uniformly reliable if and only if C selects a single test set.

Proof. If C selects only one test, it is obviously reliable for any program. Now, assume that C selects different tests T_1 and T_2 and that $t \in T_1$ but $t \notin T_2$. A program P exists which is correct with respect to test inputs in T_2 but incorrect on t. Thus, the two tests yield different results for P, and C is not reliable.

Now, we can combine the above two theorems to have the following corollary.

Corollary. A criterion C is uniformly valid and uniformly reliable if and only if C selects only the single test set $T = D$.

An important implication of the above corollary is that uniform validity and uniform reliability lead to *exhaustive testing*—and exhaustive testing is considered to be impractical. Next, the above corollary is reformulated to state that irrespective of test selection criterion used and irrespective of tests selected, except the entire D, one can always write a program which can defeat the tests. A program P is said to defeat a test T if P passes T but fails on some other valid input. This is paraphrasing the well-known statement of Dijkstra that *testing can only reveal the presence of errors, never their absence* [16].

Reliability and validity of test selection criterion are ideal goals, and ideal goals are rarely achieved. It is useful to seek less ideal but usable goals. By settling for less ideal goals, we essentially accept the reality that correctness of large programs is not something that we strive to achieve.

Weyuker and Ostrand [18] have introduced the concept of a *revealing criterion* with respect to a subdomain, where a *subdomain* S is a subset of the input domain D. A test selection criterion C is *revealing* for a subdomain S if whenever S contains an input which is processed incorrectly then every test set which satisfies C is unsuccessful. In other words, if any test selected by C is successfully executed, then every test in S produces correct output. A predicate called REVEALING(C, S) captures the above idea in the following definition:

$$\text{REVEALING}(C, S) \text{ iff } (\exists d \in S)(\neg \text{OK}(d)) \Rightarrow (\forall T \subseteq S)(C(T) \Rightarrow \neg \text{SUCC}(T))$$

The key advantage in a revealing criterion is that it concerns only a subset of the input domain, rather than the entire input domain. By considering a subset of the input domain, programmers can concentrate on local errors. An important task in applying the idea of a revealing criterion is to partition the input domain into smaller subdomains, which is akin to partitioning a problem into a set of subproblems. However, partitioning a problem into subproblems has been recognized to be a difficult task.

2.4 THEORY OF GOURLAY

An ideal goal in software development is to find out whether or not a program is correct, where a correct program is void of faults. Much research results have been reported in the field of program correctness. However, due to the highly constrained nature of program verification techniques, no developer makes any effort to prove the correctness of even small programs of, say, a few thousand lines, let alone large programs with millions of lines of code. Instead, testing is accepted in the industry as a practical way of finding faults in programs. The flip side of testing is that it cannot be used to settle the question of program correctness, which is the ideal goal. Even though testing cannot settle the program correctness issue, there is a need for a testing theory to enable us to compare the power of different test methods.

To motivate a theoretical discussion of testing, we begin with an ideal process for software development, which consists of the following steps:

- A customer and a development team specify the needs.
- The development team takes the specification and attempts to write a program to meet the specification.
- A test engineer takes both the specification and the program and selects a set of test cases. The test cases are based on the specification and the program.
- The program is executed with the selected test data, and the test outcome is compared with the expected outcome.
- The program is said to have faults if some tests fail.
- One can say the program to be ready for use if it passes all the test cases.

We focus on the selection of test cases and the interpretation of their results. We assume that the specification is correct, and the specification is the sole arbiter of the correctness of the program. The program is said to be correct if and only if it satisfies the specification. Gourlay's testing theory [19] establishes a relationship between three sets of entities, namely, *specifications*, *programs*, and *tests*, and provides a basis for comparing different methods for selecting tests.

2.4.1 Few Definitions

The set of all programs are denoted by $\mathcal{P}$, the set of all specifications by $\mathcal{S}$, and the set of all tests by $\mathcal{T}$. Members of $\mathcal{P}$ will be denoted by p and q, members of $\mathcal{S}$ will be denoted by r and s, and members of $\mathcal{T}$ will be denoted by t and u.

Uppercase letters will denote subsets of $\mathcal{P}$, $\mathcal{S}$, and $\mathcal{T}$. For examples, $p \in P \subseteq \mathcal{P}$ and $t \in T \subseteq \mathcal{T}$, where t denotes a single test case. The correctness of a program p with respect to a specification s will be denoted by $p \operatorname{corr} s$. Given s, p, and t, the predicate p ok(t) s means that the result of testing p under t is judged successful by specification s. The reader may recall that T denotes a set of test cases, and p ok(T) s is true if and only if p ok(t) s $\forall t \in T$.

We must realize that if a program is correct, then it will never produce any unexpected outcome with respect to the specification. Thus, $p \operatorname{corr} s \Rightarrow p$ ok(t) s $\forall t$.

Definition. A *testing system* is a collection $< \mathcal{P}, \mathcal{S}, \mathcal{T}, \text{corr}, \text{ok} >$, where $\mathcal{P}$, $\mathcal{S}$, and $\mathcal{T}$ are arbitrary sets, $\text{corr} \subseteq \mathcal{P} \times \mathcal{S}$, sets, ok $\subseteq \mathcal{T} \times \mathcal{P} \times \mathcal{S}$, and $\forall p \forall s \forall t (p \text{ corr } s \Rightarrow p \text{ ok}(t)s)$.

Definition. Given a testing system $< \mathcal{P}, \mathcal{S}, \mathcal{T}, \text{corr}, \text{ok} >$ a new system $< \mathcal{P}, \mathcal{S}, \mathcal{T}', \text{corr}, \text{ok}' >$ is called a *set construction*, where $\mathcal{T}'$ is the set of all subsets of $\mathcal{T}$, and where $p \text{ ok}'(T)s \Leftrightarrow \forall t (t \in T \Rightarrow p \text{ ok}(t)s)$. (The reader may recall that T is a member of $\mathcal{T}'$ because $T \subseteq \mathcal{T}$.)

Theorem. $< \mathcal{P}, \mathcal{S}, \mathcal{T}', \text{corr}, \text{ok}' >$, a set construction on a testing system $< \mathcal{P}, \mathcal{S}, \mathcal{T}, \text{corr}, \text{ok} >$, is itself a testing system.

Proof. We need to show that p corr $s \Rightarrow p$ ok$'(T)$ s. Assume that p corr s holds. By assumption, the original system is a testing system. Thus, $\forall t$, p ok(t) s. If we choose a test set T, we know that, $\forall t \in T$, p ok(t) s. Therefore, p ok$'(T)$ s holds.

The set construction is interpreted as follows. A test consists of a number of trials of some sort, and success of the test as a whole depends on success of all the trials. In fact, this is the rule in testing practice, where a test engineer must run a program again and again on a variety of test data. Failure of any one run is enough to invalidate the program.

Definition. Given a testing system $< \mathcal{P}, \mathcal{S}, \mathcal{T}, \text{corr}, \text{ok} >$ a new system $< \mathcal{P}, \mathcal{S}, \mathcal{T}', \text{corr}, \text{ok}' >$ is called a *choice construction*, where $\mathcal{T}'$ is the set of subsets of $\mathcal{T}$, and where p ok$'(T)$ $s \Leftrightarrow \exists t(t \in T \wedge p$ ok(t) $s)$. (The reader may recall that T is a member of $\mathcal{T}'$ because $T \subseteq \mathcal{T}$.)

Theorem. $< \mathcal{P}, \mathcal{S}, \mathcal{T}', \text{corr}, \text{ok}' >$, a choice construction on a testing system $< \mathcal{P}, \mathcal{S}, \mathcal{T}, \text{corr}, \text{ok} >$, is itself a testing system.

Proof. Similar to the previous theorem, we need to show that p corr $s \Rightarrow p$ ok$'(T)$ s. Assume that p corr s. Thus, $\forall t$, p ok(t) s. If we pick a nonempty test set T, we know that $\exists t \in T$ such that p ok(t) s. Thus, we can write $\forall T(T \neq \phi \Rightarrow \exists t(t \in T \wedge p$ ok(t) $s))$, and $\forall T(T \neq \phi \Rightarrow p$ ok$'(T)$ $s)$. The empty test set ϕ must be excluded from ($\mathcal{T}$') because a testing system must include at least one test.

The choice construction models the situation in which a test engineer is given a number of alternative ways of testing the program, all of which are assumed to be equivalent.

Definition. A test method is a function $M : \mathcal{P} \times \mathcal{S} \rightarrow \mathcal{T}$.

That is, in the general case, a test method takes the specification S and an implementation program P and produces test cases. In practice, test methods are predominantly program dependent, specification dependent, or totally dependent on the expectations of customers, as explained below:

- **Program Dependent:** In this case, $T = M(P)$, that is, test cases are derived solely based on the source code of a system. This is called *white-box* testing. Here, a test method has complete knowledge of the internal details of a program. However, from the viewpoint of practical testing, a white-box method is not generally applied to an entire program. One applies such a method to small *units* of a given large system. A unit refers to a function, procedure, method, and so on. A white-box method allows a test engineer to use the details of a program unit. Effective use of a program unit requires a thorough understanding of the unit. Therefore, white-box test methods are used by programmers to test their own code.

- **Specification Dependent:** In this case, $T = M(S)$, that is, test cases are derived solely based on the specification of a system. This is called *black-box* testing. Here, a test method does not have access to the internal details of a program. Such a method uses information provided in the specification of a system. It is not unusual to use an entire specification in the generation of test cases because specifications are much smaller in size than their corresponding implementations. Black-box methods are generally used by the development team and an independent system test group.
- **Expectation Dependent:** In practice, customers may generate test cases based on their *expectations* from the product at the time of taking delivery of the system. These test cases may include *continuous-operation* tests, *usability* tests, and so on.

2.4.2 Power of Test Methods

A tester is concerned with the methods to produce test cases and to compare test methods so that they can identify an appropriate test method. Let M and N be two test methods. For M to be *at least as good as* N, we must have the situation that whenever N finds an error, so does M. In other words, whenever a program fails under a test case produced by method N, it will also fail under a test case produced by method M, with respect to the same specification. Therefore, $F_N \subseteq F_M$, where F_N and F_M are the sets of faults discovered by test sets produced by methods N and M, respectively.

Let T_M and T_N be the set of test cases produced by methods M and N, respectively. Then, we need to follow two ways to compare their fault detection power.

Case 1: $T_M \supseteq T_N$. In this case, it is clear that method M is at least as good as method N. This is because method M produces test cases which reveal all the faults revealed by test cases produced by method N. This case is depicted in Figure 2.3*a*.

Case 2: T_M and T_N overlap, but $T_M \not\supseteq T_N$. This case suggests that T_M does not totally contain T_N. To be able to compare their fault detection ability, we execute the program P under both sets of test cases, namely T_M and T_N. Let F_M and F_N be the sets of faults detected by test sets T_M and T_N, respectively. If $F_M \supseteq F_N$, then we say that method M is at least as good as method N. This situation is explained in Figure 2.3*b*.

2.5 ADEQUACY OF TESTING

Testing gives designers and programmers much confidence in a software component or a complete product if it passes their test cases. Assume that a set of test cases

Figure 2.3 Different ways of comparing power of test methods: (*a*) produces all test cases produced by another method; (*b*) test sets have common elements.

T has been designed to test a program P. We execute P with the test set T. If T reveals faults in P, then we modify the program in an attempt to fix those faults. At this stage, there may be a need to design some new test cases, because, for example, we may include a new procedure in the code. After modifying the code, we execute the program with the new test set. Thus, we execute the test-and-fix loop until no more faults are revealed by the updated test set. Now we face a dilemma as follows: Is P really fault free, or is T not good enough to reveal the remaining faults in P? From testing we cannot conclude that P is fault free, since, as Dijkstra observed, testing can reveal the presence of faults, but not their absence. Therefore, if P passes T, we need to know that T is "good enough" or, in other words, that T is an adequate set of tests. It is important to evaluate the adequacy of T because if T is found to be not adequate, then more test cases need to be designed, as illustrated in Figure 2.4. Adequacy of T means whether or not T thoroughly tests P.

Ideally, testing should be performed with an *adequate* test set T. Intuitively, the idea behind specifying a criterion for evaluating test adequacy is to know whether or not sufficient testing has been done. We will soon return to the idea of test adequacy. In the absence of test adequacy, developers will be forced to use ad hoc measures to decide when to stop testing. Some examples of ad hoc measures for stopping testing are as follows [13]:

- Stop when the allocated time for testing expires.
- Stop when it is time to release the product.
- Stop when all the test cases execute without revealing faults.

Figure 2.4 depicts two important notions concerning test design and evaluating test adequacy as follows:

Figure 2.4 Context of applying test adequacy.

- Adequacy of a test set T is evaluated *after* it is found that T reveals no more faults. One may argue: Why not design test cases to meet an adequacy criterion? However, it is important to design test cases independent of an adequacy criterion because the primary goal of testing is to locate errors, and, thus, test design should not be constrained by an adequacy criterion. An example of a test design criteria is as follows: Select test cases to execute all statements in a program at least once. However, the difficulty with such a test design criterion is that we may not be able to know whether every program statement can be executed. Thus, it is difficult to judge the adequacy of the test set selected thereby. Finally, since the goal of testing is to reveal faults, there is no point in evaluating the adequacy of the test set as long as faults are being revealed.
- An adequate test set T does not say anything about the correctness of a program. A common understanding of correctness is that we have found and fixed all faults in a program to make it "correct." However, in practice, it is not realistic—though very much desirable—to find and fix all faults in a program. Thus, on the one hand, an adequacy criterion may not try

to aim for program correctness. On the other hand, a fault-free program should not turn any arbitrary test set T into an adequate test.

The above two points tell us an important notion: that the adequacy of a test set be evaluated independent of test design processes for the programs under test. Intuitively, a test set T is said to be adequate if it covers all aspects of the actual computation performed by a program and all computations intended by its specification. Two practical methods for evaluating test adequacy are as follows:

- **Fault Seeding:** This method refers to implanting a certain number of faults in a program P and executing P with test set T. If T reveals k percent of the implanted faults, we assume that T has revealed only k percent of the original faults. If 100% of the implanted faults have been revealed by T, we feel more confident about the adequacy of T. A thorough discussion of fault seeding can be found in Chapter 13.
- **Program Mutation:** Given a program P, a *mutation* is a program obtained by making a small change to P. In the program mutation method, a series of mutations are obtained from P. Some of the mutations may contain faults and the rest are equivalent to P. A test set T is said to be adequate if it causes every faulty mutation to produce an unexpected outcome. A more thorough discussion of program mutation can be found in Chapter 3.

2.6 LIMITATIONS OF TESTING

Ideally, all programs should be correct, that is, there is no fault in a program. Due to the impractical nature of proving even small programs to be correct, customers and software developers rely on the efficacy of testing. In this section, we introduce two main limitations of testing:

- Testing means executing a program with a generally small, proper subset of the input domain of the program. A small, proper subset of the input domain is chosen because cost may not allow a much larger subset to be chosen, let alone the full input set. Testing with the full input set is known as exhaustive testing. Thus, the inherent need to test a program with a small subset of the input domain poses a fundamental limit on the efficacy of testing. The limit is in the form of our inability to extrapolate the correctness of results for a proper subset of the input domain to program correctness. In other words, even if a program passes a test set T, we cannot conclude that the program is correct.
- Once we have selected a subset of the input domain, we are faced with the problem of verifying the correctness of the program outputs for individual test input. That is, a program output is examined to determine if the program performed correctly on the test input. The mechanism which verifies the correctness of a program output is known as an *oracle*. The concept of an oracle is discussed in detail in Chapter 9. Determining the correctness

of a program output is not a trivial task. If either of the following two conditions hold, a program is considered *nontestable* [20]:

There does not exist an oracle.

It is too difficult to determine the correct output.

If there is no mechanism to verify the correctness of a program output or it takes an extraordinary amount of time to verify an output, there is not much to be gained by running the test.

2.7 SUMMARY

The ideal, abstract goal of testing is to reveal all faults in a software system without exhaustively testing the software. This idea is the basis of the concept of an ideal test developed by Goodenough and Gerhart [17]. An ideal test is supposed to be a small, proper subset of the entire input domain, and we should be able to extrapolate the results of an ideal test to program correctness. In other words, in an abstract sense, if a program passes all the tests in a carefully chosen test set, called an ideal test, we are in a position to claim that the program is correct.

Coupled with the concept of an ideal test is a test selection criterion which allows us to pick members of an ideal test. A test selection criterion is characterized in terms of reliability and validity. A reliable criterion is one which selects test cases such that a program either passes all tests or fails all tests. On the other hand, a valid criterion is one which selects at least one test set which fails in case the program contains a fault. If a criterion is both valid and reliable, then any test selected by the criterion is an ideal test. The theory has a few drawback. First, the concepts of reliability and validity have been defined with respect to one program and its entire input domain. Second, neither reliability nor validity is preserved throughout the debugging phase of software development.

Faults occur due to our inadequate understanding of all conditions that a program must deal with and our failure to realize that certain combinations of conditions require special treatments. Goodenough and Gerhart categorize faults into five categories: logic faults, requirement faults, design faults, construction faults, and performance faults.

Weyuker and Ostrand [18] tried to eliminate the drawbacks of the theory of Goodenough and Gerhart by proposing the concept of a uniformly ideal test. The concept is defined with respect to *all* programs designed to satisfy a specification, rather than just one program—hence the concept of "uniformity" over all program instances for a given specification. Further, the idea of uniformity was extended to test selection criteria in the form of a uniformly reliable and uniformly valid criterion. However, their theory too is impractical because a uniformly valid and uniformly reliable criterion selects the entire input domain of a program, thereby causing exhaustive testing. Next, the idea of an ideal test was extended to a proper subset of the input domain called a subdomain, and the concept of a revealing criterion was defined.

Though testing cannot settle the question of program correctness, different testing methods continue to be developed. For example, there are specification-based testing methods and code-based testing methods. It is important to develop a theory to compare the power of different testing methods. Gourlay [19] put forward a theory to compare the power of testing methods based on their fault detection abilities.

A software system undergoes multiple test–fix–retest cycles until, ideally, no more faults are revealed. Faults are fixed by modifying the code or adding new code to the system. At this stage there may be a need to design new test cases. When no more faults are revealed, we can conclude this way: either there is no fault in the program or the tests could not reveal the faults. Since we have no way to know the exact situation, it is useful to evaluate the adequacy of the test set. There is no need to evaluate the adequacy of tests so long as they reveal faults. Two practical ways of evaluating test adequacy are fault seeding and program mutation.

Finally, we discussed two limitations of testing. The first limitation of testing is that it cannot settle the question of program correctness. In other words, by testing a program with a proper subset of the input domain and observing no fault, we cannot conclude that there are no remaining faults in the program. The second limitation of testing is that in several instances we do not know the expected output of a program. If for some inputs the expected output of a program is not known or it cannot be determined within a reasonable amount of time, then the program is called nontestable [20].

LITERATURE REVIEW

Weyuker and Ostrand [18] have shown by examples how to construct revealing subdomains from source code. Their main example is the well-known triangle classification problem. The triangle classification problem is as follows. Let us consider three positive integers A, B, and C. The problem is to find whether the given integers represent the sides of an equilateral triangle, the sides of a scalene right triangle, and so on.

Weyuker [13] has introduced the notion of *program inference* to capture the notion of test data adequacy. Essentially, program inference refers to deriving a program from its specification and a sample of its input–output behavior. On the other hand, the testing process begins with a specification S and a program P and selects input–output pairs that characterize every aspect of the actual computations performed by the program and the intended computations performed by the specification. Thus, program testing and program inference are thought of as inverse processes. A test set T is said to be adequate if T contains sufficient data to infer the computations defined by both S and P. However, Weyuker [13] explains that such an adequacy criterion is not pragmatically usable. Rather, the criterion can at best be used as a guide. By considering the difficulty in using the criterion, Weyuker defines two weaker adequacy criterion, namely *program adequate* and *specification adequate*. A test set T is said to be program adequate if it contains sufficient data to infer the computations defined by P. Similarly, the test set T is

said to be specification adequate if it contains sufficient data to infer the computations defined by S. It is suggested that depending upon how test data are selected, one of the two criteria can be eased out. For example, if T is derived from S, then it is useful to evaluate if T is program adequate. Since T is selected from S, T is expected to contain sufficient data to infer the computations defined by S, and there is no need to evaluate T's specification adequacy. Similarly, if T is derived from P, it is useful to evaluate if T is specification adequate.

The students are encouraged to read the article by Stuart H. Zweben and John S. Gourlay entitled "On the Adequacy of Weyuker's Test Data Adequacy Axioms" [15] The authors raise the issue of what makes an axiomatic system as well as what constitutes a proper axiom. Weyuker responds to the criticism at the end of the article. Those students have never seen such a professional interchange; this is worth reading for this aspect alone. This article must be read along with the article by Elaine Weyuker entitled "Axiomatizing Software Test Data Adequacy" [12].

Martin David and Elaine Weyuker [9] present an interesting notion of *distance between programs* to study the concept of test data adequacy. Specifically, they equate adequacy with the capability of a test set to be able to successfully distinguish a program being tested from all programs that are sufficiently close to it and differ in input–output behavior from the given program.

Weyuker [12, 21] proposed a set of properties to evaluate test data adequacy criteria. Some examples of adequacy criteria are to (i) ensure coverage of all branches in the program being tested and (ii) ensure that boundary values of all input data have been selected for the program under test. Parrish and Zweben [11] formalized those properties and identified dependencies within the set. They formalized the adequacy properties with respect to criteria that do not make use of the specification of the program under test.

Frankl and Weyuker [10] compared the relative fault-detecting ability of a number of structural testing techniques, namely, data flow testing, mutation testing, and a condition coverage technique, to branch testing. They showed that the former three techniques are better than branch testing according to two probabilistic measures.

A good survey on test adequacy is presented in an article by Hong Zhu, Patrick A. V. Hall, and John H. R. May entitled "Software Unit Test Coverage and Adequacy" [14]. In this article, various types of software test adequacy criteria proposed in the literature are surveyed followed by a summary of methods for comparison and assessment of adequacy criteria.

REFERENCES

1. R. Gupta, M. J. Harrold, and M. L. Soffa. *An Approach to Regression Testing Using Slicing*. Paper presented at the IEEE-CS International Conference on Software Maintenance, Orlando, FL, November 1992, pp. 299–308.
2. G. Rothermel and M. Harrold. Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, August 1996, pp. 529–551.

3. V. R. Basili and R. W. Selby. Comparing the Effectiveness of Software Testing. *IEEE Transactions on Software Engineering*, December 1987, pp. 1278–1296.
4. W. E. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, July 1982, pp. 371–379.
5. D. S. Rosenblum and E. J. Weyuker. Using Coverage Information to Predict the Cost-effectiveness of Regression Testing Strategies. *IEEE Transactions on Software Engineering*, March 1997, pp. 146–156.
6. L. Baresi and M. Young. Test Oracles, Technical Report CIS-TR-01–02. University of Oregon, Department of Computer and Information Science, Eugene, OR, August 2002, pp. 1–55.
7. Q. Xie and A. M. Memon. Designing and Comparing Automated Test Oracles for Gui Based Software Applications. *ACM Transactions on Software Engineering amd Methodology*, February 2007, pp. 1–36.
8. G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing Test Cases for Regression Testing. *IEEE Transactions on Software Engineering*, October 2001, pp. 929–948.
9. M. Davis and E. J. Weyuker. Metric Space-Based Test-Data Adequacy Criteria. *Computer Journal*, January 1988, pp. 17–24.
10. P. G. Frankl and E. J. Weyuker. Provable Improvements on Branch Testing. *IEEE Transactions on Software Engineering*, October 1993, pp. 962–975.
11. A. Parrish and S. H. Zweben. Analysis and Refinement of Software Test Data Adequacy Properties. *IEEE Transactions on Software Engineering*, June 1991, pp. 565–581.
12. E. J. Weyuker. Axiomatizing Software Test Data Adequacy. *IEEE Transactions on Software Engineering*, December 1986, pp. 1128–1138.
13. E. J. Weyuker. Assessing Test Data Adequacy through Program Inference. *ACM Transactions on Programming Languages and Systems*, October 1983, pp. 641–655.
14. H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, December 1997, pp. 366–427.
15. S. H. Zweben and J. S. Gourlay. On the Adequacy of Weyuker's Test Data Adequacy Axioms. *IEEE Transactions on Software Engineering*, April 1989, pp. 496–500.
16. E. W. Dijkstra. Notes on Structured Programming. In *Structured Programming*, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. Academic, New York, 1972, pp. 1–81.
17. J. B. Goodenough and S. L. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, June 1975, pp. 26–37.
18. E. J. Weyuker and T. J. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering*, May 1980, pp. 236–246.
19. J. S. Gourlay. A Mathematical Framework for the Investigation of Testing. *IEEE Transactions on Software Engineering*, November 1983, pp. 686–709.
20. E. J. Weyuker. On Testing Non-Testable Programs. *Computer Journal*, Vol. 25, No. 4, 1982, pp. 465–470.
21. E. J. Weyuker. The Evaluation of Program-Based Software Test Data Adequacy Criteria. *Communications of the ACM*, June 1988, pp. 668–675.

Exercises

1. Explain the concept of an ideal test.
2. Explain the concept of a selection criterion in test design.
3. Explain the concepts of a valid and reliable criterion.
4. Explain five kinds of program faults.
5. What are the drawbacks of Goodenough and Gerhart's theory of program testing?

6. Explain the concepts of a uniformly ideal test as well as the concepts of uniformly valid and uniformly reliable criteria.
7. Explain how two test methods can be compared.
8. Explain the need for evaluating test adequacy.
9. Explain two practical methods for assessing test data adequacy.
10. Explain the concept of a nontestable program.

CHAPTER 3

Unit Testing

Knowledge is of no value unless you put it into practice.
— *Anton Chekhov*

3.1 CONCEPT OF UNIT TESTING

In this chapter we consider the first level of testing, that is, unit testing. Unit testing refers to testing program units in isolation. However, there is no consensus on the definition of a unit. Some examples of commonly understood units are functions, procedures, or methods. Even a class in an object-oriented programming language can be considered as a program unit. Syntactically, a program unit is a piece of code, such as a function or method of class, that is invoked from outside the unit and that can invoke other program units. Moreover, a program unit is assumed to implement a well-defined function providing a certain level of abstraction to the implementation of higher level functions. The function performed by a program unit may not have a direct association with a system-level function. Thus, a program unit may be viewed as a piece of code implementing a "low"-level function. In this chapter, we use the terms unit and module interchangeably.

Now, given that a program unit implements a function, it is only natural to test the unit before it is integrated with other units. Thus, a program unit is tested in isolation, that is, in a stand-alone manner. There are two reasons for testing a unit in a stand-alone manner. First, errors found during testing can be attributed to a specific unit so that it can be easily fixed. Moreover, unit testing removes dependencies on other program units. Second, during unit testing it is desirable to verify that each distinct execution of a program unit produces the expected result. In terms of code details, a distinct execution refers to a distinct path in the unit. Ideally, all possible—or as much as possible—distinct executions are to be considered during unit testing. This requires careful selection of input data for each distinct execution. A programmer has direct access to the input vector of the unit by executing a program unit in isolation. This direct access makes it easier to execute as many distinct paths as desirable or possible. If multiple units are put together for

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

testing, then a programmer needs to generate test input with indirect relationship with the input vectors of several units under test. The said indirect relationship makes it difficult to control the execution of distinct paths in a chosen unit.

Unit testing has a limited scope. A programmer will need to verify whether or not a code works correctly by performing unit-level testing. Intuitively, a programmer needs to test a unit as follows:

- Execute every line of code. This is desirable because the programmer needs to know what happens when a line of code is executed. In the absence of such basic observations, surprises at a later stage can be expensive.
- Execute every predicate in the unit to evaluate them to true and false separately.
- Observe that the unit performs its intended function and ensure that it contains no known errors.

In spite of the above tests, there is no guarantee that a satisfactorily tested unit is functionally correct from a systemwide perspective. Not everything pertinent to a unit can be tested in isolation because of the limitations of testing in isolation. This means that some errors in a program unit can only be found later, when the unit is integrated with other units in the integration testing and system testing phases. Even though it is not possible to find all errors in a program unit in isolation, it is still necessary to ensure that a unit performs satisfactorily before it is used by other program units. It serves no purpose to integrate an erroneous unit with other units for the following reasons: (i) many of the subsequent tests will be a waste of resources and (ii) finding the root causes of failures in an integrated system is more resource consuming.

Unit testing is performed by the programmer who writes the program unit because the programmer is intimately familiar with the internal details of the unit. The objective for the programmer is to be satisfied that the unit works as expected. Since a programmer is supposed to construct a unit with no errors in it, a unit test is performed by him or her to their satisfaction in the beginning and to the satisfaction of other programmers when the unit is integrated with other units. This means that all programmers are accountable for the quality of their own work, which may include both new code and modifications to the existing code. The idea here is to push the quality concept down to the lowest level of the organization and empower each programmer to be responsible for his or her own quality. Therefore, it is in the best interest of the programmer to take preventive actions to minimize the number of defects in the code. The defects found during unit testing are internal to the software development group and are not reported up the personnel hierarchy to be counted in quality measurement metrics. The source code of a unit is not used for interfacing by other group members until the programmer completes unit testing and checks in the unit to the version control system.

Unit testing is conducted in two complementary phases:

- Static unit testing
- Dynamic unit testing

In static unit testing, a programmer does not execute the unit; instead, the code is examined over all possible behaviors that might arise during run time. Static unit testing is also known as non-execution-based unit testing, whereas dynamic unit testing is execution based. In static unit testing, the code of each unit is validated against requirements of the unit by reviewing the code. During the review process, potential issues are identified and resolved. For example, in the C programming language the two program-halting instructions are abort() and exit(). While the two are closely related, they have different effects as explained below:

- **Abort():** This means abnormal program termination. By default, a call to abort() results in a run time diagnostic and program self-destruction. The program destruction may or may not flush and close opened files or remove temporary files, depending on the implementation.
- **Exit():** This means graceful program termination. That is, the exit() call closes the opened files and returns a status code to the execution environment.

Whether to use abort() or exit() depends on the context that can be easily detected and resolved during static unit testing. More issues caught earlier lead to fewer errors being identified in the dynamic test phase and result in fewer defects in shipped products. Moreover, performing static tests is less expensive than performing dynamic tests. Code review is one component of the defect minimization process and can help detect problems that are common to software development. After a round of code review, dynamic unit testing is conducted. In dynamic unit testing, a program unit is actually executed and its outcomes are observed. Dynamic unit testing means testing the code by actually running it. It may be noted that static unit testing is not an alternative to dynamic unit testing. A programmer performs both kinds of tests. In practice, partial dynamic unit testing is performed concurrently with static unit testing. If the entire dynamic unit testing has been performed and a static unit testing identifies significant problems, the dynamic unit testing must be repeated. As a result of this repetition, the development schedule may be affected. To minimize the probability of such an event, it is required that static unit testing be performed prior to the final dynamic unit testing.

3.2 STATIC UNIT TESTING

Static unit testing is conducted as a part of a larger philosophical belief that a software product should undergo a phase of inspection and correction at each milestone in its life cycle. At a certain milestone, the product need not be in its final form. For example, completion of coding is a milestone, even though coding of all the units may not make the desired product. After coding, the next milestone is testing all or a substantial number of units forming the major components of the product. Thus, before units are individually tested by actually executing them, those are subject to usual review and correction as it is commonly understood. The idea behind review is to find the defects as close to their points of origin as possible so

that those defects are eliminated with less effort, and the interim product contains fewer defects before the next task is undertaken.

In static unit testing, code is reviewed by applying techniques commonly known as *inspection* and *walkthrough*. The original definition of inspection was coined by Michael Fagan [1] and that of walkthrough by Edward Yourdon [2]:

- **Inspection:** It is a step-by-step peer group review of a work product, with each step checked against predetermined criteria.
- **Walkthrough:** It is a review where the author leads the team through a manual or simulated execution of the product using predefined scenarios.

Regardless of whether a review is called an inspection or a walkthrough, it is a systematic approach to examining source code in detail. The goal of such an exercise is to assess the quality of the software in question, *not* the quality of the process used to develop the product [3]. Reviews of this type are characterized by significant preparation by groups of designers and programmers with varying degree of interest in the software development project. Code examination can be time consuming. Moreover, no examination process is perfect. Examiners may take shortcuts, may not have adequate understanding of the product, and may accept a product which should not be accepted. Nonetheless, a well-designed code review process can find faults that may be missed by execution-based testing. The key to the success of code review is to divide and conquer, that is, having an examiner inspect small parts of the unit in isolation, while making sure of the following: (i) nothing is overlooked and (ii) the correctness of all examined parts of the module implies the correctness of the whole module. The decomposition of the review into discrete steps must assure that each step is simple enough that it can be carried out without detailed knowledge of the others.

The objective of code review is to review the code, not to evaluate the author of the code. A clash may occur between the author of the code and the reviewers, and this may make the meetings unproductive. Therefore, code review must be planned and managed in a professional manner. There is a need for mutual respect, openness, trust, and sharing of expertise in the group. The general guidelines for performing code review consists of six steps as outlined in Figure 3.1: readiness, preparation, examination, rework, validation, and exit. The input to the readiness step is the criteria that must be satisfied before the start of the code review process, and the process produces two types of documents, a change request (CR) and a report. These steps and documents are explained in the following.

Step 1: Readiness The author of the unit ensures that the unit under test is ready for review. A unit is said to be ready if it satisfies the following criteria:

- **Completeness:** All the code relating to the unit to be reviewed must be available. This is because the reviewers are going to read the code and try to understand it. It is unproductive to review partially written code or code that is going to be significantly modified by the programmer.

Figure 3.1 Steps in the code review process.

- **Minimal Functionality:** The code must compile and link. Moreover, the code must have been tested to some extent to make sure that it performs its basic functionalities.
- **Readability:** Since code review involves actual reading of code by other programmers, it is essential that the code is highly readable. Some code characteristics that enhance readability are proper formatting, using meaningful identifier names, straightforward use of programming language constructs, and an appropriate level of abstraction using function calls. In the absence of readability, the reviewers are likely to be discouraged from performing the task effectively.
- **Complexity:** There is no need to schedule a group meeting to review straightforward code which can be easily reviewed by the programmer. The code to be reviewed must be of sufficient complexity to warrant group review. Here, complexity is a composite term referring to the number of conditional statements in the code, the number of input data elements of the unit, the number of output data elements produced by the unit, real-time processing of the code, and the number of other units with which the code communicates.
- **Requirements and Design Documents:** The latest approved version of the low-level design specification or other appropriate descriptions

TABLE 3.1 Hierarchy of System Documents

Requirement: High-level marketing or product proposal.
Functional specification: Software engineering response to the marketing proposal.
High-level design: Overall system architecture.
Low-level design: Detailed specification of the modules within the architecture.
Programming: Coding of the modules.

of program requirements (see Table 3.1) should be available. These documents help the reviewers in verifying whether or not the code under review implements the expected functionalities. If the low-level design document is available, it helps the reviewers in assessing whether or not the code appropriately implements the design.

All the people involved in the review process are informed of the group review meeting schedule two or three days before the meeting. They are also given a copy of the work package for their perusal. Reviews are conducted in bursts of 1–2 hours. Longer meetings are less and less productive because of the limited attention span of human beings. The rate of code review is restricted to about 125 lines of code (in a high-level language) per hour. Reviewing complex code at a higher rate will result in just glossing over the code, thereby defeating the fundamental purpose of code review. The composition of the review group involves a number of people with different roles. These roles are explained as follows:

- **Moderator:** A review meeting is chaired by the moderator. The moderator is a trained individual who guides the pace of the review process. The moderator selects the reviewers and schedules the review meetings. Myers suggests that the moderator be a member of a group from an unrelated project to preserve objectivity [4].
- **Author:** This is the person who has written the code to be reviewed.
- **Presenter:** A presenter is someone other than the author of the code. The presenter reads the code beforehand to understand it. It is the presenter who presents the author's code in the review meeting for the following reasons: (i) an additional software developer will understand the work within the software organization; (ii) if the original programmer leaves the company with a short notice, at least one other programmer in the company knows what is being done; and (iii) the original programmer will have a good feeling about his or her work, if someone else appreciates their work. Usually, the presenter appreciates the author's work.
- **Recordkeeper:** The recordkeeper documents the problems found during the review process and the follow-up actions suggested. The person should be different than the author and the moderator.
- **Reviewers:** These are experts in the subject area of the code under review. The group size depends on the content of the material under

review. As a rule of thumb, the group size is between 3 and 7. Usually this group does not have manager to whom the author reports. This is because it is the author's ongoing work that is under review, and neither a completed work nor the author himself is being reviewed.

- **Observers:** These are people who want to learn about the code under review. These people do not participate in the review process but are simply passive observers.

Step 2: Preparation Before the meeting, each reviewer carefully reviews the work package. It is expected that the reviewers read the code and understand its organization and operation before the review meeting. Each reviewer develops the following:

- **List of Questions:** A reviewer prepares a list of questions to be asked, if needed, of the author to clarify issues arising from his or her reading. A general guideline of what to examine while reading the code is outlined in Table 3.2.
- **Potential CR:** A reviewer may make a formal request to make a change. These are called change requests rather than defect reports. At this stage, since the programmer has not yet made the code public, it is more appropriate to make suggestions to the author to make changes, rather than report a defect. Though CRs focus on defects in the code, these reports are not included in defect statistics related to the product.
- **Suggested Improvement Opportunities:** The reviewers may suggest how to fix the problems, if there are any, in the code under review. Since reviewers are experts in the subject area of the code, it is not unusual for them to make suggestions for improvements.

Step 3: Examination The examination process consists of the following activities:

- The author makes a presentation of the procedural logic used in the code, the paths denoting major computations, and the dependency of the unit under review on other units.
- The presenter reads the code line by line. The reviewers may raise questions if the code is seen to have defects. However, problems are not resolved in the meeting. The reviewers may make general suggestions on how to fix the defects, but it is up to the author of the code to take corrective measures after the meeting ends.
- The recordkeeper documents the change requests and the suggestions for fixing the problems, if there are any. A CR includes the following details:
  1. Give a brief description of the issue or action item.
  2. Assign a priority level (major or minor) to a CR.
  3. Assign a person to follow up the issue. Since a CR documents a potential problem, there is a need for interaction between the author

TABLE 3.2 Code Review Checklist

1. Does the code do what has been specified in the design specification?
2. Does the procedure used in the module solve the problem correctly?
3. Does a software module duplicate another existing module which could be reused?
4. If library modules are being used, are the right libraries and the right versions of the libraries being used?
5. Does each module have a single entry point and a single exit point? Multiple exit and entry point programs are harder to test.
6. Is the cyclomatic complexity of the module more than 10? If yes, then it is extremely difficult to adequately test the module.
7. Can each atomic function be reviewed and understood in 10–15 minutes? If not, it is considered to be too complex.
8. Have naming conventions been followed for all identifiers, such as pointers, indices, variables, arrays, and constants? It is important to adhere to coding standards to ease the introduction of a new contributor (programmer) to the development of a system.
9. Has the code been adequately commented upon?
10. Have all the variables and constants been correctly initialized? Have correct types and scopes been checked?
11. Are the global or shared variables, if there are any, carefully controlled?
12. Are there data values hard coded in the program? Rather, these should be declared as variables.
13. Are the pointers being used correctly?
14. Are the dynamically acquired memory blocks deallocated after use?
15. Does the module terminate abnormally? Will the module eventually terminate?
16. Is there a possibility of an infinite loop, a loop that never executes, or a loop with a premature exit?
17. Have all the files been opened for use and closed at termination?
18. Are there computations using variables with inconsistent data types? Is overflow or underflow a possibility?
19. Are error codes and condition messages produced by accessing a common table of messages? Each error code should have a meaning, and all of the meanings should be available at one place in a table rather than scattered all over the program code.
20. Is the code portable? The source code is likely to execute on multiple processor architectures and on different operating systems over its lifetime. It must be implemented in a manner that does not preclude this kind of a variety of execution environments.
21. Is the code efficient? In general, clarity, readability, or correctness should not be sacrificed for efficiency. Code review is intended to detect implementation choices that have adverse effects on system performance.

of the code and one of the reviewers, possibly the reviewer who made the CR.

4. Set a deadline for addressing a CR.

- The moderator ensures that the meeting remains focused on the review process. The moderator makes sure that the meeting makes progress at a certain rate so that the objective of the meeting is achieved.

- At the end of the meeting, a decision is taken regarding whether or not to call another meeting to further review the code. If the review process leads to extensive rework of the code or critical issues are identified in the process, then another meeting is generally convened. Otherwise, a second meeting is not scheduled, and the author is given the responsibility of fixing the CRs.

Step 4: **Rework** At the end of the meeting, the recordkeeper produces a summary of the meeting that includes the following information:

- A list of all the CRs, the dates by which those will be fixed, and the names of the persons responsible for validating the CRs
- A list of improvement opportunities
- The minutes of the meeting (optional)

A copy of the report is distributed to all the members of the review group. After the meeting, the author works on the CRs to fix the problems. The author documents the improvements made to the code in the CRs. The author makes an attempt to address the issues within the agreed-upon time frame using the prevailing coding conventions [5].

Step 5: **Validation** The CRs are independently validated by the moderator or another person designated for this purpose. The validation process involves checking the modified code as documented in the CRs and ensuring that the suggested improvements have been implemented correctly. The revised and final version of the outcome of the review meeting is distributed to all the group members.

Step 6: **Exit** Summarizing the review process, it is said to be complete if all of the following actions have been taken:

- Every line of code in the unit has been inspected.
- If too many defects are found in a module, the module is once again reviewed after corrections are applied by the author. As a rule of thumb, if more than 5% of the total lines of code are thought to be contentious, then a second review is scheduled.
- The author and the reviewers reach a consensus that when corrections have been applied the code will be potentially free of defects.
- All the CRs are documented and validated by the moderator or someone else. The author's follow-up actions are documented.
- A summary report of the meeting including the CRs is distributed to all the members of the review group.

The effectiveness of static testing is limited by the ability of a reviewer to find defects in code by visual means. However, if occurrences of defects depend on some actual values of variables, then it is a difficult task to identify those defects by visual means. Therefore, a unit must be executed to observe its behaviors in response to a variety of inputs. Finally, whatever may be the effectiveness of static tests, one cannot feel confident without actually running the code.

Code Review Metrics It is important to collect measurement data pertinent to a review process, so that the review process can be evaluated, made visible to the upper management as a testing strategy, and improved to be more effective. Moreover, collecting metrics during code review facilitates estimation of review time and resources for future projects. Thus, code review is a viable testing strategy that can be effectively used to improve the quality of products at an early stage. The following metrics can be collected from a code review:

- Number of lines of code (LOC) reviewed per hour
- Number of CRs generated per thousand lines of code (KLOC)
- Number of CRs generated per hour
- Total number of CRs generated per project
- Total number of hours spent on code review per project

3.3 DEFECT PREVENTION

It is in the best interest of the programmers in particular and the company in general to reduce the number of CRs generated during code review. This is because CRs are indications of potential problems in the code, and those problems must be resolved before different program units are integrated. Addressing CRs means spending more resources and potentially delaying the project. Therefore, it is essential to adopt the concept of defect prevention during code development. In practice, defects are inadvertently introduced by programmers. Those accidents can be reduced by taking preventive measures. It is useful to develop a set of guidelines to construct code for defect minimization as explained in the following. These guidelines focus on incorporating suitable mechanisms into the code:

- Build internal diagnostic tools, also known as *instrumentation code*, into the units. Instrumentation codes are useful in providing information about the internal states of the units. These codes allow programmers to realize built-in tracking and tracing mechanisms. Instrumentation plays a passive role in dynamic unit testing. The role is passive in the sense of observing and recording the internal behavior without actively testing a unit.
- Use standard controls to detect possible occurrences of error conditions. Some examples of error detection in the code are divides by zero and array index out of bounds.
- Ensure that code exists for all return values, some of which may be invalid. Appropriate follow-up actions need to be taken to handle invalid return values.
- Ensure that counter data fields and buffer overflow and underflow are appropriately handled.
- Provide error messages and help texts from a common source so that changes in the text do not cause inconsistency. Good error messages

identify the root causes of the problems and help users in resolving the problems [7].

- Validate input data, such as the arguments, passed to a function.
- Use assertions to detect impossible conditions, undefined uses of data, and undesirable program behavior. An assertion is a Boolean statement which should never be false or can be false only if an error has occurred. In other words, an assertion is a check on a condition which is assumed to be true, but it can cause a problem if it not true. Assertion should be routinely used to perform the following kinds of checks:
  1. Ensure that preconditions are satisfied before beginning to execute a unit. A precondition is a Boolean function on the states of a unit specifying our expectation of the state prior to initiating an activity in the code.
  2. Ensure that the expected postconditions are true while exiting from the unit. A postcondition is a Boolean function on the state of a unit specifying our expectation of the state after an activity has been completed. The postconditions may include an invariance.
  3. Ensure that the invariants hold. That is, check invariant states—conditions which are expected not to change during the execution of a piece of code.
- Leave assertions in the code. You may deactivate them in the released version of code in order to improve the operational performance of the system.
- Fully document the assertions that appear to be unclear.
- After every major computation, reverse-compute the input(s) from the results in the code itself. Then compare the outcome with the actual inputs for correctness. For example, suppose that a piece of code computes the square root of a positive number. Then square the output value and compare the result with the input. It may be needed to tolerate a margin of error in the comparison process.
- In systems involving message passing, buffer management is an important internal activity. Incoming messages are stored in an already allocated buffer. It is useful to generate an event indicating low buffer availability before the system runs out of buffer. Develop a routine to continually monitor the availability of buffer after every use, calculate the remaining space available in the buffer, and call an error handling routine if the amount of available buffer space is too low.
- Develop a timer routine which counts down from a preset time until it either hits zero or is reset. If the software is caught in an infinite loop, the timer will expire and an exception handler routine can be invoked.
- Include a loop counter within each loop. If the loop is ever executed less than the minimum possible number of times or more than the maximum possible number of times, then invoke an exception handler routine.

- Define a variable to indicate the branch of decision logic that will be taken. Check this value after the decision has been made and the right branch has supposedly been taken. If the value of the variable has not been preset, there is probably a fall-through condition in the logic.

3.4 DYNAMIC UNIT TESTING

Execution-based unit testing is referred to as dynamic unit testing. In this testing, a program unit is actually executed in isolation, as we commonly understand it. However, this execution differs from ordinary execution in the following way:

1. A unit under test is taken out of its actual execution environment.
2. The actual execution environment is emulated by writing more code (explained later in this section) so that the unit and the emulated environment can be compiled together.
3. The above compiled aggregate is executed with selected inputs. The outcome of such an execution is collected in a variety of ways, such as straightforward observation on a screen, logging on files, and software instrumentation of the code to reveal run time behavior. The result is compared with the expected outcome. Any difference between the actual and expected outcome implies a failure and the fault is in the code.

An environment for dynamic unit testing is created by emulating the context of the unit under test, as shown in Figure 3.2. The context of a unit test consists of two parts: (i) a caller of the unit and (ii) all the units called by the unit. The environment of a unit is emulated because the unit is to be tested in isolation and the emulating environment must be a simple one so that any fault found as a result of running the unit can be solely attributed to the unit under test. The caller unit is known as a *test driver*, and all the emulations of the units called by the unit under test are called *stubs*. The test driver and the stubs are together called *scaffolding*. The functions of a test driver and a stub are explained as follows:

- **Test Driver:** A test driver is a program that invokes the unit under test. The unit under test executes with input values received from the driver and, upon termination, returns a value to the driver. The driver compares the actual outcome, that is, the actual value returned by the unit under test, with the expected outcome from the unit and reports the ensuing test result. The test driver functions as the *main* unit in the execution process. The driver not only facilitates compilation, but also provides input data to the unit under test in the expected format.
- **Stubs:** A stub is a "dummy subprogram" that replaces a unit that is called by the unit under test. Stubs replace the units called by the unit under test. A stub performs two tasks. First, it shows an evidence that the stub was,

Figure 3.2 Dynamic unit test environment.

in fact, called. Such evidence can be shown by merely printing a message. Second, the stub returns a precomputed value to the caller so that the unit under test can continue its execution.

The driver and the stubs are never discarded after the unit test is completed. Instead, those are reused in the future in regression testing of the unit if there is such a need. For each unit, there should be one dedicated test driver and several stubs as required. If just one test driver is developed to test multiple units, the driver will be a complicated one. Any modification to the driver to accommodate changes in one of the units under test may have side effects in testing the other units. Similarly, the test driver should not depend on the external input data files but, instead, should have its own segregated set of input data. The separate input data file approach becomes a very compelling choice for large amounts of test input data. For example, if hundreds of input test data elements are required to test more than one unit, then it is better to create a separate input test data file rather than to include the same set of input test data in each test driver designed to test the unit.

The test driver should have the capability to automatically determine the success or failure of the unit under test for each input test data. If appropriate, the driver should also check for memory leaks and problems in allocation and deallocation of memory. If the module opens and closes files, the test driver should check that these files are left in the expected open or closed state after each test. The test driver can be designed to check the data values of the internal variable that

normally would not be available for checking at integration, system, or acceptance testing levels.

The test driver and stubs are tightly coupled with the unit under test and should accompany the unit throughout its life cycle. A test driver and the stubs for a unit should be reusable and maintainable. Every time a unit is modified, the programmer should check whether or not to modify the corresponding driver and stubs. Whenever a new fault is detected in the unit, the corresponding test driver should be updated with a new set of input data to detect that fault and similar faults in the future. If the unit is expected to run on different platforms, the test driver and stubs should also be built to test the unit on new platforms. Finally, the test driver and stubs should be reviewed, cross-referenced with the unit for which these are written, and checked in to the version control system as a product along with the unit.

The low-level design document provides guidance for the selection of input test data that are likely to uncover faults. Selection of test data is broadly based on the following techniques:

- **Control Flow Testing:** The following is an outline of control flow testing: (i) draw a control flow graph from a program unit; (ii) select a few control flow testing criteria; (iii) identify paths in the control flow graph to satisfy the selection criteria; (iv) derive path predicate expressions from the selected paths; and (v) by solving the path predicate expression for a path, generate values of the inputs to the program unit that are considered as a test case to exercise the corresponding path. A thorough discussion of control flow testing is given in Chapter 4.
- **Data Flow Testing:** The following is an outline of data flow testing: (i) draw a data flow graph from a program unit; (ii) select a few data flow testing criteria; (iii) identify paths in the data flow graph to satisfy the selection criteria; (iv) derive path predicate expressions from the selected paths; and (v) by solving the path predicate expression, generate values of the inputs to the program unit that are considered as a test case to exercise the corresponding path. Chapter 5 discusses data flow testing in greater detail.
- **Domain Testing:** In control flow and data flow testing, no specific types of faults are explicitly considered for detection. However, domain testing takes a new approach to fault detection. In this approach, a category of faults called *domain errors* are defined and then test data are selected to catch those faults. It is discussed in detail in Chapter 6.
- **Functional Program Testing:** In functional program testing one performs the following steps: (i) identify the input and output domains of a program; (ii) for a given input domain, select some *special* values and compute the expected outcome; (iii) for a given output domain, select some *special* values and compute the input values that will cause the unit to produce those output values; and (iv) consider various combinations of the input values chosen above. Chapter 9 discusses functional testing.

3.5 MUTATION TESTING

Mutation testing has a rich and long history. It can be traced back to the late 1970s [8–10]. Mutation testing was originally proposed by Dick Lipton, and the article by DeMillo, Lipton, and Sayward [9] is generally cited as the seminal reference. Mutation testing is a technique that focuses on measuring the adequacy of test data (or test cases). The original intention behind mutation testing was to expose and locate weaknesses in test cases. Thus, mutation testing is a way to measure the quality of test cases, and the actual testing of program units is an added benefit. Mutation testing is not a testing strategy like control flow or data flow testing. It should be used to supplement traditional unit testing techniques.

A mutation of a program is a modification of the program created by introducing a single, small, legal syntactic change in the code. A modified program so obtained is called a *mutant*. The term mutant has been borrowed from biology. Some of these mutants are equivalent to the original program, whereas others are faulty. A mutant is said to be *killed* when the execution of a test case causes it to fail and the mutant is considered to be *dead*.

Some mutants are *equivalent* to the given program, that is, such mutants always produce the same output as the original program. In the real world, large programs are generally faulty, and test cases too contain faults. The result of executing a mutant may be different from the expected result, but a test suite does not detect the failure because it does not have the right test case. In this scenario the mutant is called *killable* or *stubborn*, that is, the existing set of test cases is insufficient to kill it. A *mutation score* for a set of test cases is the percentage of nonequivalent mutants killed by the test suite. The test suite is said to be *mutation adequate* if its mutation score is 100%. Mutation analysis is a two-step process:

1. The adequacy of an existing test suite is determined to distinguish the given program from its mutants. A given test suite may not be adequate to distinguish all the nonequivalent mutants. As explained above, those nonequivalent mutants that could not be identified by the given test suite are called stubborn mutants.
2. New test cases are added to the existing test suite to kill the stubborn mutants. The test suite enhancement process iterates until the test suite has reached a desired level of mutation score.

Let us consider the following program *P* that finds rank corresponding to the first time the maximum value appears in the array. For simplicity, we assume that the program *P* reads three input arguments and prints the message accordingly:

```
1. main(argc,argv)
2. int argc, r, i;
3. char *argv[];
4. { r = 1;
5.    for i = 2 to 3 do
6.      if (atoi(argv[i]) > atoi(argv[r])) r = i;
```

```
7. printf("Value of the rank is %d \n", r);
8. exit(0); }
```

Now let us assume that we have the following test suite that tests the program P:

Test case 1:

Input: 1 2 3

Output: Value of the rank is 3

Test case 2:

Input: 1 2 1

Output: Values of the rank is 2

Test case 3:

Input: 3 1 2

Output: Value of the rank is 1

Now, let us mutate the program P. We can start with the following changes:

Mutant 1: Change line 5 to

```
for i = 1 to 3 do
```

Mutant 2: Change line 6 to

```
if (i > atoi(argv[r])) r = i;
```

Mutant 3: Change line 6 to

```
if (atoi(argv[i]) >= atoi(argv[r])) r = i;
```

Mutant 4: Change line 6 to

```
if (atoi(argv[r]) > atoi(argv[r])) r = i;
```

If we run the modified programs against the test suite, we will get the following results:

Mutants 1 and 3: The programs will completely pass the test suite. In other words, mutants 1 and 3 are not killed.

Mutant 2: The program will fail test case 2.

Mutant 4: The program will fail test case 1 and test case 2.

If we calculate the mutation score, we see that we created four mutants, and two of them were killed. This tells us that the mutation score is 50%, assuming that mutants 1 and 3 are nonequivalent.

The score is found to be low. It is low because we assumed that mutants 1 and 3 are nonequivalent to the original program. We have to show that either mutants

1 and 3 are equivalent mutants or those are killable. If those are killable, we need to add new test cases to kill these two mutants. First, let us analyze mutant 1 in order to derive a "killer" test. The difference between P and mutant 1 is the starting point. Mutant 1 starts with $i = 1$, whereas P starts with $i = 2$. There is no impact on the result r. Therefore, we conclude that mutant 1 is an equivalent mutant. Second, we add a fourth test case as follows:

Test case 4:

Input: 2 2 1

Then program P will produce the output "Value of the rank is 1" and mutant 3 will produce the output "Value of the rank is 2." Thus, this test data kills mutant 3, which give us a mutation score of 100%.

In order to use the mutation testing technique to build a robust test suite, the test engineer needs to follow the steps that are outlined below:

Step 1: Begin with a program P and a set of test cases T known to be correct.

Step 2: Run each test case in T against the program P. If a test case fails, that is, the output is incorrect, program P must be modified and retested. If there are no failures, then continue with step 3.

Step 3: Create a set of mutants $\{P_i\}$, each differing from P by a simple, syntactically correct modification of P.

Step 4: Execute each test case in T against each mutant P_i. If the output of the mutant P_i differs from the output of the original program P, the mutant P_i is considered incorrect and is said to be killed by the test case. If P_i produces exactly the same results as the original program P for the tests in T, then one of the following is true:

- P and P_i are *equivalent*. That is, their behaviors cannot be distinguished by any set of test cases. Note that the general problem of deciding whether or not a mutant is equivalent to the original program is undecidable.
- P_i is *killable*. That is, the test cases are insufficient to kill the mutant P_i. In this case, new test cases must be created.

Step 5: Calculate the mutation score for the set of test cases T. The mutation score is the percentage of nonequivalent mutants killed by the test data, that is, Mutation score $= 100 \times D/(N - E)$, where D is the dead mutants, N the total number of mutants, and E the number of equivalent mutants.

Step 6: If the estimated mutation adequacy of T in step 5 is not sufficiently high, then design a new test case that distinguishes P_i from P, add the new test case to T, and go to step 2. If the computed adequacy of T is more than an appropriate threshold, then accept T as a good measure of the correctness of P with respect to the set of mutant programs P_i, and stop designing new test cases.

Mutation testing makes two major assumptions:

1. *Competent Programmer Hypothesis*: This assumption states that programmers are generally competent, and they do not create "random" programs. Therefore, we can assume that for a given problem a programmer will create a correct program except for simple errors. In other words, the mutants to be considered are the ones falling within a small deviation from the original program. In practice, such mutants are obtained by systematically and mechanically applying a set of transformations, called *mutation operators*, to the program under test. These mutation operators are expected to model programming errors made by programmers. In practice, this may be only partly true.
2. *Coupling Effect*: This assumption was first hypothesized in 1978 by DeMillo et al. [9]. The assumption can be restated as complex faults are coupled to simple faults in such a way that a test suite detecting all simple faults in a program will detect most of the complex faults. This assumption has been empirically supported by Offutt [11] and theoretically demonstrated by Wah [12]. The fundamental premise of mutation testing as coined by Geist et al. [13] is: *If the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detect that fault*.

Mutation testing helps the tester to inject, by hypothesis, different types of faults in the code and develop test cases to reveal them. In addition, comprehensive testing can be performed by proper choice of mutant operations. However, a relatively large number of mutant programs need to be tested against many of the test cases before these mutants can be distinguished from the original program. Running the test cases, analyzing the results, identifying equivalent mutants [14], and developing additional test cases to kill the stubborn mutants are all time consuming. Robust automated testing tools such as Mothra [15] can be used to expedite the mutation testing process. Recently, with the availability of massive computing power, there has been a resurgence of mutation testing processes within the industrial community to use as a white-box methodology for unit testing [16, 17]. Researchers have shown that with an appropriate choice of mutant programs mutation testing is as powerful as path testing, domain testing [18], and data flow testing [19].

3.6 DEBUGGING

The programmer, after a program failure, identifies the corresponding fault and fixes it. The process of determining the cause of a failure is known as *debugging*. Debugging occurs as a consequence of a test revealing a failure. Myers proposed three approaches to debugging in his book *The Art of Software Testing* [20]:

- **Brute Force:** The brute-force approach to debugging is preferred by many programmers. Here, "let the computer find the error" philosophy is used.

Print statements are scattered throughout the source code. These print statements provide a crude trace of the way the source code has executed. The availability of a good debugging tool makes these print statements redundant. A dynamic debugger allows the software engineer to navigate by stepping through the code, observe which paths have executed, and observe how values of variables change during the controlled execution. A good tool allows the programmer to assign values to several variables and navigate step by step through the code. Instrumentation code can be built into the source code to detect problems and to log intermediate values of variables for problem diagnosis. One may use a memory dump after a failure has occurred to understand the final state of the code being debugged. The log and memory dump are reviewed to understand what happened and how the failure occurred.

- **Cause Elimination:** The cause elimination approach can be best described as a process involving *induction* and *deduction* [21]. In the induction part, first, all pertinent data related to the failure are collected , such as what happened and what the symptoms are. Next, the collected data are organized in terms of behavior and symptoms, and their relationship is studied to find a pattern to isolate the causes. A cause hypothesis is devised, and the above data are used to prove or disprove the hypothesis. In the deduction part, a list of all possible causes is developed in order of their likelihoods, and tests are conducted to eliminate or substantiate each cause in decreasing order of their likelihoods. If the initial tests indicate that a particular hypothesis shows promise, test data are refined in an attempt to isolate the problem as needed.
- **Backtracking:** In this approach, the programmer starts at a point in the code where a failure was observed and traces back the execution to the point where it occurred. This technique is frequently used by programmers, and this is useful in small programs. However, the probability of tracing back to the fault decreases as the program size increases, because the number of potential backward paths may become too large.

Often, software engineers notice other previously undetected problems while debugging and applying a fix. These newly discovered faults should not be fixed along with the fix in focus. This is because the software engineer may not have a full understanding of the part of the code responsible for the new fault. The best way to deal with such a situation is to file a CR. A new CR gives the programmer an opportunity to discuss the matter with other team members and software architects and to get their approval on a suggestion made by the programmer. Once the CR is approved, the software engineer must file a defect in the defect tracking database and may proceed with the fix. This process is cumbersome, and it interrupts the debugging process, but it is useful for very critical projects. However, programmers often do not follow this because of a lack of a procedure to enforce it.

A Debugging Heuristic The objective of debugging is to precisely identify the cause of a failure. Once the cause is identified, corrective measures are taken to

fix the fault. Debugging is conducted by programmers, preferably by those who wrote the code, because the programmer is the best person to know the source code well enough to analyze the code efficiently and effectively. Debugging is usually a time consuming and error-prone process, which is generally performed under stress. Debugging involves a combination of systematic evaluation, intuition, and, sometimes, a little bit of luck. Given a symptom of a problem, the purpose is to isolate and determine its specific cause. The following heuristic may be followed to isolate and correct it:

Step 1: Reproduce the symptom(s).

- Read the troubleshooting guide of the product. This guide may include conditions and logs, produced by normal code, or diagnostics code specifically written for troubleshooting purpose that can be turned on.
- Try to reproduce the symptoms with diagnostics code turned on.
- Gather all the information and conduct *causal analysis* The goal of causal analysis is to identify the root cause of the problem and initiate actions so that the source of defects is eliminated.

Step 2: Formulate some likely hypotheses for the cause of the problem based on the causal analysis.

Step 3: Develop a test scenario for each hypothesis to be proved or disproved. This is done by designing test cases to provide unambiguous results related to a hypothesis. The test cases may be static (reviewing code and documentation) and/or dynamic in nature. Preferably, the test cases are nondestructive, have low cost, and need minimum additional hardware needs. A test case is said to be destructive if it destroys the hardware setup. For example, cutting a cable during testing is called destructive testing.

Step 4: Prioritize the execution of test cases. Test cases corresponding to the highly probable hypotheses are executed first. Also, the cost factor cannot be overlooked. Therefore, it is desirable to execute the low-cost test cases first followed by the more expensive ones. The programmer needs to consider both factors.

Step 5: Execute the test cases in order to find the cause of a symptom. After executing a test case, examine the result for new evidence. If the test result shows that a particular hypothesis is promising, test data are refined in an attempt to isolate the defect. If necessary, go back to earlier steps or eliminate a particular hypothesis.

Step 6: Fix the problem.

- Fixing the problem may be a simple task, such as adding a line of code or changing a variable in a line of code, or a more complex task such as modifying several lines of code or designing a new unit. In the complex case, defect fixing is a rigorous activity.
- If code review has already been completed for the module which received a fix, then code review must be done once again to avoid

any side effects (collateral damage) due to the changes effected in the module.

- After a possible code review, apply the fix.
- Retest the unit to confirm that the actual cause of failure had been found. The unit is properly debugged and fixed if tests show that the observed failure does not occur any more.
- If there are no dynamic unit test cases that reveal the problem, then add a new test case to the dynamic unit testing to detect possible reoccurrences or other similar problems.
- For the unit under consideration, identify all the test cases that have passed. Now, perform a regression test on the unit with those test cases to ensure that new errors have not been introduced. That is why it is so important to have archived all the test cases that have been designed for a unit. Thus, even unit-level test cases must be managed in a systematic manner to reduce the cost of software development.

Step 7: Document the changes which have been made. Once a defect is fixed, the following changes are required to be applied:

- Document the changes in the source code itself to reflect the change.
- Update the overall system documentation.
- Changes to the dynamic unit test cases.
- File a defect in the defect tracking database if the problem was found after the code was checked in to the version control system.

3.7 UNIT TESTING IN EXTREME PROGRAMMING

A TDD approach to code development is used in the XP methodology [22, 23]. The key aspect of the TDD approach is that a programmer writes low-level tests before writing production code. This is referred to as *test first* [24] in software development. Writing test-driven units is an important concept in the XP methodology. In XP, a few unit tests are coded first, then a simple, partial system is implemented to pass the tests. Then, one more new unit test is created, and additional code is written to pass the new test, but not more, until a new unit test is created. The process is continued until nothing is left to test. The process is illustrated in Figure 3.3 and outlined below:

Step 1: Pick a requirement, that is, a story.

Step 2: Write a test case that will verify a small part of the story and assign a fail verdict to it.

Step 3: Write the code that implements a particular part of the story to pass the test.

Step 4: Execute all tests.

Figure 3.3 Test-first process in XP. (From ref. 24. © 2005 IEEE.)

Step 5: Rework the code, and test the code until all tests pass.

Step 6: Repeat **steps 2–5** until the story is fully implemented.

The simple cycle in Figure 3.3 shows that, at the beginning of each cycle, the intention is for all tests to pass except the newly added test case. The new test case is introduced to *drive* the new code development. At the end of the cycle, the programmer executes all the unit tests, ensuring that each one passes and, hence, the planned task of the code still works. A TDD developer must follow the three laws proposed by Robert C. Martin [25]:

- One may not write production code unless the first failing unit test is written.
- One may not write more of a unit test than is sufficient to fail.
- One may not write more production code than is sufficient to make the failing unit test pass.

These three laws ensure that one must write a portion of a unit test that fails and then write just enough production code to make that unit test pass. The goal of these three laws is not to follow them strictly—it is to decrease the interval between writing unit tests and production code.

Creating unit tests helps a developer focus on what needs to be done. Requirements, that is, user stories, are nailed down firmly by unit tests. Unit tests are released into the code repository along with the code they test. Code without unit tests may not be released. If a unit test is discovered to be missing, it must be created immediately. Creating unit tests independently before coding sets up checks and balances and improves the chances of getting the system right the first time.

Unit tests provide a safety net of regression tests and validation tests so that XP programmers can refactor and integrate effectively.

In XP, the code is being developed by two programmers working together side by side. The concept is called pair programming. The two programmers sit side by side in front of the monitor. One person develops the code tactically and the other one inspects it methodically by keeping in mind the story they are implementing. It is similar to the two-person inspection strategy proposed by Bisant and Lyle [26]. Code inspection is carried out by an author–examiner pair in discrete steps, examining a small part of the implementation of the story in isolation, which is key to the success of the code review process.

3.8 JUNIT: FRAMEWORK FOR UNIT TESTING

The JUnit is a unit testing framework for the Java programming language designed by Kent Beck and Erich Gamma. Experience gained with JUnit has motivated the development of the TDD [22] methodology. The idea in the JUnit framework has been ported to other languages, including C# (NUnit), Python (PyUnit), Fortran (fUnit) and C++ (CPPUnit). This family of unit testing frameworks is collectively referred to as xUnit. This section will introduce the fundamental concepts of JUnit to the reader.

Suppose that we want to test the individual methods of a class called PlanetClass. Let Move() be a method in PlanetClass such that Move() accepts only one input parameter of type *integer* and returns a value of type integer. One can follow the following steps, illustrated using pseudocode in Figure 3.4, to test Move():

- Create an object instance of PlanetClass. Let us call the instance Mars. Now we are interested in testing the method Move() by invoking it on object Mars.
- Select a value for all the input parameters of Move()—this function has just one input parameter. Let us represent the input value to Move() by x.
- Know the expected value to be returned by Move(). Let the expected returned value be y.

```
:
Planet Mars = new Planet(); // Instantiate class Planet to create
                            // an object Mars.
x = ... ; // Select a value for x.
y = ... ; // The expected value to be returned by the call Move(x).
z = Mars.Move(x); // Invoke method Mars() on object Mars.
if (z == y) print("Test passed");
else print("Test failed.");
:
```

Figure 3.4 Sample pseudocode for performing unit testing.

- Invoke method Move() on object Mars with input value x. Let z denote the value returned by Move().
- Now compare y with z. If the two values are identical, then the method Move() in object Mars passes the test. Otherwise, the test is said to have failed.

In a nutshell, the five steps of unit testing are as follows:

- Create an object and select a method to execute.
- Select values for the input parameters of the method.
- Compute the expected values to be returned by the method.
- Execute the selected method on the created object using the selected input values.
- Verify the result of executing the method.

Performing unit testing leads to a programmer consuming some resources, especially time. Therefore, it is useful to employ a general programming framework to code individual test cases, organize a set of test cases as a *test suite*, initialize a test environment, execute the test suite, clean up the test environment, and record the result of execution of individual test cases. In the example shown in Figure 3.4, creating the object Mars is a part of the initialization process. The two print() statements are examples of recording the result of test execution. Alternatively, one can write the result of test execution to a file.

The JUnit framework has been developed to make test writing simple. The framework provides a basic class, called TestCase, to write test cases. Programmers need to *extend* the TestCase class to write a set of individual test cases. It may be noted that to write, for example, 10 test cases, one need not write 10 subclasses of the class TestCase. Rather, one subclass, say MyTestCase, of TestCase, can contain 10 methods—one for each test case.

Programmers need to make assertions about the state of objects while extending the TestCase class to write test cases. For example, in each test case it is required to compare the actual outcome of a computation with the expected outcome. Though an if() statement can be used to compare the equality of two values or two objects, it is seen to be more elegant to write an assert statement to achieve the same. The class TestCase extends a utility class called Assert in the JUnit framework. Essentially, the Assert class provides methods, as explained in the following, to make assertions about the state of objects created and manipulated while testing.

assertTrue(Boolean condition): This assertion passes if the condition is *true*; otherwise, it fails.

assertEquals(Object expected, Object actual): This assertion passes if the expected and the actual objects are equal according to the equals() method; otherwise, the assertion fails.

assertEquals(int expected, int actual): This assertion passes if expected and actual are equal according to the = = operator; otherwise, the assertion

fails. For each primitive type int, float, double, char, byte, long, short, and boolean, the assertion has an overloaded version.

assertEquals(double expected, double actual, double tolerance): This assertion passes if the absolute value of the difference between expected and actual is less than or equal to the tolerance value; otherwise, the assertion fails. The assertion has an overloaded version for float inputs.

assertSame(Object expected, Object actual): This assertion passes if the expected and actual values refer to the same object in memory; otherwise, the assertion fails.

assertNull(Object testobject): This assertion passes if testobject is null; otherwise the assertion fails.

assertFalse(Boolean condition): This is the logical opposite of assertTrue().

The reader may note that the above list of assertions is not exhaustive. In fact, one can build other assertions while extending the TestCase class. When an assertion fails, a programmer may want to know immediately the nature of the failure. This can be done by displaying a message when the assertion fails. Each assertion method listed above accepts an optional *first* parameter of type String—if the assertion fails, then the String value is displayed. This facilitates the programmer to display a desired message when the assertion fails. As an aside, upon failure, the assertEquals() method displays a customized message showing the expected value and the actual value. For example, an assertEquals() method can display the following:

```
junit.framework.AssertionFailedError: expected: <2006> but
  was:<2060>.
```

At this point it is interesting to note that only failed tests are reported. Failed tests can be reported by various means, such as displaying a message, displaying an identifier for the test case, and counting the total number of failed test cases. Essentially, an assertion method throws an exception, called AssertionFailedError, when the assertion fails, and JUnit catches the exception. The code shown in Figure 3.5 illustrates how the assertTrue() assertion works: When the JUnit framework catches an exception, it records the fact that the assertion failed and proceeds to the next test case. Having executed all the test cases, JUnit produces a list of all those tests that have failed.

In Figure 3.6, we show an example of a test suite containing two test cases. In order to execute the two test cases, one needs to create an object instance of

```
static public void assertTrue(Boolean condition) {
       if (!condition)
              throw new AssertionFailedError();
}
```

Figure 3.5 The assertTrue() assertion throws an exception.

```
import TestMe; // TestMe is the class whose methods are going
               //to be tested.
import junit.framework.*; // This contains the TestCase class.

public class MyTestSuite  extends  TestCase { // Create a subclass
                                              //of TestCase

     public void MyTest1() { // This method is the first test case
            TestMe object1 = new TestMe( ... ); // Create an
                                          //instance of TestMe with
                                          //desired parameters
            int x = object1.Method1(...); //  invoke Method1
                                           //on object1
            assertEquals(365, x); // 365 and x are expected and
                                 //actual values, respectively
     }

     public void MyTest2() { // This method is the second test case
            TestMe object2 = new TestMe( ... ); // Create another
                                          //instance of TestMe
                                          //with desired parameters
            double y = object2.Method2(...); //  invoke Method2
                                             //on object2
            assertEquals(2.99, y, 0.0001d); // 2.99 is the expected
                                     // value; y is the actual
                                     // value; 0.0001 is tolerance
                                     // level
     }
}
```

Figure 3.6 Example test suite.

MyTestSuite and invoke the two methods MyTest1() and MyTest2(). Whether or not the two methods, namely Method1() and Method()2, are to be invoked on two different instances of the class TestMe depends on the individual objectives of those two test cases. In other words, it is the programmer who decides whether or not two instances of the class TestMe are to be created.

This section is by no means a thorough exposition of the capabilities of the JUnit framework. Readers are referred to other sources, such as *JUnit Recipes* by Rainsberger [27] and *Pragmatic Unit Testing* by Hunt and Thomas [28]. In addition, tools such as Korat [29], Symstra [30], and Eclat [31] for Java unit testing are being developed and used by researchers.

3.9 TOOLS FOR UNIT TESTING

Programmers can benefit from using tools in unit testing by reducing testing time without sacrificing thoroughness. The well-known tools in everyday life are an editor, a compiler, an operating system, and a debugger. However, in some cases,

the real execution environment of a unit may not be available to a programmer while the code is being developed. In such cases, an emulator of the environment is useful in testing and debugging the code. Other kinds of tools that facilitate effective unit testing are as follows:

1. *Code Auditor:* This tool is used to check the quality of software to ensure that it meets some minimum coding standards. It detects violations of programming, naming, and style guidelines. It can identify portions of code that cannot be ported between different operating systems and processors. Moreover, it can suggest improvements to the structure and style of the source code. In addition, it counts the number of LOC which can be used to measure productivity, that is, LOC produced per unit time, and calculate defect density, that is, number of defects per KLOC.

2. *Bound Checker:* This tool can check for accidental writes into the instruction areas of memory or to any other memory location outside the data storage area of the application. This fills unused memory space with a signature pattern (distinct binary pattern) as a way of determining at a later time whether any of this memory space has been overwritten. The tool can issue diagnostic messages when boundary violations on data items occur. It can detect violation of the boundaries of array, for example, when the array index or pointer is outside its allowed range. For example, if an array z is declared to have a range from $z[0]$ to $z[99]$, it can detect reads and writes outside this range of storage, for example, $z[-3]$ or $z[10]$.

3. *Documenters:* These tools read source code and automatically generate descriptions and caller/callee tree diagram or data model from the source code.

4. *Interactive Debuggers:* These tools assist software developers in implementing different debugging approaches discussed in this chapter. These tools should have the trace-back and breakpoint capabilities to enable the programmers to understand the dynamics of program execution and to identify problem areas in the code. Breakpoint debuggers are based on deductive logic. Breakpoints are placed according to a heuristic analysis of code [32]. Another popular kind of debugger is known as omniscient debugger (ODB), in which there is no deduction. It simply follows the trail of "bad" values back to their source—no "guessing" where to put the breakpoints. An ODB is like "the snake in the grass," that is, if you see a snake in the grass and you pull its tail, sooner or later you get to its head. In contrast, breakpoint debuggers suffer from the "lizard in the grass" problem, that is, when you see the lizard and grab its tail, the lizard breaks off its tail and gets away [33].

5. *In-Circuit Emulators:* An in-circuit emulator, commonly known as ICE, is an invaluable software development tool in embedded system design. It provides a high-speed Ethernet connection between a host debugger and a target microprocessor, enabling developers to perform common source-level debugging activities, such as watching memory and controlling large numbers of registers, in a matter of seconds. It is vital for board bring-up, solving complex problems, and manufacturing or testing of products. Many emulators have advanced features, such as

performance analysis, coverage analysis, buffering of traces, and advance trigger and breakpoint possibilities.

6. *Memory Leak Detectors:* These tools test the allocation of memory to an application which requests for memory, but fails to deallocate. These detect the following overflow problems in application programs:

- Illegal read, that is, accesses to memory which is not allocated to the application or which the application is not authorized to access.
- Reads memory which has not been initialized.
- Dynamic memory overwrites to a memory location that has not been allocated to the application.
- Reading from a memory location not allocated, or not initialized, prior to the read operation.

The tools watch the heap, keep track of heap allocations to applications, and detect memory leaks. The tools also build profiles of memory use, for example, which line-of-code source instruction accesses a particular memory address.

7. *Static Code (Path) Analyzer:* These tools identify paths to test, based on the structure of the code such as McCabe's cyclomatic complexity measure (Table 3.3). Such tools are dependent on source language and require the source code to be recompiled with the tool. These tools can be used to improve productivity, resource management, quality, and predictability by providing complexity measurement metrics.

8. *Software Inspection Support:* Tools can help schedule group inspections. These can also provide status of items reviewed and follow-up actions and distribute the reports of problem resolution. They can be integrated with other tools, such as static code analyzers.

9. *Test Coverage Analyzer:* These tools measure internal test coverage, often expressed in terms of the control structure of the test object, and report the coverage metric. Coverage analyzers track and report what paths were exercised during dynamic unit testing. Test coverage analyzers are powerful tools that increase confidence in product quality by assuring that tests cover all of the structural parts of a unit or a program. An important aspect in test coverage analysis is to identify parts of source code that were never touched by any dynamic unit test. Feedback from the coverage reports to the source code makes it easier to design new unit test cases to cover the specific untested paths.

10. *Test Data Generator:* These tools assist programmers in selecting test data that cause a program to behave in a desired manner. Test data generators can offer several capabilities beyond the basics of data generation:

- They have generate a large number of variations of a desired data set based on a description of the characteristics which has been fed into the tool.
- They can generate test input data from source code.
- They can generate equivalence classes and values close to the boundaries.
- They can calculate the desired extent of boundary value testing.

TABLE 3.3 McCabe Complexity Measure

McCabe's complexity measure is based on the cyclomatic complexity of a programe graph for a module. The metric can be computed using the formula $v = e - n + 2$,
where
v = cyclomatic complexity of graph,
e = number of edges (program flow between nodes)
n = number of nodes (sequential group of program statements)
If a strongly connected graph is constructed (one in which there is an edge between the exit node and the entry node), the calculation is $v = e - n + 1$.
Example: A program graph, illustrated below, is used to depict control flow. Each circled node represents a sequence of program statements, and the flow of control is represented by directed edges. For this graph the cyclomatic complexity is $v = 9 - 8 + 2 = 3$

Source: From ref. 6.

- They can estimate the likelihood of the test data being able to reveal faults.
- They can generate data to assist in mutation analysis.

Automatic generation of test inputs is an active area of research. Several tools, such as CUTE [34], DART [35], and EGT system [36], have been developed by researchers to improve test coverage.

11. *Test Harness:* This class of tools supports the execution of dynamic unit tests by making it almost painless to (i) install the unit under test in a test environment, (ii) drive the unit under test with input data in the expected input format, (iii) generate stubs to emulate the behavior of subordinate modules, and (iv) capture the actual outcome as generated by the unit under test and log or display it in a usable form. Advanced tools may compare the expected outcome with the actual outcome and log a test verdict for each input test data.

12. *Performance Monitors:* The timing characteristics of software components can be monitored and evaluated by these tools. These tools are essential for any real-time system in order to evaluate the performance characteristics of the system, such as delay and throughput. For example, in telecommunication systems, these tools can be used to calculate the end-to-end delay of a telephone call.

13. *Network Analyzers:* Network operating systems such as software that run on routers, switches, and client/server systems are tested by network analyzers. These tools have the ability to analyze the traffic and identify problem areas. Many of these networking tools allow test engineers to monitor performance metrics and diagnose performance problems across the networks. These tools are enhanced to improve the network security monitoring (NSM) capabilities to detect intrusion [37].

14. *Simulators and Emulators:* These tools are used to replace the real software and hardware that are currently not available. Both kinds of tools are used for training, safety, and economy reasons. Some examples are flight simulators, terminal emulators, and emulators for base transceiver stations in cellular mobile networks. These tools are bundled with traffic generators and performance analyzers in order to generate a large volume of input data.

15. *Traffic Generators:* Large volumes of data needed to stress the interfaces and the integrated system are generated by traffic generators. These produce streams of transactions or data packets. For example, in testing routers, one needs a traffic that simulates streams of varying size Internet Protocol (IP) packets arriving from different sources. These tools can set parameters for mean packet arrival rate, duration, and packet size. Operational profiles can be used to generate traffic for load and stability testing.

16. *Version Control:* A version control system provides functionalities to store a sequence of revisions of the software and associated information files under development. A system release is a collection of the associated files from a version control tool perspective. These files may contain source code, compiled code, documentation, and environment information, such as version of the tool used to write the software. The objective of version control is to ensure a systematic and traceable software development process in which all changes are precisely managed, so that a software system is always in a well-defined state. With most of the version control tools, the repository is a central place that holds the master copy of all the files.

The configuration management system (CMS) extends the version control from software and documentation to control the changes made to hardware, firmware, software, documentation, test, test fixtures, test documentation, and execution environments throughout the development and operational life of a system. Therefore, configuration management tools are larger, better variations of version control tools. The characteristics of the version control and configuration management tools are as follows:

- **Access Control:** The tools monitor and control access to components. One can specify which users can access a component or group of components. One can also restrict access to components currently undergoing modification or testing.

- **Cross Referencing:** The tools can maintain linkages among related components, such as problem reports, components, fixes, and documentations.

One can merge files and coordinate multiple updates from different versions to produce one consolidated file.

- **Tracking of Modifications:** The tools maintain records of all modifications to components. These also allow merging of files and coordinate multiple updates from different versions to produce one consolidated file. These can track similarities and differences among versions of code, documentation, and test libraries. They also provide an audit trail or history of the changes from version to version.
- **Release Generation:** The tools can automatically build new system releases and insulate the development, test, and shipped versions of the product.
- **System Version Management:** The tools allow sharing of common components across system versions and controlled use of system versions. They support coordination of parallel development, maintenance, and integration of multiple components among several programmers or project teams. They also coordinate geographically dispersed development and test teams.
- **Archiving:** The tools support automatic archiving of retired components and system versions.

3.10 SUMMARY

This chapter began with a description of unit-level testing, which means identifying faults in a program unit analyzed and executed in isolation. Two complementary types of unit testing were introduced: static unit testing and dynamic unit testing. Static unit testing involves visual inspection and analysis of code, whereas a program unit is executed in a controlled manner in dynamic unit testing.

Next, we described a code review process, which comprises six steps: readiness, preparation, examination, rework, validation, and exit. The goal of code review is to assess the quality of the software in question, not the quality of the process used to develop the product. We discussed a few basic metrics that can be collected from the code review process. Those metrics facilitate estimation of review time and resources required for similar projects. Also, the metrics make code review visible to the upper management and allow upper management to be satisfied with the viability of code review as a testing tool.

We explained several preventive measures that can be taken during code development to reduce the number of faults in a program. The preventive measures were presented in the form of a set of guidelines that programmers can follow to construct code. Essentially, the guidelines focus on incorporating suitable mechanisms into the code.

Next, we studied dynamic unit testing in detail. In dynamic unit testing, a program unit is actually executed, and the outcomes of program execution are observed. The concepts of test driver and stubs were explained in the context of a unit under test. A test driver is a caller of the unit under test and all the

"dummy modules" called by the unit are known as stubs. We described how mutation analysis can be used to locate weaknesses in test data used for unit testing. Mutation analysis should be used in conjunction with traditional unit testing techniques such as domain analysis or data flow analysis. That is, mutation testing is not an alternative to domain testing or data flow analysis.

With the unit test model in place to reveal defects, we examined how programmers can locate faults by debugging a unit. Debugging occurs as a consequence of a test revealing a defect. We discussed three approaches to debugging: brute force, cause elimination, and backtracking. The objective of debugging is to precisely identify the cause of a failure. Given the symptom of a problem, the purpose is to isolate and determine its specific cause. We explained a heuristic to perform program debugging.

Next, we explained dynamic unit testing is an integral part of the XP software development process. In the XP process, unit tests are created prior to coding—this is known as test first. The test-first approach sets up checks and balances to improve the chances of getting things right the first time. We then introduced the JUnit framework, which is used to create and execute dynamic unit tests.

We concluded the chapter with a description of several tools that can be useful in improving the effectiveness of unit testing. These tools are of the following types: code auditor, bound checker, documenters, interactive debuggers, in-circuit emulators, memory leak detectors, static code analyzers, tools for software inspection support, test coverage analyzers, test data generators, tools for creating test harness, performance monitors, network analyzers, simulators and emulators, traffic generators, and tools for version control.

LITERATURE REVIEW

The Institute of Electrical and Electronics Engineers (IEEE) standard 1028-1988 (*IEEE Standard for Software Reviews and Audits: IEEE/ANSI Standard*) describes the detailed examination process for a technical review, an inspection, a software walkthrough, and an audit. For each of the examination processes, it includes an objective, an abstract, special responsibilities, program input, entry criteria, procedures, exit criteria, output, and auditability.

Several improvements on Fagan's inspection techniques have been proposed by researchers during the past three decades. Those proposals suggest ways to enhance the effectiveness of the review process or to fit specific application domains. A number of excellent articles address various issues related to software inspection as follows:

S. Biffl, and M. Halling, "Investigating the Defect Effectiveness and Cost Benefit of Nominal Inspection Teams," *IEEE Transactions on Software Engineering*, Vol. 29, No. 5, May 2003, pp. 385–397.

A. A. Porter and P. M. Johnson, "Assessing Software Review Meeting: Results of a Comparative Analysis of Two Experimental Studies," *IEEE*

Transactions on Software Engineering, Vol. 23, No. 3, March 1997, pp. 129–145.

A. A. Porter, H. P. Siy, C. A. Toman, and L. G. Votta, "An Experiment to Assess the Cost-Benefits of Code Inspection in Large Scale Software Development," *IEEE Transactions on Software Engineering*, Vol. 23, No. 6, June 1997, pp. 329–346.

A. A. Porter and L. G. Votta, "What Makes Inspection Work," *IEEE Software*, Vol. 14, No. 5, May 1997, pp. 99–102.

C. Sauer, D. Jeffery, L. Land, and P. Yetton, "The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Search," *IEEE Transactions on Software Engineering*, Vol. 26, No. 1, January 2000, pp. 1–14.

An alternative non-execution-based technique is formal verification of code. Formal verification consists of mathematical proofs to show that a program is correct. The two most prominent methods for proving program properties are those of Dijkstra and Hoare:

E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

C. A. R. Hoare, "An Axiomatic Basis of Computer Programming," *Communications of the ACM*, Vol. 12, No. 10, October 1969, pp. 576–580.

Hoare presented an axiomatic approach in which properties of program fragments are described using preconditions and postconditions. An example statement with a precondition and a postcondition is {PRE} P {POST}, where PRE is the precondition, POST is the postcondition, and P is the program fragment. Both PRE and POST are expressed in first-order predicate calculus, which means that they can include the universal quantifier $\forall$ ("for all") and existential quantifier $\exists$ ("there exists"). The interpretation of the above statement is that if the program fragment P starts executing in a state satisfying PRE, then if P terminates, P will do so in a state satisfying POST.

Hoare's logic led to Dijkstra's closely related "calculus of programs," which is based on the idea of weakest preconditions. The weakest preconditions R with respect to a program fragment P and a postcondition POST is the set of all states that, when subject to P, will terminate and leave the state of computation in POST. The weakest precondition is written as WP(P, POST).

While mutation testing systematically implants faults in programs by applying syntactic transformations, *perturbation testing* is performed to test a program's robustness by changing the values of program data during run time, so that the subsequent execution will either fail or succeed. Program perturbation is based on three parts of software hypothesis as explained in the following:

- **Execution:** A fault must be executed.
- **Infection:** The fault must change the data state of the computation directly after the fault location.

- **Propagation:** The erroneous data state must propagate to an output variable.

In the perturbation technique, the programmer injects faults in the data state of an executing program and traces the injected faults on the program's output. A fault injection is performed by applying a perturbation function that changes the program's data state. A perturbation function is a mathematical function that takes a data state as its input, changes the data state according to some specified criteria, and produces a modified data state as output. For the interested readers, two excellent references on perturbation testing are as follows:

M. A. Friedman and J. M. Voas, *Software Assessment—Reliability, Safety, Testability*, Wiley, New York, 1995.

J. M. Voas and G. McGraw, *Software Fault Injection—Inoculating Programs Against Errors*, Wiley, New York, 1998.

The paper by Steven J. Zeil ("Testing for Perturbation of Program Statement," *IEEE Transactions on Software Engineering*, Vol. 9, No. 3, May 1983, pp. 335–346) describes a method for deducing sufficient path coverage to ensure the absence of prescribed errors in a program. It models the program computation and potential errors as a vector space. This enables the conditions for nondetection of an error to be calculated. The above article is an advanced reading for students who are interested in perturbation analysis.

Those readers actively involved in software configuration management (SCM) systems or interested in a more sophisticated treatment of the topic must read the article by Jacky Estublier, David Leblang, André V. Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber ("Impact of Software Engineering Research on the Practice of Software Configuration Management," *ACM Transactions on Software Engineering and Methodology*, Vol. 14, No. 4, October 2005, pp. 383–430). The authors discussed the evolution of software configuration management technology, with a particular emphasis on the impact that university and industrial research has had along the way. This article creates a detailed record of the critical value of software configuration management research and illustrates the research results that have shaped the functionality of SCM systems.

REFERENCES

1. M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, July 1976, pp. 182–211; reprinted 1999, pp. 258–287.
2. E. Yourdon. *Structured Walkthroughs*. Prentice-Hall, Englewood Cliffs, NJ, 1979.
3. D. Parnas and M. Lawford. The Role of Inspection in Software Quality Assurance. *IEEE Transactions on Software Engineering*, August 2003, pp. 674–676.
4. G. Myers. A Controlled Experimentat in Program Testing and Code Walk-throughs/Inspections. *Communications of the ACM*, September 1978, pp. 760–768.
5. H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, Reading, MA, 2004.

6. T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, December 1976, pp. 308–320.
7. A. Davis. *Disecting Error Messages*. Dr. Dobb's Journal, June 2005, pp. 34–41.
8. T. Budd and F. Sayward. Users Guide to the Pilot Mutation System, Technical Report 114. Department of Computer Science, Yale University, 1977.
9. R. A. DeMillo, R. J. Lipton, and F. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, April 1978, pp. 34–41.
10. R. G. Hamlet. Testing Programs with the Aid of Compiler. *IEEE Transactions on Software Engineering*, July 1977, pp. 279–290.
11. A. J. Offutt. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering Methodology*, January 1992, pp. 3–18.
12. K. S. H. T. Wah. A Theoretical Study of Fault Coupling. *Journal of Software Testing, Verification, and Reliability*, March 2000, pp. 3–46.
13. R. Geist, A. J. Offutt, and F. Harris. Estimation and Enhancement of Real-Time Software Reliability through Mutation Analysis. *IEEE Transactions on Computers*, May 1992, pp. 550–558.
14. A. J. Offutt and J. Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Journal of Software Testing, Verification, and Reliability*, September 1997, pp. 165–192.
15. R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An Extended Overview of the Mothra Software Testing Environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, Alberta. IEEE Computer Society Press, New York, July 1988, pp. 142–151.
16. P. Chevalley and P. Thevenod-Fosse. A Mutation Analysis Tool for Java Programs. *International Journal on Software Tools for Technology Transfer (STTT),* Springer Berlin /Heidelberg, November 2003, pp. 90–103.
17. A. Kolawa. Mutation Testing: A New Approach to Automatic Error-Detection. *STAR-EAST*, www.StickyMinds.com, 1999.
18. P. G. Frankl and E. J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, March 1993, pp. 202–213.
19. A. P. Mathur and W. E. Wong. An Empirical Comparison of Data Flow and Mutation-Based Test Adequacy Criteria. *Journal of Software Testing, Analysis, and Verification*, March 1994, pp. 9–31.
20. G. Myers. *The Art of Software Testing*, 2nd ed. Wiley, New York, 2004.
21. R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, 2005.
22. K. Beck. *Test-Driven Development*. Addison-Wesley, Reading, MA, 2003.
23. R. Jeffries and G. Melnik. TDD: The Art of Fearless Programming. *IEEE Software*, May/June 2007, pp. 24–30.
24. H. Erdogmus, M. Morisio, and M. Torchiano. On the Effectiveness of the Test-First Approach to Programming. *IEEE Transactions on Software Engineering*, March 2005, pp. 226–237.
25. R. C. Martin. *Professionalism and Test-Driven Development. IEEE Software*, May/June 2007, pp. 32–36.
26. D. B. Bisant and J. R. Lyle. Two Person Inspection Method to Improve Programming Productivity. *IEEE Transactions on Software Engineering*, October 1989, pp. 1294–1304.
27. J. B. Rainsberger. *JUnit Recipes*. Manning Publications, Greenwich, Connecticut 2005.
28. A. Hunt and D. Thomas. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Bookshelf, Lewisville, Texas, 2004.
29. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. Paper presented at the ACM International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy, 2002, pp. 123–133.
30. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Edinburgh, U.K., Springer Berlin/Hiedelberg, 2005, pp. 365–381.
31. C. Pacheco and M. D. Ernst. Eclat: Automatic Generation and Classification of Test Inputs. Paper presented at ECOOP 2005 Object-Oriented Programming, 19th European Conference, Glasgow, Scotland, July 2005, pp. 504–527.
32. D. Spinellis. Debuggers and Logging Frameworks. *IEEE Software*, September 2006, pp. 98–99.

33. B. Lewis. Omniscient Debugging. *Dr. Dobb's Journal*, June 2005, pp. 16–24.
34. K. Sen, D. Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference*, held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lisbon, Portugal, September 2005, pp. 263–272.
35. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago IL, ACM Press, New York, 2005, pp. 213–223.
36. C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. *Lecture Notes in Computer Science (LNCS)*, Vol. 3639, 2005, pp. 2–23.
37. R. Bejtlich. *The Tao of Network Security Monitoring: Beyond Intrusion Detection*. Addison-Wesley, Boston, MA, 2005.

Exercises

1. Study the Yourdon [2] concept of a design walkthrough and the IBM concept [1] of a design inspection. Discuss the similarities and the differences between them.
2. A software engineering group is developing a mission-critical software system that will launch laser-guided missiles to its destinations. This is a new kind of product that was never built by the company. As a quality assurance manager, which code review methodology—walkthrough or inspection—would you recommend? Justify your answer.
3. What size of a review team would you recommend for the project in exercise 2, and why? What are the different roles of each member of the review team? What groups should send representatives to participate in code review?
4. Suppose that the C programming language is chosen in the project in exercise 2. Recommend a detailed code review checklist to the review team.
5. In addition to code review, what other static unit testing techniques would you recommend for the project in exercise 3? Justify your answer.
6. Describe the special role of a recordkeeper.
7. Discuss the importance of code review rework and validation.
8. Draw a control flow graph for the following sample code. Determine the cyclomatic complexity of the graph.

```
(a) sum_of_all_positive_numbers(a, num_of_entries, sum)
(b) sum = 0
(c) init = 1
(d)    while(init <= num_of_entries)
(e)        if a[init] > 0
(f)            sum = sum + a[init]
           endif
(g)        init = init + 1
       endwhile
(h) end sum_of_all_positive_numbers
```

9. A test engineer generates 70 mutants of a program P and 150 test cases to test the program P. After the first iteration of mutation testing, the tester finds 58 dead mutants and 4 equivalent mutants. Calculate the mutation score for this test suite. Is the test suite adequate for program P? Should the test engineer develop additional test cases? Justify your answer.
10. There is some debate as to whether code should be compiled before it is reviewed and vice versa. Based on your experience, give an opinion on this matter.
11. Attempt to draw a control flow graph for a module that you have recently developed. Determine the cyclomatic complexity for the module. Is the module too complex?
12. For your current software project, conduct a formal code review as described in Section 3.2.
13. For your current software project, develop dynamic unit test cases for each of the units in the JUnit framework if the code is in Java or in an appropriate xUnit framework.

CHAPTER 4

Control Flow Testing

He who controls the present, controls the past. He who controls the past, controls the future.
— *George Orwell*

4.1 BASIC IDEA

Two kinds of basic statements in a program unit are *assignment* statements and *conditional* statements. An assignment statement is explicitly represented by using an assignment symbol, " = ", such as x = 2*y;, where x and y are variables. Program *conditions* are at the core of conditional statements, such as if(), for() loop, while() loop, and goto. As an example, in if(x! = y), we are testing for the inequality of x and y. In the absence of conditional statements, program instructions are executed in the sequence they appear. The idea of successive execution of instructions gives rise to the concept of *control flow* in a program unit. Conditional statements alter the default, sequential control flow in a program unit. In fact, even a small number of conditional statements can lead to a complex control flow structure in a program.

Function calls are a mechanism to provide *abstraction* in program design. A call to a program function leads to control entering the called function. Similarly, when the called function executes its *return* statement, we say that control exits from the function. Though a function can have many return statements, for simplicity, one can restructure the function to have exactly one return. A program unit can be viewed as having a well-defined entry point and a well-defined exit point. The execution of a sequence of instructions from the entry point to the exit point of a program unit is called a program *path*. There can be a large, even infinite, number of paths in a program unit. Each program path can be characterized by an input and an expected output. A specific input value causes a specific program path to be executed; it is expected that the program path performs the desired computation, thereby producing the expected output value. Therefore, it may seem natural to execute as many program paths as possible. Mere execution of a large number of

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

paths, at a higher cost, may not be effective in revealing defects. Ideally, one must strive to execute fewer paths for better effectiveness.

The concepts of *control flow* in computer programs [1], program paths [2], and control flow testing [2–8] have been studied for many decades. Tools are being developed to support control flow testing [9]. Such tools identify paths from a program unit based on a user-defined criterion, generate the corresponding input to execute a selected path, and generate program stubs and drivers to execute the test. Control flow testing is a kind of structural testing, which is performed by programmers to test code written by them. The concept is applied to small *units* of code, such as a function. Test cases for control flow testing are derived from the source code, such as a program unit (e.g., a function or method), rather than from the entire program.

Structurally, a path is a sequence of statements in a program unit, whereas, semantically, it is an execution instance of the unit. For a given set of input data, the program unit executes a certain path. For another set of input data, the unit may execute a different path. The main idea in control flow testing is to appropriately select a few paths in a program unit and observe whether or not the selected paths produce the expected outcome. By executing a few paths in a program unit, the programmer tries to assess the behavior of the entire program unit.

4.2 OUTLINE OF CONTROL FLOW TESTING

The overall idea of generating test input data for performing control flow testing has been depicted in Figure 4.1. The activities performed, the intermediate results produced by those activities, and programmer preferences in the test generation process are explained below.

Inputs: The *source code* of a program unit and a set of *path selection criteria* are the inputs to a process for generating test data. In the following, two examples of path selection criteria are given.

Example. Select paths such that every statement is executed at least once.

Example. Select paths such that every conditional statement, for example, an if() statement, evaluates to *true* and *false* at least once on different occasions. A conditional statement may evaluate to true in one path and false in a second path.

Generation of a Control Flow Graph: A control flow graph (CFG) is a detailed graphical representation of a program unit. The idea behind drawing a CFG is to be able to visualize all the paths in a program unit. The process of drawing a CFG from a program unit will be explained in the following section. If the process of test generation is automated, a compiler can be modified to produce a CFG.

Figure 4.1 Process of generating test input data for control flow testing.

Selection of Paths: Paths are selected from the CFG to satisfy the path selection criteria, and it is done by considering the structure of the CFG.

Generation of Test Input Data: A path can be executed if and only if a certain instance of the inputs to the program unit causes all the conditional statements along the path to evaluate to true or false as dictated by the control flow. Such a path is called a *feasible* path. Otherwise, the path is said to be *infeasible*. It is essential to identify certain values of the inputs from a given path for the path to execute.

Feasibility Test of a Path: The idea behind checking the feasibility of a selected path is to meet the path selection criteria. If some chosen paths are found to be infeasible, then new paths are selected to meet the criteria.

4.3 CONTROL FLOW GRAPH

A CFG is a graphical representation of a program unit. Three symbols are used to construct a CFG, as shown in Figure 4.2. A rectangle represents a sequential

Figure 4.2 Symbols in a CFG.

computation. A maximal sequential computation can be represented either by a single rectangle or by many rectangles, each corresponding to one statement in the source code.

We label each computation and decision box with a unique integer. The two branches of a decision box are labeled with **T** and **F** to represent the true and false evaluations, respectively, of the condition within the box. We will not label a merge node, because one can easily identify the paths in a CFG even without explicitly considering the merge nodes. Moreover, not mentioning the merge nodes in a path will make a path description shorter.

We consider the openfiles() function shown in Figure 4.3 to illustrate the process of drawing a CFG. The function has three statements: an assignment statement int i = 0;, a conditional statement if(), and a return(i) statement. The reader may note that irrespective of the evaluation of the if(), the function performs the same action, namely, null. In Figure 4.4, we show a high-level representation of

```
FILE *fptr1, *fptr2, *fptr3; /* These are global variables. */

int openfiles(){
   /*
      This function tries to open files "file1", "file2", and
      "file3" for read access, and returns the number of files
      successfully opened. The file pointers of the opened files
      are put in the global variables.
   */
     int i = 0;
     if(
         ((( fptr1 = fopen("file1", "r")) != NULL) && (i++)
                                                   && (0)) ||
         ((( fptr2 = fopen("file2", "r")) != NULL) && (i++)
                                                   && (0)) ||
         ((( fptr3 = fopen("file3", "r")) != NULL) && (i++))
     );
     return(i);
}
```

Figure 4.3 Function to open three files.

Figure 4.4 High-level CFG representation of openfiles(). The three nodes are numbered 1, 2, and 3.

the control flow in openfiles() with three nodes numbered 1, 2, and 3. The flow graph shows just two paths in openfiles().

A closer examination of the condition part of the if() statement reveals that there are not only Boolean and relational operators in the condition part, but also assignment statements. Some of their examples are given below:

Assignment statements: fptr1 = fopen("file1", "r") and i++

Relational operator: fptr1! = NULL

Boolean operators: && and ||

Execution of the assignment statements in the condition part of the if statement depends upon the component conditions. For example, consider the following component condition in the if part:

```
((( fptr1 = fopen("file1", "r")) != NULL) && (i++) && (0))
```

The above condition is executed as follows:

- Execute the assignment statement fptr1 = fopen("file1", "r").
- Execute the relational operation fptr1! = NULL.
- If the above relational operator evaluates to false, skip the evaluation of the subsequent condition components (i++) && (0).
- If the relational operator evaluates to true, then first (i) is evaluated to true or false. Irrespective of the outcome of this evaluation, the next statement executed is (i++).
- If (i) has evaluated to true, then the condition (0) is evaluated. Otherwise, evaluation of (0) is skipped.

In Figure 4.5, we show a detailed CFG for the openfiles() function. The figure illustrates a fact that a CFG can take up a complex structure even for a small program unit.

We give a Java method, called ReturnAverage(), in Figure 4.6. The method accepts four parameters, namely value, AS, MIN, and MAX, where value is an integer array and AS is the maximum size of the array. The array can hold fewer number of elements than AS; such a scenarion is semantically represented by having the value −999 denoting the end of the array. For example, AS = 15,

Figure 4.5 Detailed CFG representation of openfiles(). The numbers 1–21 are the nodes.

whereas the 10th element of the array is −999, which means that there are 10 elements—0–9–in the array. MIN and MAX are two integer values that are used to perform certain computations within the method. The method sums up the values of all those elements of the array which fall within the closed range [MIN, MAX], counts their number, and returns their average value. The CFG of the method is shown in Figure 4.7.

4.4 PATHS IN A CONTROL FLOW GRAPH

We assume that a control flow graph has exactly one *entry* node and exactly one *exit* node for the convenience of discussion. Each node is labeled with a unique

```
public static double ReturnAverage(int value[],
                          int AS, int MIN, int MAX){
  /*
   Function: ReturnAverage  Computes the  average
   of all  those  numbers in the  input array  in
   the  positive  range  [MIN, MAX]. The  maximum
   size  of the array is AS. But, the  array size
   could be smaller than AS in which case the end
   of input is represented by -999.
  */
     int i, ti, tv, sum;
     double av;
     i = 0; ti = 0; tv = 0; sum = 0;
     while (ti < AS && value[i] != -999) {
         ti++;
         if (value[i] >= MIN && value[i] <= MAX) {
            tv++;
            sum = sum + value[i];
         }
         i++;
     }
     if (tv > 0)
        av = (double)sum/tv;
     else
        av = (double) -999;
     return (av);
}
```

Figure 4.6 Function to compute average of selected integers in an array. This program is an adaptation of "Figure 2. A sample program" in ref. 10. (With permission from the Australian Computer Society.)

integer value. Also, the two branches of a decision node are appropriately labeled with true (T) or false (F). We are interested in identifying entry–exit paths in a CFG. A path is represented as a sequence of computation and decision nodes from the entry node to the exit node. We also specify whether control exits a decision node via its true or false branch while including it in a path.

In Table 4.1, we show a few paths from the control flow graph of Figure 4.7. The reader may note that we have arbitrarily chosen these paths without applying any path selection criterion. We have unfolded the loop just once in **path 3**, whereas **path 4** unfolds the same loop twice, and these are two distinct paths.

4.5 PATH SELECTION CRITERIA

A CFG, such as the one shown in Figure 4.7, can have a large number of different paths. One may be tempted to test the execution of each and every path in a program unit. For a program unit with a small number of paths, executing all the paths may

Figure 4.7 A CFG representation of ReturnAverage(). Numbers 1–13 are the nodes.

TABLE 4.1 Examples of Path in CFG of Figure 4.7

Path 1	**1-2-3(F)-10(T)-12-13**
Path 2	**1-2-3(F)-10(F)-11-13**
Path 3	**1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13**
Path 4	**1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13**

be desirable and achievable as well. On the other hand, for a program unit with a large number of paths, executing every distinct path may not be practical. Thus, it is more productive for programmers to select a small number of program paths in an effort to reveal defects in the code. Given the set of all paths, one is faced with a question "What paths do I select for testing?" The concept of path selection criteria is useful is answering the above question. In the following, we state the advantages of selecting paths based on defined criteria:

- All program constructs are exercised at least once. The programmer needs to observe the outcome of executing each program construct, for example, statements, Boolean conditions, and returns.
- We do not generate test inputs which execute the same path repeatedly. Executing the same path several times is a waste of resources. However, if each execution of a program path potentially updates the *state* of the system, for example, the database state, then multiple executions of the same path may not be identical.
- We know the program features that have been tested and those not tested. For example, we may execute an if statement only once so that it evaluates to true. If we do not execute it once again for its false evaluation, we are, at least, aware that we have not observed the outcome of the program with a false evaluation of the if statement.

Now we explain the following well-known path selection criteria:

- Select *all* paths.
- Select paths to achieve complete *statement* coverage.
- Select paths to achieve complete *branch* coverage.
- Select paths to achieve *predicate* coverage.

4.5.1 All-Path Coverage Criterion

If *all* the paths in a CFG are selected, then one can detect all faults, except those due to *missing path* errors. However, a program may contain a large number of paths, or even an infinite number of paths. The small, loop-free openfiles() function shown in Figure 4.3 contains more than 25 paths. One does not know whether or not a path is feasible at the time of selecting paths, though only eight of all those paths are feasible. If one selects all possible paths in a program, then we say that the *all-path* selection criterion has been satisfied.

Let us consider the example of the openfiles() function. This function tries to open the three files file1, file2, and file3. The function returns an integer representing the number of files it has successfully opened. A file is said to be successfully opened with "read" access if the file exists. The existence of a file is either "yes" or "no." Thus, the input domain of the function consists of eight combinations of the existence of the three files, as shown in Table 4.2.

We can trace a path in the CFG of Figure 4.5 for each input, that is, each row of Table 4.2. Ideally, we identify test inputs to execute a certain path in a

TABLE 4.2 Input Domain of openfiles()

Existence of file1	Existence of file2	Existence of file3
No	No	No
No	No	Yes
No	Yes	No
No	Yes	Yes
Yes	No	No
Yes	No	Yes
Yes	Yes	No
Yes	Yes	Yes

TABLE 4.3 Inputs and Paths in openfiles()

Input	Path
< No, No, No >	**1-2-3(F)-8-9(F)-14-15(F)-19-21**
< Yes, No, No >	**1-2-3(T)-4(F)-6-8-9(F)-14-15(F)-19-21**
< Yes, Yes, Yes >	**1-2-3(T)-4(F)-6-8-9(T)-10(T)-11-13(F)-14-15(T)-16(T)-18-20-21**

program; this will be explained later in this chapter. We give three examples of the paths executed by the test inputs (Table 4.3). In this manner, we can identify eight possible paths in Figure 4.5. The all-paths selection criterion is desirable since it can detect faults; however, it is difficult to achieve in practice.

4.5.2 Statement Coverage Criterion

Statement coverage refers to executing individual program statements and observing the outcome. We say that 100% statement coverage has been achieved if *all* the statements have been executed at least once. Complete statement coverage is the *weakest* coverage criterion in program testing. Any test suite that achieves less than statement coverage for new software is considered to be unacceptable.

All program statements are represented in some form in a CFG. Referring to the ReturnAverage() method in Figure 4.6 and its CFG in Figure 4.7, the four assignment statements

```
i   = 0;
ti  = 0;
tv  = 0;
sum = 0;
```

have been represented by node **2**. The while statement has been represented as a loop, where the loop control condition

```
(ti < AS && value[i] != -999)
```

has been represented by nodes **3** and **4**. Thus, covering a statement in a program means visiting one or more nodes representing the statement, more precisely, selecting a **feasible** entry–exit path that includes the corresponding nodes. Since a single entry–exit path includes many nodes, we need to select just a few paths to cover all the nodes of a CFG. Therefore, the basic problem is to select a few feasible paths to cover all the nodes of a CFG in order to achieve the complete statement coverage criterion. We follow these rules while selecting paths:

- Select short paths.
- Select paths of increasingly longer length. Unfold a loop several times if there is a need.
- Select arbitrarily long, "complex" paths.

One can select the two paths shown in Figure 4.4 to achieve complete statement coverage.

4.5.3 Branch Coverage Criterion

Syntactically, a branch is an outgoing edge from a node. All the rectangle nodes have at most one outgoing branch (edge). The exit node of a CFG does not have an outgoing branch. All the diamond nodes have two outgoing branches. Covering a branch means selecting a path that includes the branch. Complete branch coverage means selecting a number of paths such that every branch is included in at least one path.

In a preceding discussion, we showed that one can select two paths, **SCPath 1** and **SCPath 2** in Table 4.4, to achieve complete statement coverage. These two paths cover *all* the nodes (statements) and most of the branches of the CFG shown in Figure 4.7. The branches which are not covered by these two paths have been highlighted by bold dashed lines in Figure 4.8. These uncovered branches correspond to the three independent conditions

```
value[i] != -999
value[i] >= MIN
value[i] <= MAX
```

evaluating to false. This means that as a programmer we have not observed the outcome of the program execution as a result of the conditions evaluating to false. Thus, complete branch coverage means selecting enough number of paths such that every condition evaluates to true at least once and to false at least once.

We need to select more paths to cover the branches highlighted by the bold dashed lines in Figure 4.8. A set of paths for complete branch coverage is given in Table 4.5.

TABLE 4.4 Paths for Statement Coverage of CFG of Figure 4.7

SCPath 1	**1-2-3(F)-10(F)-11-13**
SCPath 2	**1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13**

Figure 4.8 Dashed arrows represent the branches not covered by statement covering in Table 4.4.

TABLE 4.5 Paths for Branch Coverage of CFG of Figure 4.7

BCPath 1	**1-2-3(F)-10(F)-11-13**
BCPath 2	**1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13**
BCPath 3	**1-2-3(T)-4(F)-10(F)-11-13**
BCPath 4	**1-2-3(T)-4(T)-5-6(F)-9-3(F)-10(F)-11-13**
BCPath 5	**1-2-3(T)-4(T)-5-6(T)-7(F)-9-3(F)-10(F)-11-13**

4.5.4 Predicate Coverage Criterion

We refer to the partial CFG of Figure 4.9*a* to explain the concept of predicate coverage. OB1, OB2, OB3, and OB are four Boolean variables. The program computes the values of the individual variables OB1, OB2, and OB3— details of their computation are irrelevant to our discussion and have been omitted. Next, OB is computed as shown in the CFG. The CFG checks the value of OB and executes either OBlock1 or OBlock2 depending on whether OB evaluates to true or false, respectively.

We need to design just two test cases to achieve both statement coverage and branch coverage. We select inputs such that the four Boolean conditions in Figure 4.9*a* evaluate to the values shown in Table 4.6. The reader may note that we have shown just one way of forcing OB to true. If we select inputs so that these two cases hold, then we do not observe the effect of the computations taking place in nodes **2** and **3**. There may be faults in the computation parts of nodes **2** and **3** such that OB2 and OB3 always evaluate to false.

Figure 4.9 Partial CFG with (*a*) OR operation and (*b*) AND operation.

TABLE 4.6 Two Cases for Complete Statement and Branch Coverage of CFG of Figure 4.9*a*

Cases	OB1	OB2	OB3	OB
1	T	F	F	T
2	F	F	F	F

Therefore, there is a need to design test cases such that a path is executed under all possible conditions. The False branch of node **5** (Figure 4.9*a*) is executed under exactly one condition, namely, when OB1 = False, OB2 = False, and OB3 = False, whereas the true branch executes under *seven* conditions. If all possible combinations of truth values of the conditions affecting a selected path have been explored under some tests, then we say that *predicate coverage* has been achieved. Therefore, the path taking the true branch of node **5** in Figure 4.9*a* must be executed for all seven possible combinations of truth values of OB1, OB2, and OB3 which result in OB = True.

A similar situation holds for the partial CFG shown in Figure 4.9*b*, where AB1, AB2, AB3, and AB are Boolean variables.

4.6 GENERATING TEST INPUT

In Section 4.5 we explained the concept of path selection criteria to cover certain aspects of a program with a set of paths. The program aspects we considered were all statements, true and false evaluations of each condition, and combinations of conditions affecting execution of a path. Now, having identified a path, the question is how to select input values such that when the program is executed with the selected inputs, the chosen paths get executed. In other words, we need to identify inputs to force the executions of the paths. In the following, we define a few terms and give an example of generating test inputs for a selected path.

1. *Input Vector:* An input vector is a collection of all data entities read by the routine whose values must be fixed prior to entering the routine. Members of an input vector of a routine can take different forms as listed below:

- Input arguments to a routine
- Global variables and constants
- Files
- Contents of registers in assembly language programming
- Network connections
- Timers

A file is a complex input element. In one case, mere existence of a file can be considered as an input, whereas in another case, contents of the file are considered

to be inputs. Thus, the idea of an input vector is more general than the concept of input arguments of a function.

Example. An input vector for openfiles() (Figure 4.3) consists of individual presence or absence of the files file1, file2, and file3.

Example. The input vector of the ReturnAverage() method shown in Figure 4.6 is < value [], AS, MIN, MAX > .

2. *Predicate:* A predicate is a logical function evaluated at a decision point.

Example. The construct ti < AS is the predicate in decision node **3** of Figure 4.7.

Example. The construct OB is the predicate in decision node **5** of Figure 4.9.

3. *Path Predicate:* A path predicate is the set of predicates associated with a path.

The path in Figure 4.10 indicates that nodes **3**, **4**, **6**, **7**, and **10** are decision nodes. The predicate associated with node **3** appears twice in the path; in the first instance it evaluates to true and in the second instance it evaluates to false. The path predicate associated with the path under consideration is shown in Figure 4.11.

We also specify the intended evaluation of the component predicates as found in the path specification. For instance, we specify that value[i] $!= -999$ must evaluate to true in the path predicate shown in Figure 4.11. We keep this additional information for the following two reasons:

- In the absence of this additional information denoting the intended evaluation of a predicate, we will have no way to distinguish between the two instances of the predicate ti < AS, namely **3(T)** and **3(F)**, associated with node **3**.

1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13

Figure 4.10 Example of a path from Figure 4.7.

```
         ti < AS  ≡ True
value[i] != -999  ≡ True
 value[i] >= MIN  ≡ True
 value[i] <= MAX  ≡ True
         ti < AS  ≡ False
          tv > 0  ≡ True
```

Figure 4.11 Path predicate for path in Figure 4.10.

- We must know whether the individual component predicates of a path predicate evaluate to true or false in order to generate path forcing inputs.

4. *Predicate Interpretation:* The path predicate shown in Figure 4.11 is composed of elements of the input vector < value[], AS, MIN, MAX >, a vector of local variables < i, ti, tv >, and the constant −999. The local variables are not visible outside a function but are used to

- hold intermediate results,
- point to array elements, and
- control loop iterations.

In other words, they play no roles in selecting inputs that force the paths to execute. Therefore, we can easily substitute all the local variables in a predicate with the elements of the input vector by using the idea of symbolic substitution. Let us consider the method shown in Figure 4.12. The input vector for the method in Figure 4.12 is given by < x1, x2 > . The method defines a local variable y and also uses the constants 7 and 0.

The predicate

```
x1 + y >= 0
```

can be rewritten as

```
x1 + x2 + 7 >= 0
```

by symbolically substituting y with $x_2 + 7$. The rewritten predicate

```
x1 + x2 + 7 >= 0
```

has been expressed solely in terms of the input vector < x1,x2 > and the constant vector < 0,7 > . Thus, *predicate interpretation* is defined as the process of symbolically substituting operations along a path in order to express the predicates solely in terms of the input vector and a constant vector.

In a CFG, there may be several different paths leading up to a decision point from the initial node, with each path doing different computations. Therefore, a predicate may have different interpretations depending on how control reaches the predicate under consideration.

```
public static int SymSub(int x1, int x2){
int y;
y = x2 + 7;
if (x1 + y >= 0)
     return (x2 + y);
else return (x2 - y);
}
```

Figure 4.12 Method in Java to explain symbolic substitution [11].

5. *Path Predicate Expression:* An interpreted path predicate is called a path predicate expression. A path predicate expression has the following properties:

- It is void of local variables and is solely composed of elements of the input vector and possibly a vector of constants.
- It is a set of constraints constructed from the elements of the input vector and possibly a vector of constants.
- Path forcing input values can be generated by solving the set of constraints in a path predicate expression.
- If the set of constraints cannot be solved, there exist no input which can cause the selected path to execute. In other words, the selected path is said to be *infeasible*.
- An infeasible path does not imply that one or more components of a path predicate expression are unsatisfiable. It simply means that the total combination of all the components in a path predicate expression is unsatisfiable.
- Infeasibility of a path predicate expression suggests that one considers other paths in an effort to meet a chosen path selection criterion.

Example. Consider the path shown in Figure 4.10 from the CFG of Figure 4.7. Table 4.7 shows the nodes of the path in column 1, the corresponding description of each node in column 2, and the interpretation of each node in column 3. The

TABLE 4.7 Interpretation of Path Predicate of Path in Figure 4.10.

Node	Node Description	Interpreted Description
1	Input vector:	
	< value[], AS, MIN, MAX >	
2	i = 0, ti = 0,	
	tv = 0, sum = 0	
3(T)	ti < AS	0 < AS
4(T)	value[i]! = − 999	value[0]! = − 999
5	ti++	ti = 0 + 1 = 1
6(T)	value[i] > = MIN	value[0] > = MIN
7(T)	value[i] < = MAX	value[0] < = MAX
8	tv++	tv = 0 + 1 = 1
	sum = sum + value[i]	sum = 0 + value[0]
		= value[0]
9	i++	i = 0 + 1 = 1
3(F)	ti < AS	1 < AS
10(T)	tv > 0	1 > 0
12	av = (double) sum/tv	av = (double) value[0]/1
13	return(av)	return(value[0])

Note: The bold entries in column 1 denote interpreted predicates.

intended evaluation of each interpreted predicate can be found in column 1 of the same row.

We show the path predicate expression of the path under consideration in Figure 4.13 for the sake of clarity. The rows of Figure 4.13 have been obtained from Table 4.11 by combining each interpreted predicate in column 3 with its intended evaluation in column 1. Now the reader may compare Figures 4.11 and 4.13 to note that the predicates in Figure 4.13 are interpretations of the corresponding predicates in Figure 4.11.

Example. We show in Figure 4.14 an infeasible path appearing in the CFG of Figure 4.7. The path predicate and its interpretation are shown in Table 4.8, and the path predicate expression is shown in Figure 4.15. The path predicate expression is unsolvable because the constraint $0 > 0 \equiv$ True is unsatisfiable. Therefore, the path shown in Figure 4.14 is an infeasible path.

```
          0 < AS   ≡ True    ........ (1)
value[0] != -999   ≡ True    ........ (2)
 value[0] >= MIN   ≡ True    ........ (3)
 value[0] <= MAX   ≡ True    ........ (4)
          1 < AS   ≡ False   ........ (5)
           1 > 0   ≡ True    ........ (6)
```

Figure 4.13 Path predicate expression for path in Figure 4.10.

1-2-3(T)-4(F)-10(T)-12-13.

Figure 4.14 Another example path from Figure 4.7.

TABLE 4.8 Interpretation of Path Predicate of Path in Figure 4.14.

Node	Node Description	Interpreted Description
1	Input vector:	
	< value[], AS, MIN, MAX >	
2	i = 0, ti = 0,	
	tv = 0, sum = 0	
3(T)	ti < AS	0 < AS
4(F)	value[i]! = − 999	value[0]! = − 999
10(T)	tv > 0	0 > 0
12	av = (double)sum/tv	av = (double)value[0]/0
13	return(av)	return((double) value[0]/0)

Note: The bold entries in column 1 denote interpreted predicates.

```
          0 < AS   ≡ True  ........ (1)
value[0] != -999   ≡ True  ........ (2)
           0 > 0   ≡ True  ........ (3)
```

Figure 4.15 Path predicate expression for path in Figure 4.14.

```
      AS  = 1
     MIN  = 25
     MAX  = 35
value[0]  = 30
```

Figure 4.16 Input data satisfying constraints of Figure 4.13.

6. *Generating Input Data from Path Predicate Expression:* We must solve the corresponding path predicate expression in order to generate input data which can force a program to execute a selected path. Let us consider the path predicate expression shown in Figure 4.13. We observe that constraint 1 is always satisfied. Constraints 1 and 5 must be solved together to obtain AS = 1. Similarly, constraints 2, 3, and 4 must be solved together. We note that MIN $<=$ value[0] $<=$ MAX and value[0]! $=$ -999. Therefore, we have many choices to select values of MIN, MAX, and value[0]. An instance of the solutions of the constraints of Figure 4.13 is shown in Figure 4.16.

4.7 EXAMPLES OF TEST DATA SELECTION

We give examples of selected test data to achieve complete statement and branch coverage. We show four sets of test data in Table 4.9. The first two data sets cover all statements of the CFG in Figure 4.7. However, we need all four sets of test data for complete branch coverage.

If we execute the method ReturnAverage shown in Figure 4.6 with the four sets of test input data shown in Figure 4.9, then each statement of the method is executed at least once, and every Boolean condition evaluates once to true and

TABLE 4.9 Test Data for Statement and Branch Coverage

	Input Vector			
Test Data Set	AS	MIN	MAX	value[]
1	1	5	20	[10]
2	1	5	20	[− 999]
3	1	5	20	[4]
4	1	5	20	[25]

once to false. We have thoroughly tested the method in the sense of complete branch coverage. However, it is possible to introduce simple faults in the method which can go undetected when the method with the above four sets of test data is executed. Two examples of fault insertion are given below.

Example. We replace the correct statement

```
av = (double) sum/tv;
```

with a faulty statement

```
av = (double) sum/ti;
```

in the method. Here the fault is that the method computes the average of the total number of inputs, denoted by ti, rather than the total number of valid inputs, denoted by tv.

Example. We replace the correct statement

```
sum = sum + value[i];
```

with a faulty statement

```
sum = value[i];
```

in the method. Here the fault is that the method no more computes the sum of all the valid inputs in the array. In spite of the fault, the first set of test data produce the correct result due to *coincidental correctness*.

The above two examples of faults lead us to the following conclusions:

- One must generate test data to satisfy certain selection criteria, because those selection criteria identify the aspects of a program that we want to cover.
- Additional tests, which are much longer than the simple tests generated to meet coverage criteria, must be generated after the coverage criteria have been met.
- Given a set of test data for a program, we can inject faults into the program which go undetected by those test cases.

4.8 CONTAINING INFEASIBLE PATHS

Woodward, Hedley, and Hennell [12] have identified some practical problems in applying the idea of path testing. First, a CFG may contain a very large number of paths; therefore, the immediate challenge is to decide which paths to select to derive test cases. Second, it may not be feasible to execute many of the selected paths. Thus, it is useful to apply a path selection strategy: First, select as many short paths as feasible; next choose longer paths to achieve better coverage of statements, branches, and predicates. A large number of infeasible paths in a CFG complicate the process of test selection. To simplify path-based unit testing, it is

useful to reduce the number of infeasible paths in a program unit through language design, program design, and program transformation. Brown and Nelson [13] have demonstrated the possibility of writing code with no infeasible paths.

Bertolino and Marre [14] have given an algorithm to generate a set of paths, to cover all the branchs of a CFG, to reduce the number of infeasible paths in the chosen set. Their algorithm is based on the idea of a *reduced* flow graph, called a *ddgraph*. The algorithm uses the concepts of *dominance* and *implications* among the arcs of a ddgraph.

Yates and Malevris [15] have suggested a strategy to reduce the number of infeasible paths in a set of paths to achieve branch coverage. They suggest selecting a path cover, that is, a set of paths, whose constituent paths each involve a *minimum number of predicates*. On the contrary, if a path involves a large number of predicates, it is less likely that all the predicates simultaneously hold, thereby making the path infeasible. They have statistically demonstrated the efficacy of the strategy.

McCabe's [16] *cyclomatic complexity* measure (Table 3.3) gives an interesting graph-theoretic interpretation of a program flow graph. If we consider cyclomatic complexity measures as paths in a flow graph, it is likely that a few infeasible paths will be constructed. The above discussion leads us to conclude that though the idea of statement coverage and branch coverage appear simple and straightforward, it is not easy to fully achieve those coverage criteria even for small programs.

4.9 SUMMARY

The notion of a path in a program unit is a fundamental concept. Assuming that a program unit is a function, a path is an executable sequence of instructions from the start of execution of the function to a return statement in the function. If there is no branching condition in a program unit, then there is just one path in the function. Generally, there are many branching conditions in a program unit, and thus there are numerous paths. One path differs from another path by at least one instruction. A path may contain one or more loops, but, ultimately, a path is expected to terminate its execution. Therefore, a path is of finite length in terms of number of instructions it executes. One can have a graphical representation of a program unit, called a control flow graph, to capture the concept of control flow in the program unit.

Each path corresponds to a distinct behavior of the program unit, and therefore we need to test each path with at least one test case. If there are a large number of paths in a program, a programmer may not have enough time to test all the paths. Therefore, there is a need to select a few paths by using some path selection criteria. A path selection criterion allows us to select a few paths to achieve a certain kind of coverage of program units. Some well-known coverage metrics are statement coverage, branch coverage, and predicate coverage. A certain number of paths are chosen from the CFG to achieve a desired degree of coverage of a program unit. At an abstract level, each path is composed of a sequence of predicates and assignment (computation) statements. The predicates can be functions of local variables, global

variables, and constants, and those are called path predicates. All the predicates along the path must evaluate to true when control reaches the predicates for a path to be executable. One must select inputs, called path forcing inputs, such that the path predicates evaluate to true in order to be able to execute the path. The process of selecting path forcing inputs involves transforming the path predicates into a form that is void of local variables. Such a form of path predicates is called a path predicate expression. A path predicate expression is solely composed of the input vector and possibly a vector of constants. One can generate values of the input vector, which is considered as a test case, to exercise a path by solving the corresponding path predicate expression. Tools are being designed for generating test inputs from program units.

If a program unit makes function calls, it is possible that the path predicates are functions of the values returned by those functions. In such a case, it may be difficult to solve a path predicate expression to generate test cases. Path testing is more applicable to lower level program units than to upper level program units containing many function calls.

LITERATURE REVIEW

Clarke [3] describes an automated system to generate test data from FORTRAN programs. The system is based on the idea of selecting program paths, identifying path conditions, and solving those conditions to generate inputs. When the program is executed with the selected inputs, the corresponding paths are executed. Automatically generating test inputs is a difficult task. The general problem of test generation from source code is an unsolvable problem. To mitigate the problem, there have been suggestions to select paths in certain ways. For example, select paths that execute loops for a restricted number of times. Similarly, select paths that are restricted to a maximum statement count. This is because longer paths are likely to have more predicates and are likely to be more complex. The system generates test inputs for paths that can be described by a set of linear path constraints.

The students are encouraged to read the tutorial by J. C. Huang entitled "An Approach to Program Testing," *ACM Computing Surveys*, Vol. 8, No. 3, September 1975, pp. 113–128. This article discusses a method for determining path conditions to enable achievement of branch coverage. It introduces the reader to the predicate calculus notation for expressing path conditions.

Ramamoorthy, Ho, and Chen [7] discuss the usefulness of symbolic substitution in generating path predicates for testing a path. Array referencing is a major problem in symbolic substitution because index values may not be known during symbolic execution. References to arrays are recorded in a table while performing symbolic execution, and ambiguities are resolved when test input are generated to evaluate the subscript expressions. Another major problem is determination of the number of times to execute a loop.

Considering that symbolic execution requires complex algebraic manipulations, Korel [17] suggested an alternative idea based on actual execution of the program under test, function minimization methods, and data flow analysis. Test

data are gathered for the program using concrete values of the input variables. A program's control flow is monitored while executing the program. If an execution, that is, a program path, is an undesirable one, then function minimization algorithms are used to locate the values of input variables which caused the undesirable path to be executed. In this approach, values of array indexes and pointers are known at each step of program execution. Thus, this approach helps us in overcoming the difficulties in handling arrays and pointers.

An excellent book on path-based program testing is *Software Testing Techniques* by Beizer [5]. The reader can find a more through treatment of the subject in the said book.

The test tool from *ParaSoft* [9] allows programmers to perform flow-based testing of program units written in C, C++, and Java. If a program unit under test calls another program unit, the tool generates a stub replacing the called unit. If a programmer wants to control what return values are used, he or she can create a stub table specifying the input–outcome mapping.

REFERENCES

1. F. E. Allen and J. Cocke. Graph Theoretic Constructs for Program Control Flow Analysis, Technical Report RC3923. IBM T. J. Watson Research Center, New York, 1972.
2. J. B. Goodenough and S. L. Gerhart. Toward a Theory of Test data Selection. *IEEE Transactions on Software Engineering*, June 1975, pp. 26–37.
3. L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, September 1976, pp. 215–222.
4. W. E. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, September 1976, pp. 38–45.
5. B. Beizer. *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, New York, 1990.
6. G. J. Myers. *The Art of Software Testing*. Wiley, New York, 1979.
7. C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the Automated Generation of Program Test Data. *IEEE Transactions on Software Engineering*, December 1976, pp. 293–300.
8. H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, December 1997, pp. 366–427.
9. Parasoft Corporation, available: http://www.parasoft.com/. Parasoft Application Development Quality Solution, 1996-2008.
10. P. M. Herman. A Data Flow Analysis Approach to Program Testing. *Australian Computer Journal*, November 1976, pp. 92–96.
11. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, July 1976, pp. 385–394.
12. M. R. Woodward, D. Hedley, and M. A. Hennell. Experience with Path Analysis and Testing of Programs. *IEEE Transactions on Software Engineering*, May 1980, pp. 278–286.
13. J. R. Brown and E. C. Nelson. Functional programming, TRW Defence and Space Systems Group for Rome Air Development Center, Technical Report on Contract F30602-76-C-0315, July 1977.
14. A. Bertolino and M. Marre. Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs. *IEEE Transactions on Software Engineering*, December 1994, pp. 885–899.
15. D. F. Yates and N. Malevris. Reducing the Effects of Infeasible Paths in Branch Testing. *ACM SIGSOFT Software Engineering Notes*, December 1989, pp. 48–54.
16. T. J. McCabe. A complexity Measure. *IEEE Transactions on Software Engineering*, December 1976, pp. 308–320.
17. B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, August 1990, pp. 870–879.

```
int binsearch(int X, int V[], int n){
   int low, high, mid;
   low = 0;
   high = n - 1;
   while (low <= high) {
      mid = (low + high)/2;
      if (X < V[mid])
           high = mid - 1;
      else if (X > V[mid])
           low = mid + 1;
      else
           return mid;
    }
    return -1;
}
```

Figure 4.17 Binary search routine.

Exercises

You are given the binary search routine in C shown in Figure 4.17. The input array V is assumed to be sorted in ascending order, n is the array size, and you want to find the index of an element X in the array. If X is not found in the array, the routine is supposed to return -1.

The first eight questions refer to the binary search() function.

1. Draw a CFG for binsearch().
2. From the CFG, identify a set of entry–exit paths to satisfy the complete statement coverage criterion.
3. Identify additional paths, if necessary, to satisfy the complete branch coverage criterion.
4. For each path identified above, derive their path predicate expressions.
5. Solve the path predicate expressions to generate test input and compute the corresponding expected outcomes.
6. Are all the selected paths feasible? If not, select and show that a path is infeasible, if it exists.
7. Can you introduce two faults in the routine so that these go undetected by your test cases designed for complete branch coverage?
8. Suggest a general way to detect the kinds of faults introduced in the previous step.
9. What are the limitations of control flow–based testing?
10. Show that branch coverage includes statement coverage.

CHAPTER 5

Data Flow Testing

An error does not become truth by reason of multiplied propagation, nor does truth become error because nobody sees it.
— *Mohandas Karamchand Gandhi*

5.1 GENERAL IDEA

A program unit, such as a function, accepts input values, performs computations while assigning new values to local and global variables, and, finally, produces output values. Therefore, one can imagine a kind of "flow" of data values between variables along a path of program execution. A data value computed in a certain step of program execution is expected to be used in a later step. For example, a program may open a file, thereby obtaining a value for a file pointer; in a later step, the file pointer is expected to be used. Intuitively, if the later use of the file pointer is never verified, then we do not know whether or not the earlier assignment of value to the file pointer variable is all right. Sometimes, a variable may be defined twice without a use of the variable in between. One may wonder why the first definition of the variable is never used. There are two motivations for data flow testing as follows. First, a memory location corresponding to a program variable is accessed in a desirable way. For example, a memory location may not be read before writing into the location. Second, it is desirable to verify the correctness of a data value generated for a variable—this is performed by observing that all the uses of the value produce the desired results.

The above basic idea about data flow testing tells us that a programmer can perform a number of tests on data values, which are collectively known as data flow testing. Data flow testing can be performed at two conceptual levels: *static data flow testing* and *dynamic data flow testing*. As the name suggests, static data flow testing is performed by analyzing the source code, and it does not involve actual execution of source code. Static data flow testing is performed to reveal potential defects in programs. The potential program defects are commonly known as *data*

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

flow anomaly. On the other hand, dynamic data flow testing involves identifying program paths from source code based on a class of *data flow testing criteria*.

The reader may note that there is much similarity between control flow testing and data flow testing. Moreover, there is a key difference between the two approaches. The similarities stem from the fact that both approaches identify program paths and emphasize on generating test cases from those program paths. The difference between the two lies in the fact that control flow test selection criteria are used in the former, whereas data flow test selection criteria are used in the latter approach.

In this chapter, first we study the concept of data flow anomaly as identified by Fosdick and Osterweil [1]. Next, we discuss dynamic data flow testing in detail.

5.2 DATA FLOW ANOMALY

An anomaly is a deviant or abnormal way of doing something. For example, it is an abnormal situation to successively assign two values to a variable without using the first value. Similarly, it is abnormal to use a value of a variable before assigning a value to the variable. Another abnormal situation is to generate a data value and never use it. In the following, we explain three types of abnormal situations concerning the generation and use of data values. The three abnormal situations are called **type 1**, **type 2**, and **type 3** anomalies [1]. These anomalies could be manifestations of potential programming errors. We will explain why program anomalies need not lead to program failures.

Defined and Then Defined Again (Type 1): Consider the partial sequence of computations shown in Figure 5.1, where f1(y) and f2(z) denote functions with the inputs y and z, respectively. We can interpret the two statements in Figure 5.1 in several ways as follows:

- The computation performed by the first statement is redundant if the second statement performs the intended computation.
- The first statement has a fault. For example, the intended first computation might be w = f1(y).
- The second statement has a fault. For example, the intended second computation might be v = f2(z).
- A fourth kind of fault can be present in the given sequence in the form of a missing statement between the two. For example, v = f3(x) may be the desired statement that should go in between the two given statements.

```
   :
x = f1(y)
x = f2(z)
   :
```

Figure 5.1 Sequence of computations showing data flow anomaly.

It is for the programmer to make the desired interpretation, though one can interpret the given two statements in several ways, However, it can be said that there is a data flow anomaly in those two statements, indicating that those need to be examined to eliminate any confusion in the mind of a code reader.

Undefined but Referenced (Type 2): A second form of data flow anomaly is to use an undefined variable in a computation, such as $x = x - y - w$, where the variable w has not been *initialized* by the programmer. Here, too, one may argue that though w has not been initialized, the programmer intended to use another initialized variable, say y, in place of w. Whatever may be the real intention of the programmer, there exists an anomaly in the use of the variable w, and one must eliminate the anomaly either by initializing w or replacing w with the intended variable.

Defined but Not Referenced (Type 3): A third kind of data flow anomaly is to define a variable and then to undefine it without using it in any subsequent computation. For example, consider the statement $x = f(x, y)$ in which a new value is assigned to the variable x. If the value of x is not used in any subsequent computation, then we should be suspicious of the computation represented by $x = f(x, y)$. Hence, this form of anomaly is called "defined but not referenced."

Huang [2] introduced the idea of "states" of program variables to identify data flow anomaly. For example, initially, a variable can remain in an "undefined" (U) state, meaning that just a memory location has been allocated to the variable but no value has yet been assigned. At a later time, the programmer can perform a computation to define (d) the variable in the form of assigning a value to the variable—this is when the variable moves to a "defined but not referenced" (D) state. At a later time, the programmer can reference (r), that is, read, the value of the variable, thereby moving the variable to a "defined and referenced" state (R). The variable remains in the R state as long as the programmer keeps referencing the value of the variable. If the programmer assigns a new value to the variable, the variable moves back to the D state. On the other hand, the programmer can take an action to undefine (u) the variable. For example, if an opened file is closed, the value of the file pointer is no more recognized by the underlying operating system, and therefore the file pointer becomes undefined. The above scenarios describe the normal actions on variables and are illustrated in Figure 5.2.

However, programmers can make mistakes by taking the wrong actions while a variable is in a certain state. For example, if a variable is in the state U—that is, the variable is still undefined—and a programmer reads (r) the variable, then the variable moves to an abnormal (A) state. The abnormal state of a variable means that a programming anomaly has occurred. Similarly, while a variable is in the state D and the programmer undefines (u) the variable or redefines (d) the variable, then the variable moves to the abnormal (A) state. Once a variable enters the abnormal state, it remains in that state irrespective of what action—d, u, or r—is taken. The actions that take a variable from a desired state, such as U or D, to an abnormal state are illustrated in Figure 5.2.

Legend:

States	Actions
U: Undefined	*d*: Define
D: Defined but not referenced	*r*: Reference
R: Defined and referenced	*u*: Undefine
A: Abnormal	

Figure 5.2 State transition diagram of a program variable. (From ref. 2. © 1979 IEEE.)

Now it is useful to make an association between the **type 1**, **type 2**, and **type 3** anomalies and the state transition diagram shown in Figure 5.2. The **type 1**, **type 2**, and **type 3** anomalies are denoted by the action sequences *dd*, *ur*, and *du*, respectively, in Figure 5.2.

Data flow anomaly can be detected by using the idea of *program instrumentation*. Intuitively, program instrumentation means incorporating additional code in a program to monitor its execution status. For example, we can write additional code in a program to monitor the sequence of *states*, namely the *U*, *D*, *R*, and *A*, traversed by a variable. If the state sequence contains the *dd*, *ur*, and *du* subsequence, then a data flow anomaly is said to have occurred.

The presence of a data flow anomaly in a program does not necessarily mean that execution of the program will result in a failure. A data flow anomaly simply means that the program *may* fail, and therefore the programmer must investigate the cause of the anomaly. Let us consider the *dd* anomaly shown in Figure 5.1. If the real intention of the programmer was to perform the second computation and the first computation produces no side effect, then the first computation merely represents a waste of processing power. Thus, the said *dd* anomaly will not lead to program failure. On the other hand, if a statement is missing in between the two statements, then the program can possibly lead to a failure. The programmers must analyze the causes of data flow anomalies and eliminate them.

5.3 OVERVIEW OF DYNAMIC DATA FLOW TESTING

In the process of writing code, a programmer manipulates variables in order to achieve the desired computational effect. Variable manipulation occurs in several ways, such as initialization of the variable, assignment of a new value to the

variable, computing a value of another variable using the value of the variable, and controlling the flow of program execution.

Rapps and Weyuker [3] convincingly tell us that one should not feel confident that a variable has been *assigned the correct value* if no test case causes the execution of a path from the assignment to a point where the value of the variable is *used*. In the above motivation for data flow testing, (i) assignment of a correct value means whether or not a value for the variable has been correctly generated and (ii) use of a variable refers to further generation of values for the same or other variables and/or control of flow. A variable can be used in a predicate, that is, a condition, to choose an appropriate flow of control.

The above idea gives us an indication of the involvement of certain kinds of program paths in data flow testing. Data flow testing involves selecting entry–exit paths with the objective of covering certain data definition and use patterns, commonly known as data flow testing criteria. Specifically, certain program paths are selected on the basis of data flow testing criteria. Following the general ideas in control flow testing that we discussed in Chapter 4, we give an outline of performing data flow testing in the following:

- Draw a data flow graph from a program.
- Select one or more data flow testing criteria.
- Identify paths in the data flow graph satisfying the selection criteria.
- Derive path predicate expressions from the selected paths and solve those expressions to derive test input.

The reader may recall that the process of deriving a path predicate expression from a path has been explained in Chapter 4. The same idea applies to deriving a path predicate expression from a path obtained from a data flow graph. Therefore, in the rest of this chapter we will explain a procedure for drawing a data flow graph from a program unit, and discuss data flow testing criteria.

5.4 DATA FLOW GRAPH

In this section, we explain the main ideas in a data flow graph and a method to draw it. In practice, programmers may not draw data flow graphs by hand. Instead, language translators are modified to produce data flow graphs from program units. A data flow graph is drawn with the objective of identifying data definitions and their uses as motivated in the preceding section. Each occurrence of a data variable is classified as follows:

Definition: This occurs when a value is moved into the memory location of the variable. Referring to the C function VarTypes() in Figure 5.3, the assignment statement i = x; is an example of definition of the variable *i*.

Undefinition or Kill: This occurs when the value and the location become unbound. Referring to the C function VarTypes(), the first

```
(iptr = malloc(sizeof(int));
```

```
int VarTypes(int x, int y){
     int i;
     int  *iptr;
     i  = x;
     iptr  = malloc(sizeof(int));
     *iptr = i + x;
     if (*iptr > y)
         return (x);
     else {
         iptr  = malloc(sizeof(int));
         *iptr = x + y;
         return(*iptr);
         }
}
```

Figure 5.3 Definition and uses of variables.

statement initializes the integer pointer variable iptr and

```
iptr = i + x;
```

initializes the value of the location pointed to by iptr. The second

```
iptr = malloc(sizeof(int));
```

statement redefines variable iptr, thereby undefining the location previously pointed to by iptr.

Use: This occurs when the value is fetched from the memory location of the variable. There are two forms of *uses* of a variable as explained below.

- **Computation use (c-use):** This directly affects the computation being performed. In a c-use, a potentially new value of another variable or of the same variable is produced. Referring to the C function VarTypes(), the statement

  ```
  *iptr = i + x;
  ```

  gives examples of c-use of variables i and x.

- **Predicate use (p-use):** This refers to the use of a variable in a predicate controlling the flow of execution. Referring to the C function VarTypes(), the statement

  ```
  if (*iptr > y) ...
  ```

  gives examples of p-use of variables y and iptr.

A data flow graph is a directed graph constructed as follows:

- A sequence of *definitions* and c-uses is associated with each node of the graph.
- A set of p-uses is associated with each edge of the graph.

- The entry node has a definition of each parameter and each nonlocal variable which occurs in the subprogram.
- The exit node has an *undefinition* of each local variable.

Example: We show the data flow graph in Figure 5.4 for the ReturnAverage() example discussed in Chapter 4, The initial node, node **1**, represents initialization of the input vector < value, AS, MIN, MAX > . Node **2** represents the initialization of the four local variables *i*, ti, tv, and sum in the routine. Next we introduce a NULL node, node **3**, keeping in mind that control will come back to the beginning of the while loop. Node **3** also denotes the fact that program control exits from the while loop at the NULL node. The statement ti++ is represented by node **4**. The predicate associated with edge **(3, 4)** is the condition part of the while loop, namely,

```
((ti < AS) && (value[i] != -999))
```

Statements tv++ and sum = sum + value[i] are represented by node **5**. Therefore, the condition part of the first if statement forms the predicate associated with edge **(4, 5)**, namely,

Figure 5.4 Data flow graph of ReturnAverage() example.

```
((value[i] >= MIN) && (value[i] <= MAX))
```

The statement i++ is represented by node **6**. The predicate associated with edge **(4, 6)** is the negation of the condition part of the if statement, namely,

```
((value[i] >= MIN) && (value[i] <= MAX)).
```

The predicate associated with edge **(5, 6)** is true because there is an unconditional flow of control from node **5** to node **6**. Execution of the while loop terminates when its condition evaluates to false. Therefore, the predicate associated with edge **(3, 7)** is the negation of the predicate associated with edge **(3, 4)**, namely,

```
~((ti < AS) && (value[i] != -999))
```

It may be noted that there is no computation performed in a NULL node. Referring to the second if statement, av = (double) − 999 is represented by node **8**, and av = (double) sum/tv is represented by node **9**. Therefore, the predicate associated with edge **(7, 9)** is

```
(tv > 0),
```

and the predicate associated with edge **(7, 8)** is

```
~(tv > 0).
```

Finally, the return(av) statement is represented by node **10**, and the predicate True is associated with both the edges **(7, 8)** and **(7, 9)**.

5.5 DATA FLOW TERMS

A variable *defined* in a statement is *used* in another statement which may occur immediately or several statements after the definition. We are interested in finding *paths* that include pairs of definition and use of variables. In this section, we explain a family of path selection criteria that allow us to select paths with varying strength. The reader may note that for every feasible path we can generate a test case. In the following, first we explain a few terms, and then we explain a few selection criteria using those terms.

Global c-use: A c-use of a variable x in node i is said to be a *global c-use* if x has been defined before in a node other than node i.

Example: The c-use of variable tv in node **9** is a global c-use since tv has been defined in nodes **2** and **5** (Figure 5.4).

Definition Clear Path: A path $(i - n_1 - \cdots - n_m - j)$, $m \geq 0$, is called a definition clear path (def-clear path) with respect to variable x

- from node i to node j and
- from node i to edge (n_m, j)

if x has been neither *defined* nor *undefined* in nodes $n_1, \ldots, n_m$. The reader may note that the definition of a def-clear path is unconcerned about the status of x in nodes i and j. Also, a def-clear path does not preclude loops. Therefore, the path **2-3-4-6-3-4-6-3-4-5**, which includes a loop, is a def-clear path.

Example: The paths **2-3-4-5** and **2-3-4-6** are def-clear paths with respect to variable tv from node **2** to **5** and from node **2** to **6**, respectively (Figure 5.4).

Global Definition: A node i has a global definition of a variable x if node i has a definition of x and there is a def-clear path with respect to x from node i to some

- node containing a global c-use or
- edge containing a p-use of

variable x. The reader may note that we do not define *global* p-use of a variable similar to global c-use. This is because every p-use is associated with an edge—and not a node.

In Table 5.1, we show all the global definitions and global c-uses appearing in the data flow graph of Figure 5.4; def(i) denotes the set of variables which have global definitions in node i. Similarly, c-use(i) denotes the set of variables which have global c-uses in node i. We show all the predicates and p-uses appearing in the data flow graph of Figure 5.4 in Table 5.2; predicate(i,j) denotes the predicate associated with edge *(i, j)* of the data flow graph in Figure 5.4; p-use(i, j) denotes the set of variables which have p-uses on edge *(i, j)*.

Simple Path: A simple path is a path in which all nodes, except possibly the first and the last, are distinct.

TABLE 5.1 Def() and c-use() Sets of Nodes in Figure 5.4

Nodes i	def(i)	c-use(i)
1	{value, AS, MIN, MAX}	{}
2	{i, ti, tv, sum}	{}
3	{}	{}
4	{ti}	{ti}
5	{tv, sum}	{tv, i, sum, value}
6	{i}	{i}
7	{}	{}
8	{av}	{}
9	{av}	{sum, tv}
10	{}	{av}

TABLE 5.2 Predicates and p-use() Set of Edges in Figure 5.4

Edges (i, j)	predicate(i, j)	p-use(i, j)
(1, 2)	True	{}
(2, 3)	True	{}
(3, 4)	(ti < AS) && (value[i] ! = − 999)	{i, ti, AS, value}
(4, 5)	(value[i] < = MIN) && (value[i] > = MAX)	{i, MIN, MAX, value}
(4, 6)	~((value[i] < = MIN) && (value[i] > = MAX))	{i, MIN, MAX, value}
(5, 6)	True	{}
(6, 3)	True	{}
(3, 7)	~((ti < AS) && (value[i] ! = − 999))	{i, ti, AS, value}
(7, 8)	~(tv > 0)	{tv}
(7, 9)	(tv > 0)	{tv}
(8, 10)	True	{}
(9, 10)	True	{}

Example: Paths **2-3-4-5** and **3-4-6-3** are simple paths (Figure 5.4).

Loop-Free Path: A loop-free path is a path in which *all* nodes are distinct.

Complete Path: A complete path is a path from the entry node to the exit node.

Du-path: A path $(n_1 - n_2 - \cdots - n_j - n_k)$ is a definition-use path (du-path) with respect to (w.r.t) variable x if node n_1 has a global definition of x and *either*

- node n_k has a global c-use of x and $(n_1 - n_2 - \cdots - n_j - n_k)$ is a def-clear simple path w.r.t. x *or*
- edge (n_j, n_k) has a p-use of x and $(n_1 - n_2 - \cdots - n_j)$ is a def-clear, loop-free path w.r.t. x.

Example: Considering the global definition and global c-use of variable tv in nodes **2** and **5**, respectively, **2-3-4-5** is a du-path.

Example: Considering the global definition and p-use of variable tv in nodes **2** and on edge **(7, 9)**, respectively, **2-3-7-9** is a du-path.

5.6 DATA FLOW TESTING CRITERIA

In this section, we explain seven types of data flow testing criteria. These criteria are based on two fundamental concepts, namely, definitions and uses—both c-uses and p-uses—of variables.

All-defs: For each variable x and for each node i such that x has a global definition in node i, select a complete path which includes a def-clear path from node i to

- node j having a global c-use of x or
- edge (j,k) having a p-use of x.

Example: Consider the variable tv, which has global definitions in nodes **2** and **5** (Figure 5.4 and Tables 5.1 and 5.2). First, we consider its global definition in node **2**. We find a global c-use of tv in node **5**, and there exists a def-clear path **2-3-4-5** from node **2** to node **5**. We choose a complete path **1-2-3-4-5-6-3-7-9-10** that includes the def-clear path **2-3-4-5** to satisfy the all-defs criterion. We also find p-uses of variable tv on edge **(7, 8)**, and there exists a def-clear path **2-3-7-8** from node **2** to edge **(7, 8)**. We choose a complete path **1-2-3-7-8-10** that includes the def-clear path **2-3-7-8** to satisfy the all-defs criterion. Now we consider the definition of tv in node **5**. In node **9** there is a global c-use of tv, and in edges **(7, 8)** and **(7, 9)** there are p-uses of tv. There is a def-clear path **5-6-3-7-9** from node **5** to node **9**. Thus, we choose a complete path **1-2-3-4-5-6-3-7-9-10** that includes the def-clear path **5-6-3-7-9** to satisfy the all-defs criterion. The reader may note that the complete path **1-2-3-4-5-6-3-7-9-10** covers the all-defs criterion for variable tv defined in nodes **2** and **5**. To satisfy the all-defs criterion, similar paths must be obtained for variables i, ti, and sum.

All-c-uses: For each variable x and for each node i, such that x has a global definition in node i, select complete paths which include def-clear paths from node i to *all* nodes j such that there is a global c-use of x in j.

Example: Let us obtain paths to satisfy the all-c-uses criterion with respect to variable ti. We find two global definitions of ti in nodes **2** and **4**. Corresponding to the global definition in node **2**, there is a global c-use of ti in node **4**. However, corresponding to the global definition in node **4**, there is no global c-use of ti. From the global definition in node **2**, there is a def-clear path to the global c-use in node **4** in the form of **2-3-4**. The reader may note that there are *four* complete paths that include the def-clear path **2-3-4** as follows:

1-2-3-4-5-6-3-7-8-10,

1-2-3-4-5-6-3-7-9-10,

1-2-3-4-6-3-7-8-10, and

1-2-3-4-6-3-7-9-10.

One may choose one or more paths from among the four paths above to satisfy the all-c-uses criterion with respect to variable ti.

All-p-uses: For each variable x and for each node i such that x has a global definition in node i, select complete paths which include def-clear paths from node i to *all* edges (j,k) such that there is a p-use of x on edge (j,k).

Example: Let us obtain paths to satisfy the all-p-uses criterion with respect to variable tv. We find two global definitions of tv in nodes **2** and **5**. Corresponding to the global definition in node **2**, there is a p-use of tv on edges **(7, 8)** and **(7, 9)**. There are def-clear paths from node **2** to edges **(7, 8)** and **(7, 9)**, namely **2-3-7-8** and **2-3-7-9**, respectively. Also, there are def-clear paths from node **5** to edges

(7, 8) and **(7, 9)**, namely, **5-6-3-7-8** and **5-6-3-7-9**, respectively. In the following, we identify *four* complete paths that include the above four def-clear paths:

1-2-3-7-8-10,

1-2-3-7-9-10,

1-2-3-4-5-6-3-7-8-10, and

1-2-3-4-5-6-3-7-9-10.

All-p-uses/Some-c-uses: This criterion is identical to the all-p-uses criterion except when a variable x has no p-use. If x has no p-use, then this criterion reduces to the some-c-uses criterion explained below.

Some-c-uses: For each variable x and for each node i such that x has a global definition in node i, select complete paths which include def-clear paths from node i to *some* nodes j such that there is a global c-use of x in node j.

Example: Let us obtain paths to satisfy the all-p-uses/some-c-uses criterion with respect to variable i. We find two global definitions of i in nodes **2** and **6**. There is no p-use of i in Figure 5.4. Thus, we consider some c-uses of variable i. Corresponding to the global definition of variable i in node **2**, there is a global c-use of i in node **6**, and there is a def-clear path from node **2** to node **6** in the form of **2-3-4-5-6**. Therefore, to satisfy the all-p-uses/some-c-uses criterion with respect to variable i, we select the complete path **1-2-3-4-5-6-3-7-9-10** that includes the def-clear path **2-3-4-5-6**.

All-c-uses/Some-p-uses: This criterion is identical to the all-c-uses criterion except when a variable x has no global c-use. If x has no global c-use, then this criterion reduces to the some-p-uses criterion explained below.

Some-p-uses: For each variable x and for each node i such that x has a global definition in node i, select complete paths which include def-clear paths from node i to *some* edges (j,k) such that there is a p-use of x on edge (j,k).

Example: Let us obtain paths to satisfy the all-c-uses/some-p-uses criterion with respect to variable AS. We find just one global definition of AS in node **1**. There is no global c-use of AS in Figure 5.4. Thus, we consider some p-uses of AS. Corresponding to the global definition of AS in node **1**, there are p-uses of AS on edges **(3, 7)** and **(3, 4)**, and there are def-clear paths from node **1** to those two edges, namely, **1-2-3-7** and **1-2-3-4**, respectively. There are many complete paths that include those two def-clear paths. One such example path is given as **1-2-3-4-5-6-3-7-9-10.**

All-uses: This criterion is the conjunction of the all-p-uses criterion and the all-c-uses criterion discussed above.

All-du-paths: For each variable x and for each node i such that x has a global definition in node i, select complete paths which include *all* du-paths from node i

- to *all* nodes j such that there is a global c-use of x in j and
- to *all* edges (j,k) such that there is a p-use of x on (j,k).

In Chapter 4, we explained a procedure to generate a test input from an entry–exit program path. There is much similarity between the control flow–based testing and the data flow–based testing. Their difference lies in the ways the two techniques select program paths.

5.7 COMPARISON OF DATA FLOW TEST SELECTION CRITERIA

Having seen a relatively large number of test selection criteria based on the concepts of data flow and control flow, it is useful to find relationships among them. Given a pair of test selection criteria, we should be able to compare the two. If we cannot compare them, we realize that they are incomparable. Rapps and Weyuker [3] defined the concept of an *includes* relationship to find out if, for a given pair of selection criteria, one includes the other. In the following, by a *complete* path we mean a path from the entry node of a flow graph to one of its exit nodes.

Definition: Given two test selection criteria c_1 and c_2, c_1 *includes* c_2 if for every def/use graph any set of complete paths of the graph that satisfies c_1 also satisfies c_2.

Definition: Given two test selection criteria c_1 and c_2, c_1 *strictly includes* c_2, denoted by $c_1 \rightarrow c_2$, provided c_1 includes c_2 and for some def/use graph there is a set of complete paths of the graph that satisfies c_2 but not c_1.

It is easy to note that the "$\rightarrow$" relationship is a transitive relation. Moreover, given two criteria c_1 and c_2, it is possible that neither $c_1 \rightarrow c_2$ nor $c_2 \rightarrow c_1$ holds, in which case we call the two criteria *incomparable*. Proving the strictly includes relationship or the incomparable relationship between two selection criteria in a programming language with arbitrary semantics may not be possible. Thus, to show the strictly includes relationship between a pair of selection criteria, Rapps and Weyuker [3] have considered a restricted programming language with the following syntax:

Start statement:	**start**
Input statement:	**read** $x_1, \ldots, x_n$, where $x_i, \ldots, n$ are variables.
Assignment statement:	$y \leftarrow f(x_1, \ldots, x_n)$, where y, $x_i, \ldots, n$ are variables, and f is a function.
Output statement:	**print** $e_1, \ldots, e_n$, where $e_1, \ldots, e_n$ are output values.
Unconditional transfer statement:	**goto** m, where m is a label.
Conditional transfer statement:	**if** $p(x_1, \ldots, x_n)$, **then goto** m, where p is a predicate.
Halt statement:	**stop**

Figure 5.5 Relationship among DF (data flow) testing criteria. (From ref. 4. © 1988 IEEE.)

Frankl and Weyuker [4] have further extended the relationship; what they have proved has been summarized in Figure 5.5. For example, the all-paths selection criterion strictly includes the all-du-paths criterion. Similarly, the all-c-uses/some-p-uses criterion strictly includes the all-defs criterion.

However, we cannot find a strictly includes relationship between the pair all-c-uses and all-p-uses. Let P_x^c be a set of paths selected by the all-c-uses criterion with respect to a variable x. Now we cannot say with certainty whether or not the path set P_x^c satisfies the all-p-uses criterion with respect to the same variable x. Similarly, let P_x^p be a set of paths selected by the all-p-uses criterion with respect to the variable x. Now we cannot say with certainty whether or not the path set P_x^p satisfies the all-c-uses criterion with respect to the same variable x. Thus, the two criteria all-c-uses and all-p-uses are incomparable.

Note the relationship between data flow–based test selection criteria and control flow–based test selection criteria, as shown in Figure 5.5. The two control flow–based test selection criteria in Figure 5.5 are all-branches and all-statements. The all-p-uses criterion strictly includes the all-branches criterion, which implies that one can select more paths from a data flow graph of a program unit than from its control flow graph.

5.8 FEASIBLE PATHS AND TEST SELECTION CRITERIA

Given a data flow graph, a path is a sequence of nodes and edges. A complete path is a sequence of nodes and edges starting from the initial node of the graph to one of its exit nodes. A complete path is executable if there exists an assignment

of values to input variables and global variables such that all the path predicates evaluate to true, thereby making the path executable. *Executable* paths are also known as *feasible* paths. If no such assignment of values to input variables and global variables exists, then we call the path *infeasible* or *inexecutable*.

Since we are interested in selecting inputs to execute paths, we must ensure that a test selection criterion picks executable paths. Assume that we want to test a program by selecting paths to satisfy a certain selection criterion C. Let P_C be the set of paths selected according to criterion C for a given program unit. As an extreme example, if all the paths in P_C are infeasible, then the criterion C has not helped us in any way. For a criterion C to be useful, it must select a set of executable, or feasible, paths. Frankl and Weyuker [4] have modified the definitions of the test selection criteria so that each criterion selects only feasible paths. In other words, we modify the definition of criterion C to obtain a criterion $C*$ which selects only feasible paths, and $C*$ is called a feasible data flow (FDF) testing criterion. As an example, the criterion (All-c-uses)* is an adaptation of All-c-uses such that only feasible paths are selected by (All-c-uses)*, as defined below.

> *(All-c-uses)**: For each variable x and for each node i, such that x has a global definition in node i, select feasible complete paths which include def-clear paths from node i to *all* nodes j such that there is a global c-use of x in j.

Thus, test selection criteria (All-paths)*, (All-du-paths)*, (All-uses)*, (All-c-uses/Some-p-uses)*, (All-p-uses/Some-c-uses)*, (All-c-uses)*, (All-p-uses)*, (All-defs)*, (All-branches)*, and (All-statements)* choose only feasible paths, and, therefore, these are called feasible data flow (FDF) testing criteria. Frankl and Weyuker [4] have shown that the strictly includes relationships among test selection criteria, as shown in Figure 5.5, do not hold if the selection criteria choose only feasible paths. The new relationship among FDF test selection criteria is summarized in Figure 5.6. Though it is seemingly useful to select only feasible paths, and therefore consider only the FDF test selection criteria, we are faced with the decidability problem. More specifically, it is undecidable to know if a given set of paths is executable. We cannot automate the application of an FDF test selection criterion, if we do not know the executability of the path. On the other hand, a data flow testing criterion may turn out to be inadequate if all its selected paths are infeasible, in which case the criterion is considered to be inadequate. Consequently, a test engineer must make a choice between using an inadequate selection criterion and one that cannot be completely automated.

5.9 COMPARISON OF TESTING TECHNIQUES

So far we have discussed two major techniques for generating test data from source code, namely control flow–based path selection and data flow–based path selection. We also explained a few criteria to select paths from a control flow graph and data flow graph of a program. Programmers often randomly select test data based on their own understanding of the code they have written. Therefore, it is natural to

Figure 5.6 Relationship among FDF (feasible data flow) testing criteria. (From ref. 4. © 1988 IEEE.)

compare the effectiveness of the three test generation techniques, namely random test selection, test selection based on control flow, and test selection based on data flow. Comparing those techniques does not seem to be an easy task. An acceptable, straightforward way of comparing them is to apply those techniques to the same set of programs with known faults and express their effectiveness in terms of the following two metrics:

- Number of test cases produced
- percentage of known faults detected

Ntafos [5] has reported on the results of an experiment comparing the effectiveness of three test selection techniques. The experiment involved seven mathematical programs with known faults. For the control flow–based technique, the *branch coverage* criterion was selected, whereas the *all-uses* criterion was chosen for data flow testing. Random testing was also applied to the programs. The data flow testing, branch testing, and random testing detected 90%, 85.5%, and 79.5%, respectively, of the known defects. A total of 84 test cases were designed to achieve all-uses coverage, 34 test cases were designed to achieve branch coverage, and 100 test cases were designed in the random testing approach. We interpret the experimental results as follows:

- A programmer can randomly generate a large number of test cases to find most of the faults. However, one will run out of test cases to find some of the remaining faults. Random testing does not look to be ineffective, but

Figure 5.7 Limitation of different fault detection techniques.

it incurs higher costs than the systematic techniques, namely, the control flow and the data flow techniques.

- Test selection based on branch coverage produces much fewer test cases than the random technique, but achieves nearly the same level of fault detection as the random technique. Thus, there is significant saving in the cost of program testing.
- The all-uses testing criterion gives a programmer a new way to design more test cases and reveal more faults than the branch coverage criterion.
- All these techniques have inherent limitations which prevent them from revealing all faults. Therefore, there is a need to use many different testing techniques and develop new techniques. This idea is depicted in Figure 5.7. Our goal is to reduce the gap between the total number of faults present in a program and the faults detected by various test generation techniques.

5.10 SUMMARY

Flow of data in a program can be visualized by considering the fact that a program unit accepts input data, transforms the input data through a sequence of computations, and, finally, produces the output data. Therefore, one can imagine data values to be flowing from one assignment statement defining a variable to another assignment statement or a predicate where the value is used.

Three fundamental actions associated with a variable are *undefine* (u), *define* (d), and *reference* (r). A variable is implicitly undefined when it is created without being assigned a value. On the other hand, a variable can be explicitly undefined. For example, when an opened file is closed, the variable holding the file pointer becomes undefined. We have explained the idea of "states" of a variable, namely, *undefined* (U), *defined* (D), *referenced* (R), and *abnormal* (A), by considering the three fundamental actions on a variable. The A state represents the fact that the variable has been accessed in an *abnormal* manner causing data flow anomaly.

Individual actions on a variable do not cause data flow anomaly. Instead, certain sequences of actions lead to data flow anomaly, and those three sequences of actions are *dd*, *ur*, and *du*. A variable continues to remain in the abnormal state irrespective of subsequent actions once it enters that state. The mere presence of data flow anomaly in a program may not lead to program failure. The programmer must investigate the cause of an anomaly and modify the code to eliminate it. For example, a missing statement in the code might have caused a *dd* anomaly, in which case the programmer needs to write new code.

The program path is a fundamental concept in testing. One test case can be generated from one executable path. The number of different paths selected for execution is a measure of the extent of testing performed. Path selection based on statement coverage and branch coverage lead to a small number of paths being chosen for execution. Therefore, there exists a large gap between control flow testing and exhaustive testing. The concept of data flow testing gives us a way to bridge the gap between control flow testing and exhaustive testing.

The concept of data flow testing gives us new selection criteria for choosing more program paths to test than what we can choose by using the idea of control flow testing. Specifically, the data flow test selection criteria are all-du-paths, all-defs, all-c-uses, all-p-uses, all-uses, all-c-uses/some-p-uses, and all-p-uses/some-c-uses. To compare two selection criteria, the concept of a strictly includes relationship is found to be useful.

LITERATURE REVIEW

Osterweil and Fosdick [6] have implemented a system, called DAVE, to analyze FORTRAN programs and detect *ur*, *dd*, and *du* types of data flow anomalies. DAVE detects those anomalies by performing a flow graph search for each variable in a given program unit. For programs with subprogram invocations, the system works in a bottom-up manner, that is, the called subprograms are analyzed before analyzing the caller.

Programmers need to be aware that it is difficult to apply the idea of data flow analysis to all kinds of data structures and program constructs. The analysis of arrays is one such difficulty. Fosdick and Osterweil [1] have noted that problems arise when different elements of the same array are acted upon in different ways, thereby giving rise to different patterns of definition, reference, and undefinition. Static data flow analysis systems, such as DAVE, do not evaluate index expressions and therefore cannot tell us what array element is being referenced in a given expression. Such systems try to get around this problem by treating an entire array as one single variable, rather than a set of different variables of the same type. Fosdick and Osterweil have shown that recursive programs pose difficulty in data flow analysis. A programming style that can pose a difficulty in data flow analysis is to pass a single variable as an argument more than once. This is because DAVE assumes that all subprogram parameters are distinct variables.

Laski and Korel [7] argue that data flow testing bridges the gap between branch testing and all-paths testing. On the one hand, in branch testing, one selects

a set of paths to cover all branches of the control flow graph of a program unit; one needs to select a small number of paths to satisfy the criterion. On the other hand, all-paths testing is the same as exhaustive testing. Data flow testing allows programmers to select many more paths than chosen by branch testing. Essentially, in data flow testing, loops are unfolded to exercise the definition–use pairs.

Herman [8] had a programmer apply data flow testing to a number of medium-sized program units of about 800 statements. It is interesting to note that faults detected during testing were usually found while attempting to devise test data to satisfy the chosen paths, rather than while examining the test run output. The fact that program faults were found during the process of test design is significant in the sense that system development and selection of tests can simultaneously be done in producing a better quality system.

The article by Ural [9] presents a method for generating test cases from the specifications of communications protocols given in the Estelle language. The method involves static data flow analysis of specifications. The method is summarized as follows: (i) transform a specification into a graph containing both the control flow and the data flow aspects; (ii) detect data flow anomalies in the specification; and (iii) generate test cases to cover all definition–use pairs.

The article by Ntafos [10] explains an extended overview of data flow testing strategies in terms of their relative coverage of a program's structure and the number of test cases needed to satisfy each strategy. In addition, the article extends the subsumption hierarchy introduced by Rapps and Weyuker [3] by including $\mathrm{TER}_n = 1$. For details about testing hierarchy levels, denoted by n above and test effectiveness ratio (TER), the reader is referred to the article by Woodward, Hedley, and Hennell [11].

The concept of selecting program paths based on data flow has been studied in different ways by different researchers, namely, Laski and Korel [7], Ntafos [5], and Rapps and Weyuker [3]. To facilitate the comparison and simplify the discussion, Clarke, Podgurski, Richardson, and Zeil [12] define all the data flow criteria using a single set of terms. They give a new subsumption hierarchy of the data flow test selection criteria by modifying the subsumption hierarchy of Rapps and Weyuker [7] shown in Figure 5.5.

Koh and Liu [3] have presented a two-step approach for generating paths that test both the control flow and the data flow in implementations of communication protocols based on the idea of extended finite-state machines. First, select a set of paths to cover a data flow selection criterion. Second, selectively augment the state transitions in the chosen set of paths with state check sequences so that control flow can be ensured and data flow coverage can be preserved. The test design methodology of Sarikaya, Bochmann, and Cerny [14] also generates paths to achieve joint coverage of control flow and data flow in the Estelle specifications of communication protocols.

Researchers have extended the classical data flow testing approach to the testing of object-oriented programs [15–18]. Harrold and Rothermel [16] have applied the concept of data flow testing to the testing of classes in object-oriented programs. The three levels of testing that they have proposed are *intramethod* testing, *intermethod* testing, and *intraclass* testing. Intermethod testing is the same as data

flow testing performed on a unit in a procedural programming language, such as C. Intermethod testing is similar to integrating program units in a procedural programming language. Finally, intraclass testing refers to calling the public methods of a class in a random, acceptable sequence.

The concept of classical data flow testing that is applied to one program unit at a time has been extended to *interprocedural* data flow testing. The idea of interprocedural data flow has been extensively studied in the literature (see ref. 19 and the bibliography of the article by Harrold and Soffa [20]).

Often programmers utilize the capability of pointers in the C and C++ languages. Data flow analysis becomes difficult when pointers are passed between procedures. Pande, Landi, and Ryder [21] have defined a term called *reaching definitions*, which is the set of all points where a value of a variable was last written. For one level of pointer indirection, they give a polynomial time algorithm for the problem. To develop the algorithm, the authors have introduced the concept of an *interprocedural control flow graph*, which is a hybrid of the control flow graph and call graph. They prove that the general problem of identifying interprocedural reaching definitions is *NP*-hard.

Lemos, Vincenzi, Maldonado, and Masiero [22] have applied the idea of data flow testing to aspect-oriented programs [23]. The concept of aspect-oriented programming was developed to address the difficulty in clearly capturing certain high-level design decisions at the code level. The properties of those design decisions are called *aspects*, and hence the name *aspect-oriented programming*. The reason some design decisions are difficult to represent at the code level is that they cross-cut the system's basic functionality [23]. Naturally, an aspect that is difficult to code is likely to be more difficult to test and verify. To this end, the work of Lemos et al. [22] gains significance. They have proposed the concept of an *aspect-oriented def-use* (AODU) graph, based on the idea of a data flow instruction graph [24], and identified new coverage criteria, such as *all-exception-independent-uses*, *all-exception-dependent-uses*, and *all-crosscutting-uses*. Zhao [25] has applied the idea of data flow testing at a coarse-grain level in aspect-oriented programs. The author extended the concept of class testing studied by Harrold and Rothermel [16].

The 1990 book *Software Testing Techniques* by Beizer [26] gives an excellent exposition of the concept of data flow testing.

REFERENCES

1. L. D. Fosdick and L. J. Osterweil. Data Flow Analysis in Software Reliability. *Computing Surveys*, September 1976, pp. 305–330.
2. J. C. Huang. Detection of Data Flow Anomaly through Program Instrumentation. *IEEE Transactions on Software Engineering*, May 1979, pp. 226–236.
3. S. Rapps and E. J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, April 1985, pp. 367–375.
4. P. G. Frankl and E. J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, October 1988, pp. 1483–1498.
5. S. C. Ntafos. On Required Element Testing. *IEEE Transactions on Software Engineering*, November 1984, pp. 795–803.

6. L. J. Osterweil and L. D. Fosdick. Dave—A Validation, Error Detection, and Documentation System for Fortran Programs. *Software—Practice and Experience*, October/December 1976, pp. 473–486.
7. J. W. Laski and B. Korel. A Data Flow Oriented Program Testing Strategy. *IEEE Transactions on Software Engineering*, May 1983, pp. 347–354.
8. P. M. Herman. A Data Flow Analysis Approach to Program Testing. *Australian Computer Journal*, November 1976, pp. 92–96.
9. H. Ural. Test Sequence Selection Based on Static Data Flow Analysis. *Computer Communications*, October 1987, pp. 234–242.
10. S. C. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, June 1988, pp. 868–874.
11. M. R. Woodward, D. Hedley, and M. A. Hennell. Experience with Path Analysis and Testing of Programs. *IEEE Transactions on Software Engineering*, May 1980, pp. 278–286.
12. L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal Evaluation of Data Flow Path Selection Criteria. *IEEE Transactions on Software Engineering*, November 1989, pp. 1318–1332.
13. L. S. Koh and M. T. Liu. Test Path Selection Based on Effective Domains. In *Proceedings of the International Conference on Network Protocols*, Boston, October 1994, IEEE Press, Piscataway, pp. 64–71.
14. B. Sarikaya, G. v. Bochmann, and E. Cerny. A Test Design Methodology for Protocol Testing. *IEEE Transactions on Software Engineering*, May 1987, pp. 518–531.
15. R. Doong and P. Frankl. The ASTOOT Approach to Testing Object-Oriented Programs. *ACM Transactions on Software Engineering and Methodology*, April 1994, pp. 101–130.
16. M. J. Harrold and G. Rothermel. Performing Data Flow Testing on Classes. In *Proceedings of ACM SIGSOFT Foundation of Software Engineering*, New Orleans, December 1994, ACM Press, New York pp. 154–163.
17. D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, K.-S. Kim, and Y.-K. Song. Developing an Object-Oriented Software Testing and Maintenance Environment. *Communications of the ACM*, October 1995, pp. 75–86.
18. A. S. Parrish, R. B. Borie, and D. W. Cordes. Automated Flow Graph-Based Testing of Object-Oriented Software Modules. *Journal of Systems and Software*, November 1993, pp. 95–109.
19. J. M. Barth. A Practical Interprocedural Data Flow Analysis Algorithm. *Communications of the ACM*, September 1978, pp. 724–736.
20. M. J. Harrold and M. L. Sofa. Efficient Computation of Interprocedural Definition-Use Chains. *ACM Transactions on Programming Languages and Systems*, March 1994, pp. 175–204.
21. H. Pande, W. Landi, and B. G. Ryder. Interprocedural Def-Use Associations in C Programs. *IEEE Transactions on Software Engineering*, May 1994, pp. 385–403.
22. O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado, and P. C. Masiero. Control and Data Flow Structural Testing Criteria for Aspect-Oriented Programs. *Journal of Systems and Software*, June 2007, pp. 862–882.
23. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS 1241, Finland, June 1997, pp. 220–242.
24. A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong, and M. E. Delamaro. Coverage Testing of Java Programs and Components. *Science of Computer Programming*, April 2005, pp. 211–230.
25. J. Zhao. Data-Flow Based Unit Testing of Aspect-Oriented Programs. In *Proceedings of the 27th Annual International Computer Software and Applications Conference*, Dallas, Texas, IEEE Press, Piscataway, 2003, pp. 188–197.
26. B. Beizer. *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, New York, 1990.

Exercises

1. Draw a data flow graph for the binsearch() function given in Figure 5.8.
2. Assuming that the input array $V[\ ]$ has at least one element in it, find an infeasible path in the data flow graph for the binsearch() function.

```
int binsearch(int X, int V[], int n){
    int low, high, mid;
    low = 0;
    high = n - 1;
    while (low <= high) {
      mid = (low + high)/2;
      if (X < V[mid])
          high = mid - 1;
      else if (X > V[mid])
           low = mid + 1;
      else
           return mid;
    }
    return -1;
}
```

Figure 5.8 Binary search routine.

```
int modifiedbinsearch(int X, int V[], int n){
  int low, high, mid;
  low = 0;
  high = n - 1;
  while (low <= high) {
    mid = (low + high)/2;
    if (X < V[mid]) {
        high = mid - 1;
        mid = mid - 1;
    }
    else if (X > V[mid])
         low = mid + 1;
    else
         return mid;
  }
  return -1;
}
```

Figure 5.9 Modified binary search routine.

3. Identify a data flow anomaly in the code given in Figure 5.9.
4. By referring to the data flow graph obtained in exercise 1, find a set of complete paths satisfying the all-defs selection criterion with respect to variable mid.
5. By referring to the data flow graph obtained in exercise 1, find a set of complete paths satisfying the all-defs selection criterion with respect to variable high.
6. Write a function in C such that the all-uses criterion produces more test cases than the all-branches criterion.
7. What is meant by the *gap* between all-branches testing and all-paths testing and how does data flow testing fill the gap?

8. Explain why the presence of data flow anomaly does not imply that execution of the program will definitely produce incorrect results.
9. Program anomaly has been defined by considering three operations, namely, define (d), reference (r), and undefine (u). The three sequences of operations identified to be program anomaly are dd, du, and ur. Explain why the rest of the two-operation sequences are not considered to be program anomaly.
10. Identify some difficulties in identifying data flow anomaly in programs.

CHAPTER 6

Domain Testing

> Even granting that the genius subjected to the test of critical inspection emerges free from all error, we should consider that everything he has discovered in a given domain is almost nothing in comparison with what is left to be discovered.
> — *Santiago Ramón y Cajal*

6.1 DOMAIN ERROR

Two fundamental elements of a computer program are *input domain* and *program paths*. The input domain of a program is the set of all input data to the program. A program path is a sequence of instructions from the start of the program to some point of *interest* in the program. For example, the end of the program is a point of interest. Another point of interest is when the program waits to receive another input from its environment so that it can continue its execution. In other words, a program path, or simply *path*, corresponds to some flow of control in the program. A path is said to be *feasible* if there exists an input data which causes the program to execute the path. Otherwise, the path is said to be *infeasible*.

Howden [1] identified two broad classes of errors, namely, *computation error* and *domain error*, by combining the concepts of *input data* and program path. The two kinds of errors have been explained in the following.

Computation Error: A computation error occurs when a specific input data causes the program to execute the correct, i.e., desired path, but the output value is wrong. Note that the output value can be wrong even if the desired path has been executed. This can happen due to a wrong function being executed in an *assignment* statement. For example, consider a desired path containing the statement result = f(a, b), where *a* and *b* are input values. A computation error may occur if the statement is replaced by a faulty one, such as result = f(b, a). Therefore, the result of executing the path can be erroneous because of a fault in the assignment statement, and this can happen in spite of executing a correct path.

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

Domain Error: A domain error occurs when a specific input data causes the program to execute a *wrong*, that is, undesired, path in the program. An incorrect path can be selected by a program if there is a fault in one or more of the *conditional* statements in the program. Let us consider a conditional statement of the form *if (p) then f1() else f2()*. If there is a fault in the formulation of the predicate p, then the wrong function call is invoked, thereby causing an incorrect path to be executed.

The above two kinds of program errors lead us to view a computer program as performing an abstract mapping function as follows. Ideally, for each input value, the program assigns a program path to execute; the same program path can be exclusively assigned (i.e., executed) for a subset of the input values. Here, the subset of the input values causing the same path to be executed is referred to an input *domain* or *subdomain*. Thus, the program is said to map a domain to a path within itself. Since there are a large number of values in the input domain of the program and there are a large number of paths in a program, we can view a program as partitioning the input space into a finite number of subdomains and assigning a distinct program path to each of the input subdomains.

We further explain the concept of program domains using Figure 6.1. The set D is the entire input set of a program P (Figure 6.1a). We call D the domain of the entire program. Set D can be an infinite set, and P may not have different computation behavior for each element of D. Instead, P may perform the same computation for all the elements in a certain subset of D. For example, as shown in Figure 6.1b, P performs *five* different computations, one for each subset D_1, $\ldots$, D_5. It may be noted that the partition of D is not visible outside P. Instead, P has a conceptual, in-built mechanism, as illustrated in Figure 6.1c, to decide the computation method needed to choose a specific branch when P is invoked with a certain input. Such an input classifier may not exist in a program in a single, clearly identifiable form. The concept can exist as an entity as a cross-cutting concept; it is cross-cutting because portions of the input classifier can be found in different program modules. We show five different computations, *computation for* D_1 through *computation for* D_5, for subsets $D_1, \ldots, D_5$, respectively (Figure 6.1c). The part of P that decides what computation to invoke for a given element of D is called an *input classifier*. We remind the reader that the structure of a program may not resemble the case we have shown inside the larger circle in Figure 6.1c. The figure simply denotes the fact that a program does different computations for different subsets of its input domain. Programs perform input classification through sequences of predicates, though an input classifier may not exist as a single module.

Therefore, a program will perform the wrong computation if there are faults in the input classification portion. With the above backdrop, we define the following two terms:

- A *domain* is a set of input values for which the program performs the same computation for every member of the set. We are interested in maximal domains such that the program performs different computations on *adjacent* domains.

Figure 6.1 Illustration of the concept of program domains.

- A program is said to have a *domain error* if the program incorrectly performs input classification. Assuming that adjacent domains perform different computations, a domain error will cause the program to produce incorrect output.

6.2 TESTING FOR DOMAIN ERRORS

The idea of *domain testing* was first studied by White and Cohen in 1978 [2, 3]. There is a fundamental difference between flow graph–based testing techniques and domain testing. By flow graph we mean control flow graph and data flow graph. The difference is explained as follows:

- Select paths from a control flow graph or a data flow graph to satisfy certain *coverage criteria*. To remind the reader, the *control flow* coverage

criteria are *statement coverage*, *branch coverage*, and *predicate coverage*. Similarly, the criteria studied to cover the *definition* and *use* aspects of variables in a program are all-defs, all-c-uses, all-p-uses, and all-uses, to name a few. The path predicates were analyzed to derive test data. While selecting paths and the corresponding test data, *no assumption is made regarding the actual type of faults that the selected test cases could potentially uncover*, that is, no specific types of faults are explicitly considered for detection.

- Domain testing takes an entirely new approach to fault detection. One defines a category of faults, called *domain errors*, and selects test data to detect those faults. If a program has domain errors, those will be revealed by the test cases.

We discuss the following concepts in detail:

- **Sources of Domains:** By means of an example program, we explain how program predicates behave as an input classifier.
- **Types of Domain Errors:** We explain how minor modifications to program predicates, which can be interpreted as programming defects, can lead to domain errors.
- **Selecting Test Data to Reveal Domain Errors:** A test selection criterion is explained to pick input values. The test data so chosen reveal the specific kinds of domain errors.

6.3 SOURCES OF DOMAINS

Domains can be identified from both specifications and programs. We explain a method to identify domains from source code using the following steps:

- Draw a control flow graph from the given source code.
- Find all possible interpretations of the predicates. In other words, express the predicates solely in terms of the input vector and, possibly, a vector of constants. The reader may note that a predicate in a program may have multiple interpretations, because control may arrive at a predicate node via different paths.
- Analyze the interpreted predicates to identify domains.

In the following, we explain the above procedure to identify domains. We show an example C function in Figure 6.2 to illustrate a procedure to identify domains.

The function accepts two inputs x and y and returns an integer. A control flow graph representation of codedomain() is shown in Figure 6.3. The two predicates in the two if() statements have been represented by nodes **3** and **6** in Figure 6.3. The predicate

$$P_1 : \qquad c > 5$$

```
int codedomain(int x, int y){
  int c, d, k
  c = x + y;
  if (c > 5) d = c - x/2;
  else       d = c + x/2;
  if (d >= c + 2) k = x + d/2;
  else            k = y + d/4;
  return(k);
}
```

Figure 6.2 A function to explain program domains.

in the first if() statement has just one interpretation, namely,

$$P_1 : \qquad x + y > 5$$

because program control reaches the if() statement via only one path from the initial node. However, predicate

Figure 6.3 Control flow graph representation of the function in Figure 6.2.

$$P_2 : \qquad d \geq c + 2$$

in the second if() statement gets two interpretations, because program control can reach the second if() statement along two paths: (i) when the first if() evaluates to true and (ii) when the first if() evaluates to false. These two interpretations are summarized in Table 6.1.

We explain a procedure to obtain domains from the interpretations of P_1 and P_2 (Figure 6.3). We show a two-dimensional grid labeled x and y in Figure 6.4. The grid size is large enough to show all the domains of the program under consideration. We consider the predicate nodes of the control flow graph one by one (Figure 6.3). Predicate P_1 divides the grid into two regions. The P_1 boundary is shown by a straight line represented by the equality $x + y = 5$. All the points above, but excluding this line, satisfy predicate P_1.

TABLE 6.1 Two Interpretations of Second if() Statement in Figure 6.2

Evaluation of P_1	Interpretation of P_2
True	$x \leq -4$
False	$x \geq 4$

Figure 6.4 Domains obtained from interpreted predicates in Figure 6.3.

Next, we consider the two interpretations of predicate P_2. For $P_1 = \text{True}$, P_2 has the following interpretation

$$P_2: \qquad x \leq -4$$

Therefore, P_2 further divides the area, or set of points, defined by $P_1 = \text{True}$ into two sets corresponding to its two truth values. The P_2 boundary, when P_1 evaluates to true, is represented by the straight line $x = -4$. The area to the left of the P_2 boundary and above the P_1 boundary corresponds to $P_1P_2 = \text{TT}$, and the area to the right of the P_2 boundary and above the P_1 boundary corresponds to $P_1P_2 = \text{TF}$.

For $P_1 = \text{False}$, P_2 has the following interpretation:

$$P_2: \qquad x > 4$$

In other words, P_2 further divides the area, or set of points, defined by $P_1 = \text{False}$ into two sets corresponding to its two truth values. The P_2 boundary, when P_1 evaluates to false, is represented by the straight line $x = 4$. The area to the right of the P_2 boundary and below the P_1 boundary corresponds to $P_1P_2 = \text{FT}$, and the area to the left of the P_2 boundary and below the P_1 boundary corresponds to $P_1P_2 = \text{FF}$ in Figure 6.4.

The reader may note that if a program contains k predicates in a sequence, the maximum number of domains obtained is 2^k. In practice, the number of domains obtained is much smaller than 2^k, because certain combinations of truth values of those k predicates may not hold simultaneously.

6.4 TYPES OF DOMAIN ERRORS

The reader may recall the following properties of a domain:

- A domain is a set of values for which the program performs identical computations.
- A domain can be represented by a set of predicates. Individual elements of the domain satisfy the predicates of the domain.

Example: The domain **TT** in Figure 6.4 is mathematically represented by the set of predicates shown in Figure 6.5.

A domain is defined, from a geometric perspective, by a set of constraints called *boundary inequalities*. Properties of a domain are discussed in terms of the properties of its *boundaries* as follows:

P1:	$x + y > 5$	$\equiv$ *True*
P2:	$x <= -4$	$\equiv$ *True*

Figure 6.5 Predicates defining the **TT** domain in Figure 6.4.

Closed Boundary: A boundary is said to be closed if the points on the boundary are included in the domain of interest.

Example: Consider the domain **TT** in Figure 6.4 and its boundary defined by the inequality

$$P_2: \qquad x \leq -4$$

The above boundary is a closed boundary of the domain **TT**.

Open Boundary: A boundary is said to be open if the points on the boundary do not belong to the domain of interest.

Example: Consider the domain **TT** in Figure 6.4 and its boundary defined by the inequality

$$P_1: \qquad x + y > 5$$

The above boundary is an open boundary of the domain **TT**. The reader may notice that it is the *equality symbol* (=) in a relational operator that determines whether or not a boundary is closed. If the relational operator in a boundary inequality has the equality symbol in it, then the boundary is a closed boundary; otherwise it is an open boundary.

Closed Domain: A domain is said to be closed if all of its boundaries are closed.

Open Domain: A domain is said to be open if some of its boundaries are open.

Extreme Point: An extreme point is a point where two or more boundaries cross.

Adjacent Domains: Two domains are said to be adjacent if they have a boundary inequality in common.

A program path will have a *domain error* if there is incorrect formulation of a path predicate. After an interpretation of an incorrect path predicate, the path predicate expression causes a boundary segment to

- be shifted from its correct position or
- have an incorrect relational operator.

A domain error can be caused by

- an incorrectly specified predicate or
- an incorrect assignment which affects a variable used in the predicate.

Now we discuss different types of domain errors:

Closure Error: A closure error occurs if a boundary is open when the intention is to have a closed boundary, or vice versa. Some examples of closure error are:

- The relational operator $\leq$ is implemented as $<$.
- The relational operator $<$ is implemented as $\leq$.

Shifted-Boundary Error: A shifted-boundary error occurs when the implemented boundary is parallel to the intended boundary. This happens when the *constant term* of the inequality defining the boundary takes up a value different from the intended value. In concrete terms, a shifted-boundary error occurs due to a change in the *magnitude* or the *sign* of the constant term of the inequality.

Example: Consider the boundary defined by the following predicate (Figure 6.4):

$$P_1 : \qquad x + y > 5$$

If the programmer's intention was to define a boundary represented by the predicate

$$P_1' : \qquad x + y > 4$$

then the boundary defined by P_1 is parallel, but not identical, to the boundary defined by P_1'.

Tilted-Boundary Error: If the constant coefficients of the variables in a predicate defining a boundary take up wrong values, then the tilted-boundary error occurs.

Example: Consider the boundary defined by the following predicate (Figure 6.4):

$$P_1 : \qquad x + y > 5$$

If the programmer's intention was to define a boundary represented by the predicate

$$P_1'' : \qquad x + 0.5y > 5$$

then the boundary defined by P_1 is tilted with respect to the boundary defined by P_1''.

The reader may recall that for all the data points in a domain the program performs identical computations. It is not difficult to notice that input data points fall in the wrong domain if there is a closure defect, a shifted boundary, or a tilted boundary. Assuming that domains are maximal in size in the sense that adjacent domains perform different computations, a program will produce a wrong outcome because of wrong computations performed on those input data points which fall in the wrong domains.

6.5 ON AND OFF POINTS

In domain testing a programmer targets domain errors where test cases are designed with the objective of revealing the domain errors as discussed in Section 6.4. Therefore, it is essential that we consider an important characteristic of domain errors, stated as follows: *Data points on or near a boundary are most sensitive to domain errors*. In this observation, by *sensitive* we mean data points falling in the wrong domains. Therefore, the objective is to identify the data points that are most sensitive to domain errors so that errors can be detected by executing the program with those input values. In the following, we define two kinds of data points *near* domain boundaries, namely, **ON** point and **OFF** point:

ON Point: Given a boundary, an **ON** point is a point *on* the boundary or "very close" to the boundary.

This definition suggests that we can choose an **ON** point in two ways. Therefore, one must know when to choose an **ON** point in which way:

- If a point can be chosen to lie exactly on the boundary, then choose such a point as an **ON** point. If the boundary inequality leads to an *exact* solution, choose such an exact solution as an **ON** point.
- If a boundary inequality leads to an *approximate* solution, choose a point very close to the boundary.

Example: Consider the following boundary inequality. This inequality is not related to our running example of Figure 6.4.

$$P_{\text{ON1}}: \qquad x + 7y \geq 6$$

For $x = -1$, the predicate P_{ON1} leads to an exact solution of $y = 1$. Therefore, the point $(-1, 1)$ lies on the boundary.

However, if we choose $x = 0$, the predicate P_{ON1} leads to an approximate solution of y in the form of $y = 0.8571428\ldots$. Since y does not have an exact solution, we either truncate it to 0.857 or round it off to 0.858. We notice that the point (0, 0.857) does not satisfy the predicate P_{ON1}, whereas the point (0, 0.858) does. Thus, (0, 0.858) is an **ON** point which lies very close to the P_{ON1} boundary.

Example: Consider a domain with the following *open* boundary:

$$P_{\text{ON2}}: \qquad x + 7y < 6$$

Points lying exactly on the boundary defined by the predicate

$$P'_{\text{ON2}}: \qquad x + 7y = 6$$

are not a part of the domain under consideration. The point $(-1, 1)$ lies exactly on the boundary P'_{ON2} and is an ON point. Note that the point $(-1, 1)$ is *not* a part of the domain under consideration. Similarly, the point (0, 0.858), which is almost on the boundary, that is, very close to the boundary, is an ON point and it lies outside the domain of interest.

OFF Point: An **OFF** point of a boundary lies *away* from the boundary. However, while choosing an **OFF** point, we must consider whether a boundary is *open* or *closed* with respect to a domain:

- If the domain is *open* with respect to the boundary, then an **OFF** point of that boundary is an *interior* point inside the domain within an ϵ-distance from the boundary.
- If the domain is *closed* with respect to the boundary, then an **OFF** point of that boundary is an *exterior* point outside the boundary within an ϵ-distance. The symbol ϵ denotes an arbitrarily small value.

Example: Consider a domain D_1 with a *closed* boundary as follows:

$$P_{\text{OFF1}}: \qquad x + 7y \geq 6$$

Since the boundary is closed, an OFF point lies outside the domain; this means that the boundary inequality is *not* satisfied. Note that the point $(-1, 1)$ lies exactly on the boundary and it belongs to the domain. Therefore, $(-1, 1)$ is *not* an OFF point. However, the point $(-1, 0.99)$ lies outside the domain, and it is not a part of the domain under consideration. This is easily verified by substituting $x = -1$ and $y = 0.99$ in the above P_{OFF1} inequality which produces a value of 5.93. Therefore, $(-1, 0.99)$ is an OFF point.

Example: Consider a domain D_2 which is adjacent to domain D_1 in the above example with an *open* boundary as follows:

$$P_{\text{OFF2}}: \qquad x + 7y < 6$$

It may be noted that we have obtained **POFF2** from **POFF1** by simply reversing the $\geq$ inequality. Since the P_{OFF2} boundary is open, an OFF point lies inside the domain. It can be easily verified that the point $(-1, 0.99)$ lies inside D_2, and hence it is an OFF point for domain D_2 with respect to boundary P_{OFF2}.

Summary The above ideas of ON and OFF points lead to the following conclusions:

- While testing a *closed* boundary, the ON points are in the domain under test, whereas the OFF points are in an adjacent domain.
- While testing an *open* boundary, the ON points are in an adjacent domain, whereas the OFF points are in the domain being tested.

The above ideas have been further explained in Figure 6.6, which shows two domains D_1 and D_2 defined by predicates $x < 4$ and $x \geq 4$, respectively. Therefore, the actual boundary is defined by the following predicate:

$$P_{\text{ON,OFF}}: \qquad x = 4$$

In the figure, we show two ON points A and B, where A lies exactly on the boundary and B lies "very close" to the boundary. Therefore, we have $A = 4$ and

Figure 6.6 ON and OFF points.

$B = 4.00001$, for example. We show an OFF point C lying in D_1 away from the boundary. Point $C = 3.95$ lies inside domain D_1 and outside domain D_2.

6.6 TEST SELECTION CRITERION

In this section, we explain a criterion for test selection and show that test data so selected reveal the domain errors identified in Section 6.4. Before we explain the selection criterion, we state the assumptions made in domain testing as follows:

- A program performs different computations in adjacent domains. If this assumption does not hold, then data points falling in the wrong domains may not have any influence on program outcome, and therefore failures will not be observed.
- Boundary predicates are *linear functions* of input variables. This is not a strong assumption given that most of the predicates in real-life programs are linear. This is because programmers can easily visualize linear predicates and use them.

We present the following criterion for domain testing and show that test data selected using this criterion reveal domain errors:

Test Selection Criterion: For each domain and for each boundary, select three points A, C, and B in an ON–OFF–ON sequence.

This criterion generates test data that reveal domain errors. Specifically, the following kinds of errors are considered:

1. Closed inequality boundary
   a. Boundary shift resulting in a reduced domain
   b. Boundary shift resulting in an enlarged domain
   c. Boundary tilt
   d. Closure error
2. Open inequality boundary
   a. Boundary shift resulting in a reduced domain
   b. Boundary shift resulting in an enlarged domain
   c. Boundary tilt
   d. Closure error
3. Equality boundary

In our analysis below, we consider two adjacent domains D_1 and D_2. We assume that the program computation associated with D_1 and D_2 are f_1 and f_2, respectively, and $f_1 \neq f_2$.

1a *(Closed Inequality) Boundary Shift Resulting in Reduced Domain:* The boundary between the two domains D_1 and D_2 has shifted by a certain amount (see Figure 6.7). The figure shows the actual boundary between the two domains and an arbitrary position of the expected boundary. One must remember that we do not know the exact position of the expected boundary. The expected boundary has been shown only to explain that the actual boundary has moved away from the expected boundary for conceptual understanding of boundary shift. The boundary between the two domains is closed with respect to domain D_1. Therefore, the two

Figure 6.7 Boundary shift resulting in reduced domain (closed inequality).

ON points A and B belong to domain D_1, and the OFF point C belongs to domain D_2. Hence the actual output from the program corresponding to test data A, B, and C are $f_1(A)$, $f_1(B)$, and $f_2(C)$, respectively. It is obvious from Figure 6.7 that in the absence of any boundary shift all the test points belong to domain D_1. Therefore, the expected output corresponding to test data A, B, and C are $f_1(A)$, $f_1(B)$, and $f_1(C)$, respectively. These outputs are listed in Table 6.2. We observe, by comparing the second and the third columns of Table 6.2, that the actual output and the expected output are not identical for data point C. Hence, data point C reveals the shifted-boundary fault.

It is important to understand the following at this point:

- We do not need to know the exact position of the expected boundary. This is because what we actually need are the expected program outcomes in response to the three data points A, B, and C, which can be computed from the specification of a program without explicitly finding out the expected boundary.
- All three data points A, B, and C need not reveal the same fault. Our purpose is to show that test data selected according to the stated criterion reveal all domain errors. The purpose is satisfied if at least one data point reveals the fault. Different elements of the set $\{A, B, C\}$ reveal different kinds of domain errors.
- If point C is away from the boundary by a magnitude of ϵ, then a boundary shift of magnitude less than ϵ cannot be detected. This is because the expected output $f_2(C)$ is identical to the actual output $f_2(C)$.

1b *(Closed Inequality) Boundary Shift Resulting in Enlarged Domain:* To detect this fault, we use Figure 6.8, where the boundary between the two domains D_1 and D_2 has shifted from its expected position such that the size of the domain D_1 under consideration has enlarged. Once again, we do not know the exact position of the expected boundary. The boundary between the two domains is closed with respect to domain D_1. Therefore, the two ON points A and B belong to domain D_1, and the OFF point C belongs to domain D_2. Hence the actual outputs from the program corresponding to test data A, B, and C are $f_1(A)$, $f_1(B)$, and $f_2(C)$, respectively. From Figure 6.8 it is clear that, in the absence of any boundary shift, all the test points belong to domain D_2. Therefore, the expected outputs corresponding to test data A, B, and C are $f_2(A)$, $f_2(B)$, and $f_2(C)$, respectively. We observe from Table 6.3 that the actual output and the expected output are not identical for data points A and B. Hence, data points A and B

TABLE 6.2 Detection of Boundary Shift Resulting in Reduced Domain (Closed Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_1(A)$	$f_1(A)$	No
B	$f_1(B)$	$f_1(B)$	No
C	$f_2(C)$	$f_1(C)$	Yes

Figure 6.8 Boundary shift resulting in enlarged domain (closed inequality).

TABLE 6.3 Detection of Boundary Shift Resulting in Enlarged Domain (Closed Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_1(A)$	$f_2(A)$	Yes
B	$f_1(B)$	$f_2(B)$	Yes
C	$f_2(C)$	$f_2(C)$	No

reveal the shifted-boundary fault. If the magnitude of the shift is less than ϵ—the magnitude by which the OFF point is away from the boundary—the boundary shift cannot be detected by these test data.

1c *(Closed Inequality) Boundary Tilt:* In Figure 6.9 the boundary between the two domains D_1 and D_2 has tilted by an appreciable amount. The boundary between the two domains is closed with respect to domain D_1. Therefore, the two ON points A and B belong to domain D_1, and the OFF point C belongs to domain D_2. Hence the actual outputs from the program corresponding to test data A, B, and C are $f_1(A)$, $f_1(B)$, and $f_2(C)$, respectively. It is clear from Figure 6.9 that in the absence of any boundary tilt test point A falls in domain D_1 and test points B and C fall in domain D_2. Therefore, the expected outputs corresponding to test

Figure 6.9 Tilted boundary (closed inequality).

data A, B, and C are $f_1(A)$, $f_2(B)$, and $f_2(C)$, respectively. By comparing the second and the third columns of Table 6.4 we observe that the actual output and the expected output are not identical for test point B. Hence, test point B reveals the tilted-boundary fault.

1d *(Closed Inequality) Closure Error:* The expected boundary between the two domains in Figure 6.10 is closed with respect to domain D_1. However, in an actual implementation, it is open with respect to D_1, resulting in a closure error. The boundary between the two domains belongs to domain D_2. The two ON points A and B belong to domain D_2, and the OFF point C belongs to domain D_1. Hence the actual outputs from the program corresponding to test data A, B, and C are $f_2(A)$, $f_2(B)$, and $f_1(C)$, respectively. In the absence of any closure error all three test points A, B and C fall in domain D_1. These outputs are listed in Table 6.5. By comparing the second and the third columns of Table 6.5 we observe that the actual output and the expected output are not identical for data points A and B. Therefore, data points A and B reveal the closure boundary fault.

2a *(Open Inequality) Boundary Shift Resulting in Reduced Domain:* To explain the detection of this type of error, we use Figure 6.11, where the boundary between the two domains D_1 and D_2 has shifted by a certain amount. The boundary between the two domains is open with respect to domain D_1. Therefore, the two ON points A and B belong to domain D_2, and the OFF point C belongs to domain D_1. Hence the actual outputs from the program corresponding to test data A, B, and C are $f_2(A)$, $f_2(B)$, and $f_1(C)$, respectively. It is obvious from Figure 6.11 that, in the absence of any boundary shift, all the test points belong to domain D_1. Therefore, the expected outputs corresponding to test data A, B,

TABLE 6.4 Detection of Boundary Tilt (Closed Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_1(A)$	$f_1(A)$	No
B	$f_1(B)$	$f_2(B)$	Yes
C	$f_2(C)$	$f_2(C)$	No

Figure 6.10 Closure error (closed inequality).

TABLE 6.5 Detection of Closure Error (Closed Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_2(A)$	$f_1(A)$	Yes
B	$f_2(B)$	$f_1(B)$	Yes
C	$f_1(C)$	$f_1(C)$	No

Figure 6.11 Boundary shift resulting in reduced domain (open inequality).

and C are $f_1(A)$, $f_1(B)$, and $f_1(C)$, respectively. By comparing the second and third columns of Table 6.6 we observe that the actual output and the expected output are not identical for the data point C. Therefore, data point C reveals the shifted-boundary fault.

2b *(Open Inequality) Boundary Shift Resulting in Enlarged Domain:* We use Figure 6.12 to explain the detection of this kind of errors. The boundary between the two domains D_1 and D_2 has shifted to enlarge the size of the domain D_1 under consideration. The boundary between the two domains is open with respect to domain D_1. Therefore, the two ON points A and B belong to domain D_2, and the OFF point C belongs to domain D_1. Hence the actual outputs from the program corresponding to test data A, B, and C are $f_2(A)$, $f_2(B)$, and $f_1(C)$, respectively. It follows from Figure 6.12 that, in the absence of any boundary shift, all the test points belong to domain D_2. Therefore, the expected outputs corresponding to test data A, B, and C are $f_2(A)$, $f_2(B)$, and $f_2(C)$, respectively. These outputs

TABLE 6.6 Detection of Boundary Shift Resulting in Reduced Domain (Open Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_2(A)$	$f_1(A)$	Yes
B	$f_2(B)$	$f_1(B)$	Yes
C	$f_1(C)$	$f_1(C)$	No

Figure 6.12 Boundary shift resulting in enlarged domain (open inequality).

are listed in Table 6.7. From Table 6.7 we observe that data point C reveals the shifted-boundary fault.

2c *(Open Inequality) Boundary Tilt:* We explain the boundary tilt fault by referring to Figure 6.13, where the boundary between the two domains D_1 and D_2 has tilted. Once again, we do not know the exact position of the expected boundary. The boundary between the two domains is open with respect to domain D_1. Therefore, the two ON points A and B belong to domain D_2, and the OFF point C belongs to domain D_1. Hence the actual outputs from the program corresponding to test data A, B, and C are $f_2(A)$, $f_2(B)$, and $f_1(C)$, respectively. Figure 6.13 shows that in the absence of any boundary tilt test points A and C fall in domain D_1, and test point B falls in domain D_2. Therefore, the expected outputs corresponding to test data A, B, and C are $f_1(A)$, $f_2(B)$, and $f_1(C)$, respectively. We compare the second and third columns of Table 6.8 to observe that the actual output and the expected output are not identical for the test point A. Hence, the test point A reveals the tilted-boundary fault.

2d *(Open Inequality) Closure Error:* Detection of this kind of fault is explained by using the two domains of Figure 6.14, where the expected boundary between the two domains is open with respect to domain D_1. However, in an actual implementation it is closed with respect to D_1, resulting in a closure error. The two ON points A and B belong to domain D_1, and the OFF point C belongs to domain D_2. Hence the actual outputs from the program corresponding to test data A, B, and C are $f_1(A)$, $f_1(B)$, and $f_2(C)$, respectively. Figure 6.14 shows that, in the absence of any closure error, all three test points A, B and C fall in domain D_2. Table 6.9 shows the actual outputs and the expected outputs. By

TABLE 6.7 Detection of Boundary Shift Resulting in Enlarged Domain (Open Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_2(A)$	$f_2(A)$	No
B	$f_2(B)$	$f_2(B)$	No
C	$f_1(C)$	$f_2(C)$	Yes

Figure 6.13 Tilted boundary (open inequality).

TABLE 6.8 Detection of Boundary Tilt (Open Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_2(A)$	$f_1(A)$	Yes
B	$f_2(B)$	$f_2(B)$	No
C	$f_1(C)$	$f_1(C)$	No

comparing the second and the third columns of Table 6.9 we observe that the actual output and the expected output are not identical for data points A and B. Therefore, data points A and B reveal the closure boundary fault.

3. *Equality Boundary:* Sometimes a domain may consist of an equality boundary sandwiched between two open domains, as shown in Figure 6.15, where D_1 and D_2 are two domains open with respect to their common equality boundary. In this case, to test the common boundary, we choose two ON points A and B on the boundary and two OFF points **C** and **D**—one in each open domain.

Figure 6.14 Closure error (open inequality).

TABLE 6.9 Detection of Closure Error (Open Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_1(A)$	$f_2(A)$	Yes
B	$f_1(B)$	$f_2(B)$	Yes
C	$f_2(C)$	$f_2(C)$	No

6.7 SUMMARY

Two kinds of program errors, namely *computation error* and *domain errors*, were identified. A computation error occurs when an input value causes the program to execute the correct path, but the program output is incorrect due to a fault in an assignment statement. A domain error occurs when an input value causes the program to execute the wrong path. A program executes a wrong path because of faults in conditional statements. A program can be viewed as an abstract classifier that partitions the input domain into a finite number of subdomains such that a separate program path executes for each input subdomain. Thus, a program is seen to be mapping the input subdomains to its execution paths. Program subdomains can be identified by considering individual paths in the program and evaluating path predicates. Each subdomain, also called domain, is defined by a set of boundaries. Often input data points would fall in a wrong domain if there are faults in defining domain boundaries, thereby executing the wrong paths. Input domains were characterized by means of a few properties, such as *closed boundary*, *open boundary*, *closed domain*, *open domain*, *extreme point*, and *adjacent domain*. Next, three kinds of boundary errors, namely, *closure error*, *shifted-boundary error*, and *tilted-boundary error*, were identified. Given a domain and its boundaries, the concept of **ON** and **OFF** points were explained. Finally, a test selection criterion was defined to choose test points to reveal domain errors. Specifically, the selection criterion is as follows: For each domain and for each boundary, select three points A, C, and B in an **ON–OFF–ON** sequence.

Figure 6.15 Equality border.

LITERATURE REVIEW

Since White and Cohen proposed the concept of domain testing in 1978, it has been analyzed and extended in several ways. In 1982, Clarke, Hassell, and Richardson [4] showed that some domain errors go undetected by the White and Cohen strategy. Next, they proposed a strategy, namely the $V \times V$ strategy, to improve domain testing. If a domain border under consideration contains V vertices, then the $V \times V$ strategy selects V **ON** points—one **ON** point as close as possible to each vertex—and V **OFF** points. The V **OFF** points are chosen at a uniform distance from the border. Zeil, Afifi, and White [5] introduced a domain testing strategy to detect linear errors in *nonlinear* predicate functions. A few other variants of domain testing have been proposed by White and Perera [6] and Onoma, Yamaura, and Kobayashi [7].

Zeil [8] considers domain errors that may be caused by faults in arithmetic and simple relational expressions. These expressions are restricted to floating-point or integer computations. Fault detection techniques, called perturbation techniques, are presented to reveal domain errors.

Koh and Liu [9] have presented an approach for generating paths that test both the control flow and the data flow in implementations of communication protocols. The protocols are assumed to be modeled as extended finite-state machines. The path selection approach consists of two steps: (i) select a set of paths to cover a data flow selection criterion and (ii) selectively augment the state transitions in the chosen set of paths with state check sequences so that control flow can be ensured and data flow coverage can be preserved. Augmentation of state transitions is performed by using the concept of *effective domains*.

Jeng and Weyuker [10] have proposed a simplified domain testing strategy that is applicable to arbitrary types of predicates and detects both linear and nonlinear errors for both continuous and discrete variable spaces. Moreover, the strategy requires a *constant* number of test points. That is, the number of test points is independent of the dimension or the type of border or the number of vertices on the border under consideration. Their simplified technique requires us to generate one **ON** point and one **OFF** point for an inequality (i.e., $\leq$, $<$, $\geq$, or $>$) border. For an equality (i.e., $=$) or nonequality (i.e., $\neq$) border, one **ON** and two **OFF** test points are required. The test generation technique requires (i) an **ON** point to lie on the border, (ii) an **OFF** point to lie outside the border, and (iii) an **ON–OFF** pair to be as close to each other as possible. Hajnal and Forgacs [11] have given an algorithm to generate **ON–OFF** points that can be used by the simplified domain testing strategy.

In contrast, the test selection strategy of White and Cohen [3] requires the selection of N **ON** points in all cases, where N is the dimension of the input space, and the Clarke, Hassell, and Richardson [4] strategy requires the selection of V **ON** points, where V is the number of vertices on the border under consideration.

Zhao, Lyu, and Min [12] have studied an approach to generate **ON–OFF** test points for character string predicate borders associated with program paths. They use the idea of program slicing [13] to compute the current values of variables in the

predicates. The same authors have shown in reference [14] that partition testing strategies are relatively ineffective in detecting faults related to small shifts in input domain boundary, and presented a different testing approach based on input domain analysis of specifications and programs.

An elaborate treatment of domain testing can be found in the book by Beizer [15]. Beizer explains how the idea of domains can be used in testing interfaces between program units.

REFERENCES

1. W. E. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, September 1976, pp. 208–215.
2. L. J. White and E. I. Cohen. A Domain Strategy for Computer Program Testing. Paper presented at the IEEE Workshop on Software Testing and Documentation, Fort Lauderdale, FL, 1978, pp. 335–346.
3. L. J. White and E. I. Cohen. A Domain Strategy for Computer Program Testing. *IEEE Transactions on Software Engineering*, May 1980, pp. 247–257.
4. L. Clarke, H. Hassell, and D. Richardson. A Close Look at Domain Testing. *IEEE Transactions on Software Engineering*, July 1982, pp. 380–392.
5. S. J. Zeil, F. H. Afifi, and L. J. White. Detection Linear Errors via Domain Testing. *ACM Transactions on Software Engineering and Methodology*, October 1992, pp. 422–451.
6. L. J. White and I. A. Perera. An Alternative Measure for Error Analysis of the DomainTesting Strategy. In *Proceedings of the ACM SIGSOFT/IEEE Workshop on Software Testing*, Banff, Canada, IEEE Press, New York, 1986, pp. 122–131.
7. A. K. Onoma, T. Yamaura, and Y. Kobayashi. Practical Approaches to Domain Testing: Improvements and Generalization. In *Proceedings of COMPSAC*, Tokyo, IEEE Computer Society Press, Piscataway, NJ, 1987, pp. 291–297.
8. S. J. Zeil. Perturbation Technique for Detecting Domain Errors. *IEEE Transactions on Software Engineering*, June 1989, pp. 737–746.
9. L. S. Koh and M. T. Liu. Test Path Selection Based on Effective Domains. In *Proceedings of the International Conference on Network Protocols*, Boston, IEEE Computer Society Press, Piscataway, NJ, October 1994, pp. 64–71.
10. B. Jeng and E. J. Weyuker. A Simplified Domain Testing Strategy. *ACM Transactions on Software Engineering and Methodology*, July 1994, pp. 254–270.
11. A. Hajnal and I. Forgacs. An Applicable Test Data Generation Algorithm for Domain Errors. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, Clearwater Beach, FL, ACM Press, New York, March 1998, pp. 63–72.
12. R. Zhao, M. R. Lyu, and Y. Min. Domain Testing Based on Character String Predicate. Paper presented at the Asian Test Symposium, Xian, China, IEEE Computer Society Press, Piscataway, NJ, 2003, pp. 96–101.
13. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, September 1995, pp. 121–189.
14. R. Zhao, M. R. Lyu, and Y. Min. A New Software Testing Approach Based on Domain Analysis of Specifications and Programs. In *Proceedings of 14th Symposium on Software Reliability Engineering*, Colorado, IEEE Computer Society Press, New York, 2003, pp. 60–70.
15. B. Beizer. *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, New York, 1990.

Exercises

1. Explain what are *computation error* and *domain error*.
2. Give an example of code showing a domain error.

Figure 6.16 Domains D_1, D_2, and D_3.

3. Explain the difference between control flow–based testing and domain error–based testing.
4. Recall that the domain testing strategy requires us to select test points on and/or very close to domain boundaries. Why do we not select test points far from the boundaries?
5. Consider the three domains D_1, D_2, and D_3 shown in Figure 6.16. Domain D_3 consists of all those points lying on the indicated straight line. Assuming that the maximum X and Y span of all the three domains are $[-5, 5]$ and $[-5, 5]$, respectively, give concrete values of test points for domain D_3.
6. State four kinds of domain errors and explain how they occur.
7. Explain the following terms: *closed boundary*, *open boundary*, *closed domain*, *open domain*, *extreme point*, *adjacent domain*.
8. Explain the idea of **ON** points and **OFF** points.
9. Clearly explain the test selection criterion in domain-based testing and show the closed inequality error (boundary shift resulting in a reduced domain) is detected by the test points chosen by the selection criterion.
10. Identify some difficulties in applying the concept of domain testing to actual program testing.

CHAPTER 7

System Integration Testing

I criticize by creation, not by finding fault.
— *Marcus Tullius Cicero*

7.1 CONCEPT OF INTEGRATION TESTING

A software module, or component, is a self-contained element of a system. Modules have well-defined interfaces with other modules. A module can be a subroutine, function, procedure, class, or collection of those basic elements put together to deliver a higher level service. A system is a collection of modules interconnected in a certain way to accomplish a tangible objective. A subsystem is an interim system that is not fully integrated with all the modules. It is also known as a subassembly.

In moderate to large projects, from tens to hundreds of programmers implement their share of the code in the form of modules. Modules are individually tested, which is commonly known as *unit testing*, by their respective programmers using white-box testing techniques. At the unit testing level, the system exists in pieces under the control of the programmers. The next major task is to put the modules, that is, pieces, together to construct the complete system. Constructing a working system from the pieces is not a straightforward task, because of numerous interface errors. Even constructing a reasonably stable system from the components involves much testing. The path from tested components to constructing a deliverable system contains two major testing phases, namely, integration testing and system testing. The primary objective of integration testing is to assemble a reasonably stable system in a laboratory environment such that the integrated system can withstand the rigor of a full-blown system testing in the actual environment of the system. The importance of integration testing stems from three reasons as outlined below.

- Different modules are generally created by groups of different developers. The developers may be working at different sites. In spite of our best effort in system design and documentation, misinterpretation, mistakes, and

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

oversights do occur in reality. Interface errors between modules created by different programmers and even by the same programmers are rampant. We will discuss the sources of interface errors in Section 7.2.

- Unit testing of individual modules is carried out in a controlled environment by using test drivers and stubs. Stubs are dummy modules which merely return predefined values. If a module under unit test invokes several other modules, the effectiveness of unit testing is constrained by the programmer's ability to effectively test all the paths. Therefore, with the inherent limitations of unit testing, it is difficult to predict the behavior of a module in its actual environment after the unit testing is performed.
- Some modules are more error prone than other modules, because of their inherent complexity. It is essential to identify the ones causing most failures.

The objective of system integration is to build a "working" version of the system by (i) putting the modules together in an incremental manner and (ii) ensuring that the additional modules work as expected without disturbing the functionalities of the modules already put together. In other words, system integration testing is a systematic technique for assembling a software system while conducting tests to uncover errors associated with interfacing. We ensure that unit-tested modules operate correctly when they are combined together as dictated by the design. Integration testing usually proceeds from small subassemblies containing a few modules to larger ones containing more and more modules. Large, complex software products can go through several iterations of build-and-test cycles before they are fully integrated.

Integration testing is said to be complete when the system is fully integrated together, all the test cases have been executed, all the severe and moderate defects found have been fixed, and the system is retested.

7.2 DIFFERENT TYPES OF INTERFACES AND INTERFACE ERRORS

Modularization is an important principle in software design, and modules are interfaced with other modules to realize the system's functional requirements. An interface between two modules allows one module to access the service provided by the other. It implements a mechanism for passing control and data between modules. Three common paradigms for interfacing modules are as follows:

- **Procedure Call Interface:** A procedure in one module calls a procedure in another module. The caller passes on control to the called module. The caller can pass data to the called procedure, and the called procedure can pass data to the caller while returning control back to the caller.
- **Shared Memory Interface:** A block of memory is shared between two modules. The memory block may be allocated by one of the two modules

or a third module. Data are written into the memory block by one module and are read from the block by the other.

- **Message Passing Interface:** One module prepares a message by initializing the fields of a data structure and sending the message to another module. This form of module interaction is common in client–server-based systems and web-based systems.

Programmers test modules to their satisfaction. The question is: If all the unit-tested modules work individually, why can these modules not work when put together? The problem arises when we "put them together" because of rampant interface errors. Interface errors are those that are associated with structures existing outside the local environment of a module but which the module uses [1]. Perry and Evangelist [2] reported in 1987 that interface errors accounted for up to a quarter of all errors in the systems they examined. They found that of all errors that required a fix within one module, more than half were caused by interface errors. Perry and Evangelist have categorized interface errors as follows:

1. *Construction*: Some programming languages, such as C, generally separate the interface specification from the implementation code. In a C program, programmers can write a statement #include header.h, where header.h contains an interface specification. Since the interface specification lies somewhere away from the actual code, programmers overlook the interface specification while writing code. Therefore, inappropriate use of #include statements cause construction errors.
2. *Inadequate Functionality*: These are errors caused by implicit assumptions in one part of a system that another part of the system would perform a function. However, in reality, the "other part" does not provide the expected functionality—intentionally or unintentionally by the programmer who coded the other part.
3. *Location of Functionality*: Disagreement on or misunderstanding about the location of a functional capability within the software leads to this sort of error. The problem arises due to the design methodology, since these disputes should not occur at the code level. It is also possible that inexperienced personnel contribute to the problem.
4. *Changes in Functionality*: Changing one module without correctly adjusting for that change in other related modules affects the functionality of the program.
5. *Added Functionality*: A completely new functional module, or capability, was added as a system modification. Any added functionality after the module is checked in to the version control system without a CR is considered to be an error.
6. *Misuse of Interface*: One module makes an error in using the interface of a called module. This is likely to occur in a procedure–call interface. Interface misuse can take the form of wrong parameter type, wrong parameter order, or wrong number of parameters passed.

7. *Misunderstanding of Interface*: A calling module may misunderstand the interface specification of a called module. The called module may assume that some parameters passed to it satisfy a certain condition, whereas the caller does not ensure that the condition holds. For example, assume that a called module is expected to return the index of an element in an array of integers. The called module may choose to implement binary search with an assumption that the calling module gives it a sorted array. If the caller fails to sort the array before invoking the second module, we will have an instance of interface misunderstanding.
8. *Data Structure Alteration*: These are similar in nature to the functionality problems discussed above, but they are likely to occur at the detailed design level. The problem arises when the size of a data structure is inadequate or it fails to contain a sufficient number of information fields. The problem has its genesis in the failure of the high-level design to fully specify the capability requirements of the data structure. Let us consider an example in which a module reads the data and keeps it in a record structure. Each record holds the person name followed by their employee number and salary. Now, if the data structure is defined for 1000 records, then as the number of record grows beyond 1000, the program is bound to fail. In addition, if management decides to award bonuses to a few outstanding employees, there may not be any storage space allocated for additional information.
9. *Inadequate Error Processing*: A called module may return an error code to the calling module. However, the calling module may fail to handle the error properly.
10. *Additions to Error Processing*: These errors are caused by changes to other modules which dictated changes in a module error handling. In this case either necessary functionality is missing from the current error processing that would help trace errors or current techniques of error processing require modification.
11. *Inadequate Postprocessing*: These errors are caused by a general failure to release resources no longer required, for example, failure to deallocate memory.
12. *Inadequate Interface Support*: The actual functionality supplied was inadequate to support the specified capabilities of the interface. For example, a module passes a temperature value in Celsius to a module which interprets the value in Fahrenheit.
13. *Initialization/Value Errors*: A failure to initialize, or assign, the appropriate value to a variable data structure leads to this kind of error. Problems of this kind are usually caused by simple oversight. For example, the value of a pointer can change; it might point to the first character in a string, then to the second character, after that to the third character, and so on. If the programmer forgets to reinitialize the pointer before using that function once again, the pointer may eventually point to code.

14. *Violation of Data Constraints*: A specified relationship among data items was not supported by the implementation. This can happen due to incomplete detailed design specifications.
15. *Timing/Performance Problems*: These errors were caused by inadequate synchronization among communicating processes. A race condition is an example of these kinds of error. In the classical race, there are two possible events event a and event b happening in communicating processes process A and process B, respectively. There is logical ground for expecting event a to precede event b. However, under an abnormal condition event b may occur before event a. The program will fail if the software developer did not anticipate the possibility of event b preceding event a and did not write any code to deal with the situation.
16. *Coordination of Changes*: These errors are caused by a failure to communicate changes to one software module to those responsible for other interrelated modules.
17. *Hardware/Software Interfaces*: These errors arise from inadequate software handling of hardware devices. For example, a program can send data at a high rate until the input buffer of the connected device is full. Then the program has to pause until the device frees up its input buffer. The program may not recognize the signal from the device that it is no longer ready to receive more data. Loss of data will occur due to a lack of synchronization between the program and the device.

Interface errors cannot be detected by performing unit testing on modules since unit testing causes computation to happen within a module, whereas interactions are required to happen between modules for interface errors to be detected. It is difficult to observe interface errors by performing system-level testing, because these errors tend to be buried in system internals. The major advantages of conducting system integration testing are as follows:

- Defects are detected early.
- It is easier to fix defects detected earlier.
- We get earlier feedback on the health and acceptability of the individual modules and on the overall system.
- Scheduling of defect fixes is flexible, and it can overlap with development.

System integration testing is performed by the system integration group, also known as a build engineering group. The integration test engineers need to know the details of the software modules. This means that the team of engineers who built the modules needs to be involved in system integration. The integration testers should be familiar with the interface mechanisms. The system architects should be involved in the integration testing of complex software systems because of the fact that they have the bigger picture of the system.

7.3 GRANULARITY OF SYSTEM INTEGRATION TESTING

System integration testing is performed at different levels of granularity. Integration testing includes both white- and black-box testing approaches. *Black-box* testing ignores the internal mechanisms of a system and focuses solely on the outputs generated in response to selected inputs and execution conditions. The code is considered to be a big black box by the tester who cannot examine the internal details of the system. The tester knows the input to the black box and observes the expected outcome of the execution. *White-box* testing uses information about the structure of the system to test its correctness. It takes into account the internal mechanisms of the system and the modules. In the following, we explain the ideas of *intrasystem* testing, *intersystem* testing, and *pairwise* testing.

1. *Intrasystem Testing:* This form of testing constitutes low-level integration testing with the objective of combining the modules together to build a cohesive system. The process of combining modules can progress in an incremental manner akin to constructing and testing successive builds, explained in Section 7.4.1. For example, in a client–server-based system both the client and the server are distinct entities running at different locations. Before the interactions of clients with a server are tested, it is essential to individually construct the client and the server systems from their respective sets of modules in an incremental fashion. The low-level design document, which details the specification of the modules within the architecture, is the source of test cases.

2. *Intersystem Testing:* Intersystem testing is a high-level testing phase which requires interfacing independently tested systems. In this phase, all the systems are connected together, and testing is conducted from end to end. The term *end to end* is used in communication protocol systems, and end-to-end testing means initiating a test between two access terminals interconnected by a network. The purpose in this case is to ensure that the interaction between the systems work together, but not to conduct a comprehensive test. Only one feature is tested at a time and on a limited basis. Later, at the time of system testing, a comprehensive test is conducted based on the requirements, and this includes functional, interoperability, stress, performance, and so on. Integrating a client–server system, after integrating the client module and the server module separately, is an example of intersystem testing. Integrating a call control system and a billing system in a telephone network is another example of intersystem testing. The test cases are derived from the high-level design document, which details the overall system architecture.

3. *Pairwise Testing:* There can be many intermediate levels of system integration testing between the above two extreme levels, namely intrasystem testing and intersystem testing. Pairwise testing is a kind of intermediate level of integration testing. In pairwise integration, only two interconnected systems in an overall system are tested at a time. The purpose of pairwise testing is to ensure that two systems under consideration can function together, assuming that the other systems

within the overall environment behave as expected. The whole network infrastructure needs to be in place to support the test of interactions of the two systems, but the rest of the systems are not subject to tests. The network test infrastructure must be simple and stable during pairwise testing. While pairwise testing may sound simple, several issues can complicate the testing process. The biggest issue is unintended side effects. For example, in testing communication between a network element (radio node) and the element management systems, if another device (radio node controller) within the 1xEV-DO wireless data network, discussed in Chapter 8, fails during the test, it may trigger a high volume of traps to the element management systems. Untangling this high volume of traps may be difficult.

7.4 SYSTEM INTEGRATION TECHNIQUES

One of the objectives of integration testing is to combine the software modules into a working system so that system-level tests can be performed on the complete system. Integration testing need not wait until all the modules of a system are coded and unit tested. Instead, it can begin as soon as the relevant modules are available. A module is said to be available for combining with other modules when the module's *check-in request form*, to be discussed in this section, is ready. Some common approaches to performing system integration are as follows:

- Incremental
- Top down
- Bottom up
- Sandwich
- Big bang

In the remainder of this section, we explain the above approaches.

7.4.1 Incremental

In this approach, integration testing is conducted in an incremental manner as a series of test cycles as suggested by Deutsch [3]. In each test cycle, a few more modules are integrated with an existing and tested build to generate a larger build. The idea is to complete one cycle of testing, let the developers fix all the errors found, and continue the next cycle of testing. The complete system is built incrementally, cycle by cycle, until the whole system is operational and ready for system-level testing.

The system is built as a succession of layers, beginning with some core modules. In each cycle, a new layer is added to the core and tested to form a new core. The new core is intended to be self-contained and stable. Here, "self-contained" means containing all the necessary code to support a set of functions, and "stable" means that the subsystem (i.e., the new, partial system) can stay up for 24 hours without any anomalies. The number of system integration test cycles and the total integration time are determined by the following parameters:

- Number of modules in the system
- Relative complexity of the modules (cyclomatic complexity)
- Relative complexity of the interfaces between the modules
- Number of modules needed to be clustered together in each test cycle
- Whether the modules to be integrated have been adequately tested before
- Turnaround time for each test–debug–fix cycle

Constructing a build is a process by which individual modules are integrated to form an interim software image. A *software image* is a compiled software binary. A *build* is an interim software image for internal testing within the organization. Eventually, the final build will be a candidate for system testing, and such a tested system is released to the customers. Constructing a software image involves the following activities:

- Gathering the latest unit tested, authorized versions of modules
- Compiling the source code of those modules
- Checking in the compiled code to the repository
- Linking the compiled modules into subassemblies
- Verifying that the subassemblies are correct
- Exercising version control

A simple build involves only a small number of modules being integrated with a previously tested build on a reliable and well-understood platform. No special tool or procedure needs to be developed and documented for a simple build. On the other hand, organized, well-documented procedures are applied for complex builds. A build process becomes complicated if a large number of modules are integrated together, and a significant number of those modules are new with complex interfaces. These interfaces can be between software modules and hardware devices, across platforms, and across networks. For complex builds, a version control tool is highly recommended for automating the build process and for fast turnaround of a test–debug–fix cycle.

Creating a daily build [4] is very popular in many organizations because it facilitates to a faster delivery of the system. It puts emphasis on small incremental testing, steadily increasing the number of test cases, and regression testing from build to build. The integrated system is tested using automated, reusable test cases. An effort is made to fix the defects that were found during the testing cycle. A new version of the system is constructed from the existing, revised, and newly developed modules and is made available for retesting. Prior versions of the build are retained for reference and rollback. If a defect is not found in a module of a build in which the module was introduced, the module will be carried forward from build to build until one is found. Having access to the version where the defective module was originally introduced is useful in debugging and fixing, limiting the side effects of the fixes, and performing a root cause analysis. During system development, integration, and testing, a typical practice is to retain the past 7–10 builds.

The software developer fills out a check-in request form before a new software module or a module with an error fix is integrated into a build. The form is reviewed by the build engineering group for giving approval. Once it is approved, the module can be considered for integration. The main portions of a check-in form are given in Table 7.1. The idea behind having a check-in request mechanism is fourfold:

1. All the files requiring an update must be identified and known to other team members.
2. The new code must have been reviewed prior to its integration.
3. The new code must have been unit tested.
4. The scope of the check-in is identified.

A release note containing the following information accompanies a build:

- What has changed since the last build?
- What outstanding defects have been fixed?
- What are the outstanding defects in the build?
- What new modules or features have been added?

TABLE 7.1 Check-in Request Form

Author	Name of the person requesting this check-in
Today's date	month, day, year
Check-in request date	month, day, year
Category (identify all that apply)	New Feature: (Y, N) Enhancement: (Y, N) Defect: (Y, N); if yes: defect numbers: Are any of these major defects: (Y, N) Are any of these moderate defects: (Y, N)
Short description of check-in	Describe in a short paragraph the feature, the enhancement, or the defect fixes to be checked in.
Number of files to be checked in	Give the number of files to be checked in. Include the file names, if possible.
Code reviewer names	Provide the names of the code reviewers.
Command line interface changes made	(Y, N); if yes, were they: Documented? (Y, N) Reviewed? (Y, N, pending)
Does this check-in involve changes to global header?	(Y, N); if yes, include the header file names.
Does this check-in involve changes in output logging?	(Y, N); if yes, were they documented? (Y, N)
Unit test description	Description of the unit tests conducted
Comments	Any other comments and issues

- What existing modules or features have been enhanced, modified, or deleted?
- Are there any areas where unknown changes may have occurred?

A test strategy is created for each new build based on the above information. The following issues are addressed while planning a test strategy:

- What test cases need to be selected from the system integration test plan, as discussed in Section 7.6, in order to test the changes? Will these test cases give feature coverage of the new and modified features? If necessary, add new test cases to the system integration test plan.
- What existing test cases can be reused without modification in order to test the modified system? What previously failed test cases should now be reexecuted in order to test the fixes in the new build?
- How should the scope of a partial regression test be determined? A full regression test may not be run on each build because of frequent turnaround of builds. At the least, any earlier test cases which pertain to areas that have been modified must be reexecuted.
- What are the estimated time, resource demand, and cost to test this build? Some builds may be skipped based on this estimate and the current activities, because the integration test engineers may choose to wait for a later build.

7.4.2 Top Down

Systems with hierarchical structures easily lend themselves to top-down and bottom-up approaches to integration. In a hierarchical system, there is a first, top-level module which is decomposed into a few second-level modules. Some of the second-level modules may be further decomposed into third-level modules, and so on. Some or all the modules at any level may be terminal modules, where a terminal module is one that is no more decomposed. An internal module, also known as a nonterminal module, performs some computations, invokes its subordinate modules, and returns control and results to its caller. In top-down and bottom-up approaches, a design document giving the module hierarchy is used as a reference for integrating modules. An example of a module hierarchy is shown in Figure 7.1, where module A is the topmost module; module A has been decomposed into modules B, C, and D. Modules B, D, E, F, and G are terminal modules, as these have not been further decomposed. The top-down approach is explained in the following:

Step 1: Let IM represent the set of modules that have already been integrated and the required stubs. Initially, IM contains the top-level module and stubs corresponding to all the subordinate modules of the top-level

Figure 7.1 Module hierarchy with three levels and seven modules.

module. It is assumed that the top-level module has passed its entry criteria.

Step 2: Choose a stub member M′ in set IM. Let M be the actual module corresponding to stub M′. We obtain a new set CM from IM by replacing stub M′ with M and including in CM all stubs corresponding to the subordinate modules of M. We consider CM to be a union of four sets: {M}, CMs, CMi, CMr, where CMs is the set of stubs, CMi is the set of modules having direct interfaces with M, and CMr is the rest of the modules in CM.

Step 3: Now, test the combined behavior of CM. Testing CM means applying input to the top-level module of the system. It may be noted that though the integration team has access to the top module of the system, all kinds of tests cannot be performed. This is apparent from the fact that CM does not represent the full system. In this step, the integration team tests a subset of the system functions implemented by the actual modules in CM. The integration team performs two kinds of tests:

1. Run test cases to discover any interface defects between M and members of CMi.
2. Perform regression tests to ensure that integration of the modules in the two sets CMi and CMr is satisfactory in the presence of module M. One may note that in previous iterations the interfaces between modules in CMi and CMr were tested and the defects fixed. However, the said tests were executed with M′—a stub of M—and not M. The presence of M in the integrated system up to this moment allows us to test the interfaces between the modules in the combined set of CMi and CMr, because of the possibility of the system supporting more functionalities with M.

   The above two kinds of tests are continued until the integration team is satisfied that there is no known interface error. In case an interface error is discovered, the error must be fixed before moving on to the next step.

Step 4: If the set CMs is empty, then stop; otherwise, set IM = CM and go to step 2.

Now, let us consider an example of top-down integration using Figure 7.1. The integration of modules A and B by using stubs C′ and D′ (represented by grey boxes) is shown in Figure 7.2. Interactions between modules A and B is severely constrained by the dummy nature of C′ and D′. The interactions between A and B are concrete, and, as a consequence, more tests are performed after additional modules are integrated. Next, as shown in Figure 7.3, stub D′ has been replaced with its actual instance D. We perform two kinds of tests: first, test the interface between A and D; second, perform regression tests to look for interface defects between A and B in the presence of module D. Stub C′ has been replaced with the actual module C, and new stubs E′, F′, and G′ have been added to the integrated system (Figure 7.4). We perform tests as follows: First, test the interface between A and C; second, test the combined modules A, B, and D in the presence of C (Figure 7.4). The rest of the integration process is depicted in Figures 7.5 and 7.6 to obtain the final system of Figure 7.7.

The advantages of the top-down approach are as follows:

- System integration test (SIT) engineers continually observe system-level functions as the integration process continues. How soon such functions are observed depends upon their choice of the order in which modules are integrated. Early observation of system functions is important because it gives them better confidence.
- Isolation of interface errors becomes easier because of the incremental nature of top-down integration. However, it cannot be concluded that an interface error is due to a newly integrated module M. The interface error

Figure 7.2 Top-down integration of modules A and B.

Figure 7.3 Top-down integration of modules A, B, and D.

Figure 7.4 Top-down integration of modules A, B, D, and C.

Figure 7.5 Top-down integration of modules A, B, C, D, and E.

Figure 7.6 Top-down integration of modules A, B, C, D, E, and F.

Figure 7.7 Top-down integration of modules A, B, C, D, E, F and G.

may be due to faulty implementation of a module that was already integrated much earlier. This is possible because earlier tests were conducted with or without a stub for M, and the full capability of M simply allowed the test engineers to conduct more tests that were possible due to M.

- Test cases designed to test the integration of a module M are reused during the regression tests performed after integrating other modules.
- Since test inputs are applied to the top-level module, it is natural that those test cases correspond to system functions, and it is easier to design those test cases than test cases designed to check internal system functions. Those test cases can be reused while performing the more rigorous, system-level tests.

The limitations of the top-down approach are as follows:

- Until a certain set of modules has been integrated, it may not be possible to observe meaningful system functions because of an absence of lower level modules and the presence of stubs. Careful analysis is required to identify an ordering of modules for integration so that system functions are observed as early as possible.

- Test case selection and stub design become increasingly difficult when stubs lie far away from the top-level module. This is because stubs support limited behavior, and any test run at the top level must be constrained to exercise the limited behavior of lower level stubs.

7.4.3 Bottom Up

In the bottom-up approach, system integration begins with the integration of lowest level modules. A module is said to be at the lowest level if it does not invoke another module. It is assumed that all the modules have been individually tested before. To integrate a set of lower level modules in this approach, we need to construct a test driver module that invokes the modules to be integrated. Once the integration of a desired group of lower level modules is found to be satisfactory, the driver is replaced with the actual module and one more test driver is used to integrate more modules with the set of modules already integrated. The process of bottom-up integration continues until all the modules have been integrated.

Now we give an example of bottom-up integration for the module hierarchy of Figure 7.1. The lowest level modules are E, F, and G. We design a test driver to integrate these three modules, as shown in Figure 7.8. It may be noted that modules E, F, and G have no direct interfaces among them. However, return values generated by one module is likely to be used in another module, thus having an indirect interface. The test driver in Figure 7.8 invokes modules E, F, and G in a way similar to their invocations by module C. The test driver mimics module C to integrate E, F, and G in a limited way, because it is much simpler in capability than module C. The test driver is replaced with the actual module—in this case C—and a new test driver is used after the testers are satisfied with the combined behavior of E, F, and G (Figure 7.9). At this moment, more modules, such as B and D, are integrated with the so-far integrated system. The test driver mimics the behavior of module A. We need to include modules B and D because those are invoked by A and the test driver mimics A (Figure 7.9). The test driver is replaced with module A (Figure 7.10), and further tests are performed after the testers are satisfied with the integrated system shown in Figure 7.9.

The advantages of the bottom-up approach are as follows. If the low-level modules and their combined functions are often invoked by other modules, then it is more useful to test them first so that meaningful effective integration of other modules can be done. In the absence of such a strategy, the testers write stubs to emulate the commonly invoked low-level modules, which will provide only a limited test capability of the interfaces.

Figure 7.8 Bottom-up integration of modules E, F, and G.

Figure 7.9 Bottom-up integration of modules B, C, and D with E, F, and G.

Figure 7.10 Bottom-up integration of module A with all others.

The disadvantages of the bottom-up approach are as follows:

- Test engineers cannot observe system-level functions from a partly integrated system. In fact, they cannot observe system-level functions until the top-level test driver is in place.
- Generally, major design decisions are embodied in top-level modules, whereas most of the low-level modules largely perform commonly known input–output functions. Discovery of major flaws in system design may not be possible until the top-level modules have been integrated.

Now we compare the top-down and bottom-up approaches in the following:

- **Validation of Major Design Decisions:** The top-level modules contain major design decisions. Faults in design decisions are detected early if integration is done in a top-down manner. In the bottom-up approach, those faults are detected toward the end of the integration process.
- **Observation of System-Level Functions:** One applies test inputs to the top-level module, which is akin to performing system-level tests in a very limited way in the top-down approach. This gives an opportunity to the SIT personnel and the development team to observe system-level functions early in the integration process. However, similar observations can be done in the bottom-up approach only at the end of system integration.
- **Difficulty in Designing Test Cases:** In the top-down approach, as more and more modules are integrated and stubs lie farther away from the top-level module, it becomes increasingly difficult to design stub behavior and test input. This is because stubs return predetermined values, and a

test engineer must compute those values for a given test input at the top level. However, in the bottom-up approach, one designs the behavior of a test driver by simplifying the behavior of the actual module.

- **Reusability of Test Cases:** In the top-down approach, test cases designed to test the interface of a newly integrated module is reused in performing regression tests in the following iteration. Those test cases are reused as system-level test cases. However, in the bottom-up approach, all the test cases incorporated into test drivers, except for the top-level test driver, cannot be reused. The top-down approach saves resources in the form of time and money.

7.4.4 Sandwich and Big Bang

In the sandwich approach, a system is integrated by using a mix of the top-down and bottom-up approaches. A hierarchical system is viewed as consisting of three layers. The bottom layer contains all the modules that are often invoked. The bottom-up approach is applied to integrate the modules in the bottom layer. The top layer contains modules implementing major design decisions. These modules are integrated by using the top-down approach. The rest of the modules are put in the middle layer. We have the advantages of the top-down approach where writing stubs for the low-level module is not required. As a special case, the middle layer may not exist, in which case a module falls either in the top layer or in the bottom layer. On the other hand, if the middle layer exists, then this layer can be integrated by using the big-bang approach after the top and the bottom layers have been integrated.

In the big-bang approach, first all the modules are individually tested. Next, all those modules are put together to construct the entire system which is tested as a whole. Sometimes developers use the big-bang approach to integrate small systems. However, for large systems, this approach is not recommended for the following reasons:

- In a system with a large number of modules, there may be many interface defects. It is difficult to determine whether or not the cause of a failure is due to interface errors in a large and complex system.
- In large systems, the presence of a large number of interface errors is not an unlikely scenario in software development. Thus, it is not cost effective to be optimistic by putting the modules together and hoping it will work.

Solheim and Rowland [5] measured the relative efficacy of top-down, bottom-up, sandwich, and big-bang integration strategies for software systems. The empirical study indicated that top-down integration strategies are most effective in terms of defect correction. Top-down and big-bang strategies produced the most reliable systems. Bottom-up strategies are generally least effective at correcting defects and produce the least reliable systems. Systems integrated by the sandwich strategy are moderately reliable in comparison.

7.5 SOFTWARE AND HARDWARE INTEGRATION

A component is a fundamental part of a system, and it is largely independent of other components. Many products require development of both hardware and software components. These two kinds of components are integrated to form the complete product. In addition, a third kind of component, a product documentation, is developed in parallel with the first two components. A product documentation is an integration of different kinds of individual documentations. The overall goal is to reduce the time to market of the product by removing the sequential nature of product development processes.

On the hardware side, the individual hardware modules, or components, are diverse in nature, such as a chassis, a printed circuit board, a power supply, a fan tray for cooling, and a cabinet to hold the product. On the documentation side, the modules that are integrated together include an installation manual, a troubleshooting guide, and a user's manual in more than one natural language.

It is essential to test both the software and the hardware components individually as much as possible before integrating them. In many products, neither component can be completely tested without the other. Usually, the entry criteria for both the hardware and software components are establishedand satisfied before beginning to integrate those components. If the target hardware is not available at the time of system integration, then a hardware emulator is developed. The emulator replaces the hardware platform on which the software is tested until the real hardware is available. However, there is no guarantee that the software will work on the real hardware even if it worked on the emulator.

Integration of hardware and software components is often done in an iterative manner. A software image with a minimal number of core software modules is loaded on the prototype hardware. In each step, a small number of tests are performed to ensure that all the desired software modules are present in the build. Next, additional tests are run to verify the essential functionalities. The process of assembling the build, loading on the target hardware, and testing the build continues until the entire product has been integrated. If a problem is discovered early in the hardware/software integration and the problem can be resolved easily, then the problem is fixed without any delay. Otherwise, integration of software and hardware components may continue in a limited way until the root cause of the problem is found and analyzed. The integration is delayed until the fixes, based on the outcome of the root cause analysis, are applied.

7.5.1 Hardware Design Verification Tests

A hardware engineering process is viewed as consisting of four phases: (i) planning and specification, (ii) design, prototype implementation, and testing, (iii) integration with the software system, and (iv) manufacturing, distribution, and field service. Testing of a hardware module in the second phase of hardware development without software can be conducted to a limited degree. A hardware design verification test (DVT) plan is prepared and executed by the hardware group before integration with the software system. The main hardware tests are discussed below.

Diagnostic Test Diagnostic tests are the most fundamental hardware tests. Such tests are often imbedded in the basic input–output system (BIOS) component and are executed automatically whenever the system powers up. The BIOS component generally resides on the system's read-only memory (ROM). This test is performed as a kind of sanity test of the hardware module. A good diagnostic test covers all the modules in the system. A diagnostic test is the first test performed to isolate a faulty hardware module.

Electrostatic Discharge Test The concept of electrostatic discharge (ESD) testing is very old and it ensures that the system operation is not susceptible to ESD after having taken commonly accepted precautions. There are three common industry standards on ESD testing based on three different models: the human body model (HBM), the machine model (MM), and the charged device model (CDM). The HBM is the oldest one, and it is the most widely recognized ESD model. It was developed at a time when most ESD damages occurred as people touched hardware components without proper grounding. The capacitance and impedance of the human body vary widely, so the component values in the model were arbitrarily set to facilitate comparative testing. Devices damaged by the HBM generally have thermally damaged junctions, melted metal lines, or other types of damages caused by a high peak current and a high charge dissipated over several hundred nanoseconds. This model still applies whenever people handle devices so one should perform HBM testing on all new devices.

The MM is used primarily in Japan and Europe. This model was developed originally as a "worst-case" HBM to duplicate the type of failures caused by automated pick-and-place machines used to assemble printed circuit boards (PCBs). The model simulates a low-impedance machine that discharges a moderately high capacitance (e.g., 200 pF) through a device. A discharge produced using the MM can cause damage at relatively low voltages. Finally, the CDM reproduces realistic ESD damages that occur in small, plastic-packaged devices. As a packaged device slides on a surface, it accumulates charge due to triboelectric (friction) action between the plastic body and the surface. Thus, the device picks up a charge that produces a potential. In the HBM and the MM, something external to the device accumulates the charge. For small devices, the potential can be surprisingly high. Potentials of at least 650 V are needed to duplicate the observed damage.

Electromagnetic Emission Test Tests are conducted to ensure that the system does not emit excessive radiation to impact operation of adjacent equipments. Similarly, tests are conducted to ensure that the system does not receive excessive radiation to impact its own operation. The emissions of concern are as follows:

- Electric field radiated emissions
- Magnetic field radiated emissions
- Alternating-current (AC) power lead conducted emission (voltage)
- AC and direct-current (DC) power and signal lead conducted emission (current)
- Analog voice band lead conducted emission

Electrical Test A variety of electrical tests are performed on products with a hardware component. One such test is called "signal quality" testing in which different parts of the system are checked for any inappropriate voltages or potential current flows at the externally accessible ports and peripherals. Another type of electrical test is observing how the system behaves in response to various types of power conditions such as AC, DC, and batteries. In addition, tests are conducted to check the safety limits of the equipment when it is exposed to abnormal conditions. An abnormal condition can result from lightning surges or AC power faults.

Thermal Test Thermal tests are conducted to observe whether or not the system can stand the temperature and humidity conditions it will experience in both operating and nonoperating modes. The system is placed in a thermal chamber and run through a series of temperature and humidity cycles. The heat producing components, such as CPU and Ethernet cards, are instrumented with thermal sensors to verify whether the components exceed their maximum operating temperatures. A special kind of thermal test is thermal shock, where the temperature changes rapidly.

Environmental Test Environmental tests are designed to verify the response of the system to various types of strenuous conditions encountered in the real world. One such test involves shock and vibration from adjacent constructions and highways. Nearby heavy machinery, heavy construction, heavy industrial equipment, truck/train traffic, or standby generators can result in low-frequency vibration which can induce intermittent problems. It is important to know if such low-frequency vibration will cause long-term problems such as connectors becoming loose or short-term problems such as a disk drive crashing. Environmental tests also include other surprises the system is likely to encounter. For example, in a battlefield environment, the computers and the base transceiver stations are often subjected to smoke, sand, bullets, fire, and other extreme conditions.

Equipment Handling and Packaging Test These tests are intended to determine the robustness of the system to normal shipping and installation activities. Good pack and packaging design ensures that the shipping container will provide damage-free shipment of the system. Early involvement of these design skills will provide input on the positioning of handles and other system protrusions that can be the sources of failure. Selection of reasonable metal thickness and fasteners will provide adequate performance of systems to be installed into their final location.

Acoustic Test Acoustic noise limits are specified to ensure that personnel can work near the system without exceeding the safety limits prescribed by the local Occupational Safety and Health Administration (OSHA) agency or other negotiated levels. For example, noise levels of spinning hard disks, floppies, and other drives must be tested against their limits.

Safety Test Safety tests are conducted to ensure that people using or working on or near the system are not exposed to hazards. For example, many portable computers contain rechargeable batteries which frequently include dangerous toxic substances such as cadmium and nickel. Adequate care must be taken to ensure that these devices do not leak the dangerous chemicals under any circumstances.

Reliability Test Hardware modules tends to fail over time. It is assumed that (i) modules have constant failure rates during their useful operating life periods and (ii) module failure rates follow an exponential law of distribution. Failure rate is often measured in terms of the mean time between failures (MTBF), and it is expressed as MTBF = total time/number of failures. The probability that a module will work for some time T without failure is given by $R(T) = \exp(-T/\text{MTBF})$. The MTBF metric is a reliability measurement metric for hardware modules. It is usually given in units of hours, days, or months.

The MTBF for a module or a system is derived from various sources: laboratory tests, actual field failure data, and prediction models. Another way of calculating the reliability and lifetime of a system is to conduct highly accelerated life tests (HALTs). The HALTs rely on the principle of *logarithmic time compression* to simulate a system's entire life in just a few days. This is done by applying a much higher level of stress than what exists in actual system use. A high level of stress forces failures to occur in significantly less time than under normal operating conditions. The HALTs generally include rapid temperature cycling, vibrating on all axes, operating voltage variations, and changing clock frequency until the system fails. The HALTs require only a few units of the product and a short testing period to identify the fundamental limits of the technologies in use. Generally, every weak point must be identified and fixed (i.e., redesigned) if it does not meet the system's specified limits. Understanding the concept of product reliability is important for any organization if the organization intends to offer warranty on the system for an extended period of time. One can predict with high accuracy the exact cost associated with the returns over a limited and an extended period of warranty. For example, for a system with an MTBF of 250,000 hours and an operating time of interest of five years (438,000 hours), we have $R(438{,}000) = \exp(-43{,}800/250{,}000) = 0.8392$, which says that there is a probability of 0.8932 that the product will operate for five years without a failure. Another interpretation of the quantity is that 83.9% of the units in the field will still be working at the end of five years. In other words, 16.1% of the units need to be replaced within the first five years.

7.5.2 Hardware and Software Compatibility Matrix

The hardware and software compatibility information is maintained in the form of a compatibility matrix. Such a matrix documents the compatibility between different revisions of the hardware and different versions of the software and is used for official release of the product. An engineering change order (ECO) is a formal document that describes a change to the hardware or software. An ECO document includes the hardware/software compatibility matrix and is distributed to

TABLE 7.2 Example Software/Hardware Compatibility Matrix

Software Release	RNC Hardware Version	RN Hardware Version	EMS Hardware Version	PDSN Hardware Version	Tested by SIT	Tested by ST
2.0	hv1.0	hv2.0	hv3.2	hv2.0.3	Yes	Yes
2.5	hv2.0 and hv1.0	hv2.0	hv4.0 and hv3.2	hv3.0 and hv2.0.3	Yes	Yes
3.0	hv3.0	hv3.0 and hv2.0	hv5.0	hv4.0 and hv3.0	Yes	Not yet
3.0	hv4.0 and hv3.0	hv3.0	hv5.0	hv4.5	Not yet	Not yet
Not yet decided	hv4.0 and hv3.0	hv3.0	hv6.0	hv5.0	Not yet	Not yet

the operation, customer support, and sales teams of the organization. An example compatibility matrix for a 1xEV-DO wireless data network, discussed in Chapter 8, is given in Table 7.2.

In the following, we provide the hardware and software ECO approval process in an organization. The first scenario describes the ECO process to incorporate a new hardware in the product. The second scenario describes the ECO process for a software revision.

Scenario 1 A hardware ECO process is shown in Figure 7.11. Assume that the hardware group needs to release a new revision of a hardware or has to recommend a new revision of an original equipment manufacturer (OEM) hardware to the operation/manufacturing group. The steps of the ECO process to incorporate the new hardware in the product are as follows:

1. The hardware group issues a design change notification for the new hardware revision. This notification includes identification of specific hardware changes likely to impact the software and incompatibilities between the revised hardware and other hardware. The software group reviews the notification with the hardware group to assess the impact of the hardware changes and to identify software changes that affect the hardware structure.
2. The hardware group creates an ECO and reviews it with the change control board (CCB) to ensure that all impacts of the ECO are understood and agreed upon and that the version numbering rules for software components are followed. The CCB constitutes a group of individuals from multiple departments responsible for reviewing and approving each ECO.
3. The ECO is released, and the hardware group updates the hardware/software compatibility matrix based on the information received from the review process.

Figure 7.11 Hardware ECO process.

4. The system testing group updates the compatibility matrix after it has tested a given combination of released hardware and software versions.

Scenario 2 A software release ECO process is shown in Figure 7.12. Assume that the group needs to release a new version of software to the operation/manufacturing group. The steps of the ECO process to incorporate the new version of the software product are as follows:

1. The system integration group releases a build with a release note identifying the hardware compatibility information to the system test group.
2. The system test group tests the build and notifies the software group and other relevant groups in the organization of the results of the tests.
3. The system test group deems a particular build to be viable for customer release. The system test group calls a cross-functional readiness review meeting to ensure, by verifying the test results, that all divisions of the organization are prepared for an official release.
4. The software group writes an ECO to officially release the software build and reviews it with the CCB after the readiness review is completed.

Figure 7.12 Software ECO process.

The build is considered to be released after the ECO is approved and documented.

5. The software group updates the hardware/software compatibility matrix with information about the new release.
6. The system test group updates the compatibility matrix after it has tested a given combination of released hardware and software versions.

7.6 TEST PLAN FOR SYSTEM INTEGRATION

System integration requires a controlled execution environment, much communication between the developers and the test engineers, judicious decision making along the way, and much time, on the order of months, in addition to the fundamental tasks of test design and test execution. Integrating a large system is a challenging task, which is handled with much planning in the form of developing a SIT plan. A useful framework for preparing an SIT plan is outlined in Table 7.3.

TABLE 7.3 Framework for SIT Plan

1. Scope of testing
2. Structure of integration levels
  a. Integration test phases
  b. Modules or subsystems to be integrated in each phase
  c. Building process and schedule in each phase
  d. Environment to be set up and resources required in each phase
3. Criteria for each integration test phase *n*
  a. Entry criteria
  b. Exit criteria
  c. Integration Techniques to be used
  d. Test configuration set-up
4. Test specification for each integration test phase
  a. Test case ID number
  b. Input data
  c. Initial condition
  d. Expected results
  e. Test procedure
    How to execute this test?
    How to capture and interpret the results?
5. Actual test results for each integration test phase
6. References
7. Appendix

In the scope of testing section, one summarizes the system architecture. Specifically, the focus is on the functional, internal, and performance characteristics to be tested. System integration methods and assumptions are included in this section.

The next section, structure of integration levels, contains four subsections. The first subsection explains the division of integration testing into different phases, such as functional, end-to-end, and endurance phases. The second subsection describes the modules to be integrated in each of the integration phases. The third subsection describes the build process to be followed: daily build, weekly build, biweekly build, or a combination thereof. A schedule for system integration is given in the third subsection. Specifically, one identifies the start and end dates for each phase of testing. Moreover, the availability windows for unit-tested modules are defined. In the fourth subsection, the test environment and the resources required are described for each integration phase. The hardware configuration, emulators, software simulators, special test tools, debuggers, overhead software (i.e., stubs and drivers), and testing techniques are discussed in the fourth subsection.

An important decision to be made for integration testing is establishing the start and stop dates of each phase of integration testing. The start date and stop date for a phase are specified in terms of *entry criteria* and *exit criteria*, respectively. These criteria are described in the third section of the plan. A framework for defining entry criteria to start system integration is given in Table 7.4. Similarly, the exit criteria for system integration are given in Table 7.5. Test configuration and integration techniques (e.g., top down or bottom up) to be used in each of these phases are described in this section.

The test specification section describes the test procedure to be followed in each integration phase. The detailed test cases, including the input and expected outcome for each case, are documented in the test specification section. The history of actual test results, problems, or peculiarities is recorded in the fifth section of the SIT plan. Finally, references and an appendix, if any, are included in the test plan.

System integration testing is performed in phases of increasing complexity for better efficiency and effectiveness. In the first phase interface integrity and functional validity within a system are tested. In the second phase end-to-end and pairwise tests are conducted. Finally, in the third phase, stress and endurance tests are performed. Each of the system integration phases identified in the SIT plan delineates a broad functionality category within the software structure, and it can be related to a specific domain of the system software structure. The categories

TABLE 7.4 Framework for Entry Criteria to Start System Integration

Softwave functional and design specifications must be writen, reviewed, and approved.
Code is reviewed and approved.
Unit test plan for each module is written, reviewed, and executed.
All of the unit tests passed.
The entire check-in request form must be completed, submitted, and approved.
Hardware design specification is written, reviewed, and approved.
Hardware design verification test is written, reviewed, and executed.
All of the design verification tests passed.
Hardware/software integration test plan is written, reviewed, and executed.
All of the hardware/software integrated tests passed.

TABLE 7.5 Framework for System Integration Exit Criteria

All code is completed and frozen and no more modules are to be integrated.
All of the system integration tests passed.
No major defect is outstanding.
All the moderate defects found in the SIT phase are fixed and retested.
Not more than 25 minor defects are outstanding.
Two weeks system uptime in system integration test environment without any anomalies, i.e., crashes.
System integration tests results are documented.

of system integration tests and the corresponding test cases discussed below are applicable for the different test phases.

Interface Integrity Internal and external interfaces are tested as each module is integrated into the structure. When two modules are integrated, the communication between them is called internal, whereas when they communicate with the outside environment, it is called external. Tests that are required to be designed to reveal interface errors are discussed in Section 7.2. An important part of interface testing is to map an incoming message format to the expected message format of the receiving module and ensure that the two match. Tests are designed for each message passing through the interface between two modules. Essentially, tests must ensure that:

- The number of parameters sent in a message agree with the number of parameters expected to be received.
- The parameter order in the messages match the order expected.
- The field sizes and the data types match.
- The boundaries of each data field in a message match the expected boundaries.
- When a message is generated from stored data prior to being sent, the message truly reflects the stored data.
- When a received message is stored, data copying is consistent with the received message.

Functional Validity Tests are designed to uncover functional errors in each module after it is integrated with the system. Errors associated with local or global data structures are uncovered with such tests. Selected unit tests that were designed for each module are reexecuted during system integration by replacing the stubs with their actual modules.

End-to-End Validity Tests are performed to ensure that a completely integrated system works together from end to end. Interim checkpoints on an end-to-end flow provide a better understanding of internal mechanisms. This helps in locating the sources of failures.

Pairwise Validity Tests are performed to ensure that any two systems work properly when connected together by a network. The difference between pairwise tests and end-to-end tests lies in the emphasis and type of test cases. For example, a toll-free call to an 800 number is an end-to-end test of a telephone system, whereas a connectivity test between a handset and local private branch exchange (PBX) is a pairwise test.

Interface Stress Stress is applied at the module level during the integration of the modules to ensure that the interfaces can sustain the load. On the other hand, full-scale system stress testing is performed at the time of system-level testing. The following areas are of special interest during interface stress testing:

- **Error Handling:** Trigger errors that should be handled by the modules.
- **Event Handling:** Trigger events (e.g., messages, timeouts, callbacks) that should be handled by the modules.
- **Configuration:** Repeatedly add, modify, and delete managed objects.
- **Logging:** Turn on the logging mechanism during stress tests to ensure proper operation for boundary conditions.
- **Module Interactions:** Run tests repeatedly that stress the interactions of a module with other modules.
- **CPU Cycles:** Induce high CPU utilization by using a CPU overload tool; pay attention to any resulting queue overflows and other producer/consumer overrun conditions.
- **Memory/Disk Usage:** Artificially reduce the levels of heaps and/or memory buffers and disk space;
- **Starvation:** Ensure that the processes or tasks are not starved; otherwise eventually the input queues overflow for the starved processes.
- **Resource Sharing:** Ensure that resources, such as heap and CPU, are shared among processes without any contention and bottlenecks.
- **Congestion:** Run tests with a mechanism that randomly discards packets; test modules with congested links.
- **Capacity:** Run tests to ensure that modules can handle the maximum numbers of supporting requirements such as connections and routes.

System Endurance A completely integrated system is expected to stay up continuously for weeks without any crashes. In the case of communication protocol systems, formal rules govern that two systems communicate with each other via an interface, that is, a communication channel. The idea here is to verify that the format and the message communication across the interface of the modules works for an extended period.

7.7 OFF-THE-SHELF COMPONENT INTEGRATION

Instead of developing a software component from scratch, organizations occasionally purchase off-the-shelf (OTS) components form third-party vendors and integrate them with their own components [6]. In this process, organizations create less expensive software systems. A major issue that can arise while integrating different components is mismatches among code pieces developed by different parties usually unaware of each other [7, 8].

Vigder and Dean [9] have presented elements of an architecture for integration and have defined rules that facilitate integration of components. They have identified a useful set of supporting components for integrating the actual, serving components. The supporting components are *wrappers*, *glue*, and *tailoring*.

A wrapper is a piece of code that one builds to isolate the underlying components from other components of the system. Here *isolate* means putting restrictions around the underlying component to constrain its capabilities. A glue component provides the functionality to combine different components. Component tailoring refers to the ability to enhance the functionality of a component. Tailoring is done by adding some elements to a component to enrich it with a functionality not provided by the vendor. Tailoring does not involve modifying the source code of the component. An example of tailoring is "scripting," where an application can be enhanced by executing a script upon the occurrence of some event. Rine et al. [10] have proposed the concept of adapters to integrate components. An adapter is associated with each component; the adapter runs an interaction protocol to manage communications among the components. Components request services from others through their associated adapters. An adapter is responsible for resolving any syntactic interface mismatch.

7.7.1 Off-the-Shelf Component Testing

Buyer organizations perform two types of testing on an OTS component before purchasing: (i) acceptance testing of the OTS component based on the criteria discussed in Chapter 14 and (ii) integration of the component with other components developed in-house or purchased from a third party. The most common cause of problems in the integration phase is inadequate acceptance testing of the OTS component. A lack of clear documentation of the system interface and less cooperation from the vendor may create an ordeal in integration testing, debugging, and fixing defects. Acceptance testing of an OTS component requires the development and execution of an acceptance test plan based on the acceptance criteria for the candidate component. All the issues are resolved before the system integration process begins. During integration testing, additional software components, such as a glue or a wrapper, can be developed to bind an OTS component with other components for proper functioning. These new software components are also tested during the integration phase. Integration of OTS components is a challenging task because of the following characteristics identified by Basili and Boehm [11]:

The buyer has no access to the source code.

The vendor controls its development.

The vendor has nontrivial installed base.

Voas [12] proposed three types of testing techniques to determine the suitability of an OTS component:

- **Black-Box Component Testing:** This is used to determine the quality of the component.
- **System-Level Fault Injection Testing:** This is used to determine how well a system will tolerate a failing component. System-level fault injection does not demonstrate the reliability of the system; instead, it can predict the behavior of the system if the OTS component fails.

- **Operational System Testing:** This kind of test is used to determine the tolerance of a software system when the OTS component is functioning correctly. Operational system testing is conducted to ensure that an OTS component is a good match for the system.

The OTS components produced by the vendor organizations are known as commercial off-the-shelf (COTS) components. A COTS component is defined by Szyperski et al. [13] (p. 34) as "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." Interfaces are the access points of COTS components through which a client component can request a service declared in an interface of the service providing component. Weyuker [14] recommends that vendors building COTS components must try to envision many possible uses of the component and develop a comprehensive range of test scenarios. A component should be defect free because prudent buyers will perform significant amount of testing while integrating the component with their system. If potential buyers encounter a large number of defects in a COTS component, they may not accept the component. Therefore, it is in the best interest of the component builder to ensure that components are thoroughly tested. Several related artefacts should be archived and/or modified for each COTS component, because a potential buyer may demand these artifacts to be of high quality. The artefacts to be archived include the following:

- Individual requirements, including the pointers between the software functional specification and the corresponding implementations: This information makes it easy to track when either the code or the specification is modified, which implies that the specification remains up to date.
- The test suite, including the requirement traceability matrix: This will show which part of the functionality is tested inadequately or not at all. In addition, it identifies the test cases that (i) must be executed as regression tests or (ii) need to be updated when the component undergoes a change.
- The individual pass–fail result of each test case in the test suite: This indicates the quality of the COTS component.
- The details of individual test cases, including input and expected output: This facilitates regression testing of changes to a component.
- The system test report corresponding to the final release of the COTS component, which includes performance characteristics, scalability limitation, stability observations, and interoperability of the COTS component: This document is a useful resource for potential buyers before they begin acceptance testing of the COTS components.

7.7.2 Built-in Testing

A component reused in a new application environment requires real-time detection, diagnosis, and handling of software faults. Built-in test (BIT) methods for producing

self-testable software components hold potential for detecting faults during run time. A software component can contain test cases or can possess facilities that are capable of generating test cases which can be accessed by a component user on demand [15]. The corresponding capabilities allowing this are called built-in testing capabilities of software components. In the BIT methodology, testability is incorporated into software components, so that testing and maintenance can be self-contained.

Wang et al. [16] have proposed a BIT model that can operate in two modes, namely, *normal* mode and *maintenance* mode. In the normal mode, the BIT capabilities are transparent to the component user, and the component does not differ from other non-BIT-enabled components. In the maintenance mode, however, the component user can test the component with the help of its BIT features. The component user can invoke the respective methods of the component, which execute the test, evaluate autonomously its results, and output the test summary. The authors describe a generic technical framework for enhancing BIT. One of their assumptions is that the component is implemented as a class. A benefit of such an implementation is that the methods for BIT can be passed to a subclass by inheritance.

Hörnstein and Edler [17] have proposed a *component + BIT* architecture comprising three types of components, namely, BIT, testers, and handlers. The BIT components are the BIT-enabled components. These components implement certain mandatory interfaces. Testers are components which access the BIT capabilities of BIT components through the corresponding interfaces and contain the test cases in a certain form. Finally, handlers are components that do not directly contribute to testing but provide recovery mechanisms in case of failures.

7.8 SUMMARY

This chapter began with the objective and a description of system integration testing. One creates a "working version of the system" by putting the modules together in an incremental manner while performing tests to uncover different types of errors associated with interfacing. Next, we explored various levels of granularities in system integration testing: intrasystem testing, intersystem testing, and pairwise testing.

We then examined five types of commonly used system integration techniques: topdown, bottomup, sandwich, bigbang, and incremental. We compared those techniques in detail. The incremental technique is widely used in the industry.

We described the integration of hardware and software components to form a complete product. This led to the discussion of the hardware engineering process and, specifically, of different types of hardware design verification tests: diagnostic, electrostatic discharge, electromagnetic emission, electrical, thermal, environmental, packaging and handling, acoustic, safety, and reliability. Finally, we described two scenarios of an engineering change order process. The two scenarios are used to keep track of the hardware/software compatibility matrix of a released product.

We provided a framework of an integration test plan. The following categories of tests, which are included in an integration test plan, were discussed in detail: interface integrity tests, functional validity tests, end-to-end validity tests, pairwise validity tests, interface stress tests, and endurance tests.

Finally, we described the integration of OTS components with other components. An organization, instead of developing a software component from scratch, may decide to purchase a COTS software from a third-party source and integrate it with its own software system. A COTS component seller must provide BITs along with the components, whereas a buyer organization must perform three types of testing to assess the COTS software: (i) acceptance testing of the OTS software component to determine the quality of the component, (ii) system-level fault injection testing, which is used to determine the tolerance of the software system to a failing OTS component, and (iii) operational system testing, which is used to determine the tolerance of the software system to a properly functioning OTS component.

LITERATURE REVIEW

For those actively involved in software testing or interested in knowing more about common software errors, appendix A of the book by C. Kaner, J. Falk, and H. Q. Nguyen (*Testing Computer Software*, Wiley, New York, 1999) is an excellent repository of real-life software errors. The appendix contains 12 categories of approximately 400 different types of errors with illustrations.

A good discussion of hardware test engineering, such as mechanical, electronics, and accelerated tests, can be found in Patrick O'Connor's book (*Test Engineering: A Concise Guide to Cost-effective Design, Development and Manufacture*, Wiley, New York, 2001). The author (i) describes a broad spectrum of modern methods and technologies in hardware test engineering, (ii) offers principles of cost-effective design, development, and manufacture of products and systems, and (iii) gives a breakdown of why product and systems fail and which methods would best prevent these failures.

Researchers are continuously working on topics related to certification of COTS components. The interested reader is recommended to read the following articles. Each of these articles helps in understanding the issues of software certification and why it is important:

J. Voas, "Certification Reducing the Hidden Costs of Poor Quality," *IEEE Software*, Vol. 16, No. 4, July/August 1999, pp. 22–25.

J. Voas, "Certifying Software for High-Assurance Environments," *IEEE Software*, Vol. 16, No. 4, July/August 1999, pp. 48–54.

J. Voas, "Developing Usage-Based Software Certification Process," *IEEE Computer*, Vol. 16, No. 8, August 2000, pp. 32–37.

S. Wakid, D. Kuhn, and D. Wallace, "Toward Credible IT Testing and Certification," *IEEE Software*, Vol. 16, No. 4, July/August 1999, pp. 39–47.

Readers actively involved in COTS component testing or interested in a more sophisticated treatment of the topic are recommended to read the book edited by S. Beydeda and V. Gruhn (*Testing Commercial-off-the-Shelf Components and Systems*, Springer, Bonn, 2005). The book contains 15 articles that discuss in great detail: (i) testing components context independently, (ii) testing components in the context of a system, and (iii) testing component–based systems. The book lists several excellent references on the subject in a bibliography.

REFERENCES

1. V. R. Basili and B. T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, January 1984, pp. 42–52.
2. D. E. Perry and W. M. Evangelist. An Empirical Study of Software Interface Faults—An Update. In *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, Hawaii Vol. II, IEEE Computer Society Press, Piscataway, NJ, January 1987, pp. 113–126.
3. M. S. Deutsch. *Software Verification and Validation: Realistic Project Approaches*. Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 95–101.
4. M. Cusumano and R. W. Selby. How Microsoft Builds Software. *Communications of the ACM*, June 1997, pp. 53–61.
5. J. A. Solheim and J. H. Rowland. An Empirical Study of Testing and Integration Strategies Using Artificial Software Systems. *IEEE Transactions on Software Engineering*, October 1993, pp. 941–949.
6. S. Mahmood, R. Lai, and Y. S. Kim. Survey of Component-Based Software Development. *IET Software*, April 2007, pp. 57–66.
7. G. T. Heineman and W. T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Reading, MA, 2001.
8. A. Cechich, M. Piattini, and A. Vallecillo. Assessing Component Based Systems. Component Based Software Quality. *Lecture Notes in Computer Science*, Vol. 2693, 2003, pp. 1–20.
9. M. R. Vigder and J. Dean. An Architectural Approach to Building Systems from COTS Software Components. In *Proceedings of the 22nd Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, MD, December 1997, pp. 99–113, NRC41610.
10. D. Rine, N. Nada, and K. Jaber. Using Adapters to Reduce Interaction Complexity in Reusable Component-Based Software Development. In *Proceedings of the 1999 Symposium on Software Reusability*, Los Angeles, CA, ACM Press, New York, May 1999, pp. 37–43.
11. V. R. Basili and B. Boehm. COTS-Based Systems Top 10 List. *IEEE Computer*, May 2001, pp. 91–93.
12. J. Voas. Certifying Off-the-Shelf Software Component. *IEEE Computer*, June 1998, pp. 53–59.
13. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley, Reading, MA, 2002.
14. E. J. Weyuker. Testing Component-Based Software: A Cautionary Tale. *IEEE Software*, September/October 1998, pp. 54–59.
15. S. Beydeda and V. Gruhn. Merging Components and Testing Tools: The Self-Testing COTS Components (STECC) Strategy. In *Proceedings of the 29th EUROMICRO Conference (EUROMICRO'03)*, Belek-Antalya, Turkey, IEEE Computer Society Press, Piscataway, September 2003, pp. 107–114.
16. Y. Wang, G. King, and H. Wickburg. A Method for Built-in Tests in Component-Based Software Maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance and Reengineering (CSMR-99)*, University of Amsterdam, The Netherland, IEEE Computer Society Press, Piscataway, March 1999, pp. 186–189.

17. J. Höornstein and H. Edler. Test Reuse in CBSE Using Built-in Tests. In *Proceedings of the 9th IEEE Conference and Workshops on Engineering of Computer Based Systems*, Workshop on Component-Based Software Engineering, Lund University, Lund, Sweden, IEEE Computer Society Press, Piscataway, 2002.

Exercises

1. Describe the difference between black-box and white-box testing techniques?
2. If a program passes all the black-box tests, it means that the program should work properly. Then, in addition to black-box testing, why do you need to perform white-box testing?
3. Describe the difference between unit testing and integration testing?
4. Why should integration testing be performed? What types of errors can this phase of testing reveal?
5. Discuss the advantages and disadvantages of top-down and bottom-up approaches to integration testing.
6. Does automation of integration tests help the verification of the daily build process? Justify your answer.
7. Using the module hierarchy given in Figure 7.13, show the orders of module integration for the top-down and bottom-up integration approaches. Estimate the number of stubs and drivers needed for each approach. Specify the integration testing activities that can be done in parallel, assuming you have three SIT engineers. Based on the resource needs and the ability to carry out concurrent SIT activities, which approach would you select for this system and why?
8. Suppose that you plan to purchase COTS components and integrate them with your communication software project. What kind of acceptance criteria will you develop to conduct acceptance testing of the COTS components?
9. During integration testing of COTS components with a software system, it may be required to develop a wrapper software around the OTS component to limit what it can do. Discuss the general characteristics that a wrapping software

Figure 7.13 Module hierarchy of software system.

should have in order to be able to integrate COTS with the software system without any problem.

10. Describe the circumstances under which you would apply white-box testing, back-box testing, or both techniques to evaluate a COTS component.
11. For your current test project, develop an integration test plan.
12. Complete Section 5 (i.e., actual test results for each integration test phase) of the integration test plan after executing the integration test cases you developed in exercise 11.

CHAPTER 8

System Test Categories

As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications.
— *Dave Parnas*

8.1 TAXONOMY OF SYSTEM TESTS

The objective of system-level testing, also called system testing, is to establish whether an implementation conforms to the requirements specified by the customers. It takes much effort to guarantee that customer requirements have been met and the system is acceptable. A variety of tests are run to meet a wide range of unspecified expectations as well. As integrated systems, consisting of both hardware and software components, are often used in reality, there is a need to have a much broader view of the behavior of the systems. For example, a telephone switching system not only is required to provide a connection between two users but also is expected to do so even if there are many ongoing connections below a certain upper limit. When the upper limit on the number of simultaneous connections is reached, the system is not expected to behave in an undesired manner. In this chapter, we identify different categories of tests in addition to the core functionality test. Identifying test categories brings us the following advantages:

- Test engineers can accurately focus on different aspects of a system, one at a time, while evaluating its quality.
- Test engineers can prioritize their tasks based on test categories. For example, it is more meaningful and useful to identify the limitations of a system only after ensuring that the system performs all basic functions to the test an engineer's satisfactions. Therefore, stress tests, which thrive to identify the limitations of a system, are executed after functionality tests.
- Planning the system testing phase based on test categorization lets a test engineer obtain a well-balanced test suite. Practical limitations make it difficult to be exhaustive, and economic considerations may restrict the

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

testing process from continuing any further. However, it is important to design a balanced test suite, rather than an unbalanced one with many test cases in one category and no tests in another.

In the following, first we present the taxonomy of system tests (Figure 8.1). Thereafter, we explain each category in detail.

- *Basic tests* provide an evidence that the system can be installed, configured, and brought to an operational state.
- *Functionality tests* provide comprehensive testing over the full range of the requirements within the capabilities of the system.
- *Robustness tests* determine how well the system recovers from various input errors and other failure situations.
- *Interoperability tests* determine whether the system can interoperate with other third-party products.
- *Performance tests* measure the performance characteristics of the system, for example, throughput and response time, under various conditions.
- *Scalability tests* determine the scaling limits of the system in terms of user scaling, geographic scaling, and resource scaling.
- *Stress tests* put a system under stress in order to determine the limitations of a system and, when it fails, to determine the manner in which the failure occurs.

Figure 8.1 Types of system tests.

- *Load and stability tests* provide evidence that the system remains stable for a long period of time under full load.
- *Reliability tests* measure the ability of the system to keep operating for a long time without developing failures.
- *Regression tests* determine that the system remains stable as it cycles through the integration of other subsystems and through maintenance tasks.
- *Documentation tests* ensure that the system's user guides are accurate and usable.
- *Regulatory tests* ensure that the system meets the requirements of government regulatory bodies in the countries where it will be deployed.

8.2 BASIC TESTS

The basic tests (Figure 8.2) give a *prima facie* evidence that the system is ready for more rigorous tests. These tests provide limited testing of the system in relation to the main features in a requirement specification. The objective is to establish that there is sufficient evidence that a system can operate without trying to perform thorough testing. Basic tests are performed to ensure that commonly used functions, not all of which may directly relate to user-level functions, work to our satisfaction. We emphasize the fact that test engineers rely on the proper implementation of these functions to carry out tests for user-level functions. The following are the major categories of subsystems whose adequate testing is called the basic test.

8.2.1 Boot Tests

Boot tests are designed to verify that the system can boot up its software image (or build) from the supported boot options. The boot options include booting from ROM, FLASH card, and PCMCIA (Personal Computer Memory Card International Association) card. The minimum and maximum configurations of the system must be tried while booting. For example, the minimum configuration of a router consists of one line card in its slots, whereas the maximum configuration of a router means that all slots contains line cards.

Figure 8.2 Types of basic tests.

8.2.2 Upgrade/Downgrade Tests

Upgrade/downgrade tests are designed to verify that the system software can be upgraded or downgraded (rollback) in a graceful manner from the previous version to the current version or vice versa. Suppose that the system is running the $(n-1)$th version of the software build and the new nth version of the software build is available. The question is how one upgrades the build from the $(n-1)$th version to the nth version. An upgradation process taking a system from the $(n-1)$th version to the nth version may not be successful, in which case the system is brought back to the $(n-1)$th version. Tests are designed in this subgroup to verify that the system successfully reverts back, that is, rolls back, to the $(n-1)$th version. An upgradation process may fail because of a number of different conditions: user-invoked abort (the user interrupts the upgrade process), in-process network disruption (the network environment goes down), in-process system reboot (there is a power glitch), or self-detection of upgrade failure (this is due to such things as insufficient disk space and version incompatibilities).

8.2.3 Light Emitting Diode Tests

The LED (light emitting diode) tests are designed to verify that the system LED status indicators function as desired. The LEDs are located on the front panels of the systems. These provide visual indication of the module operational status. For example, consider the status of a system chassis: Green indicates that the chassis is operational, off indicates that there is no power, and a blinking green may indicate that one or more of its submodules are faulty. The LED tests are designed to ensure that the visual operational status of the system and the submodules are correct. Examples of LED tests at the system and subsystem levels are as follows:

- System LED test: green = OK, blinking green = fault, off = no power.
- Ethernet link LED test: green = OK, blinking green = activity, off = fault.
- Cable link LED test: green = OK, blinking green = activity, off = fault.
- User defined T1 line card LED test: green = OK, blinking green = activity, red = fault, off = no power.

8.2.4 Diagnostic Tests

Diagnostic tests are designed to verify that the hardware components (or modules) of the system are functioning as desired. It is also known as the built-in self-test (BIST). Diagnostic tests monitor, isolate, and identify system problems without manual troubleshooting. Some examples of diagnostic tests are as follows:

- **Power-On Self-Test (POST):** This is a set of automatic diagnostic routines that are executed during the boot phase of each submodule in the system. The POSTs are intended to determine whether or not the hardware is in a proper state to execute the software image. It is not intended to be comprehensive in the analysis of the hardware; instead, it provides a high

level of confidence that the hardware is operational. The POSTs execute on the following kinds of elements:

Memory

Address and data buses

Peripheral devices

- **Ethernet Loop-Back Test:** This test generates and sends out the desired number, which is a tunable parameter, of packets and expects to receive the same number of Ethernet packets through the loop-back interface—external or internal. If an error occurs (e.g., packet mismatch or timeout), an error message indicating the type of error, its probable cause(s), and recommended action(s) is displayed on the console. The data sent out are generated by a random-number generator and put into a data buffer. Each time a packet is sent, it is selected from a different starting point of the data buffer, so that any two consecutively transmitted packets are unlikely to be identical. These tests are executed to ensure that the Ethernet card is functioning as desired.
- **Bit Error Test (BERT):** The on-board BERT provides standard bit error patterns, which can be transmitted over a channel for diagnostic purpose. BERT involves transmitting a known bit pattern and then testing the transmitted pattern for errors. The ratio of the number of bits with errors to the total number of bits transmitted is called the bit error rate (BER). Tests are designed to configure all BERTs from the command line interface (CLI). These tests are executed to ensure that that the hardware is functioning as desired.

8.2.5 Command Line Interface Tests

The CLI tests are designed to verify that the system can be configured, or provisioned, in specific ways. This is to ensure that the CLI software module processes the user commands correctly as documented. This includes accessing the relevant information from the system using CLI. In addition to the above tests, test scenarios may be developed to verify the error messages displayed.

8.3 FUNCTIONALITY TESTS

Functionality tests (Figure 8.3) verify the system as thoroughly as possible over the full range of requirements specified in the requirements specification document. This category of tests is partitioned into different functionality subgroups as follows.

8.3.1 Communication Systems Tests

Communication systems tests are designed to verify the implementation of the communication systems as specified in the customer requirements specification.

Figure 8.3 Types of functionality tests.

For example, one of the customer requirements can be to support Request for Comment (RFC) 791, which is the Internet Protocol (IP) specification. Tests are designed to ensure that an IP implementation conforms to the RFC791 standard. Four types of communication systems tests are recommended according to the extent to which they provide an indication of conformance [1]:

- *Basic interconnection tests* provide evidence that an implementation can establish a basic connection before thorough testing is performed.
- *Capability tests* check that an implementation provides the observable capabilities based on the static communication systems requirements. The static requirements describes the options, ranges of values for parameters, and timers.
- *Behavior tests* endeavor to verify the dynamic communication systems requirements of an implementation. These are the requirements and options that define the observable behavior of a protocol. A large part of behavior tests, which constitute the major portion of communication systems tests, can be generated from the protocol standards.
- *Systems resolution tests* probe to provide definite "yes" or "no" answers to specific requirements.

8.3.2 Module Tests

Module tests are designed to verify that all the modules function individually as desired within the systems. Mutual interactions among the modules glue these

components together into a whole system. The idea here is to ensure that individual modules function correctly within the whole system. One needs to verify that the system, along with the software that controls these modules, operate as specified in the requirement specification. For example, an Internet router contains modules such as line cards, system controller, power supply, and fan tray. Tests are designed to verify each of the functionalities. For Ethernet line cards, tests are designed to verify (i) autosense, (ii) latency, (iii) collisions, (iv) frame types, and (v) frame lengths. Tests are designed to ensure that the fan status is accurately read, reported by the software, and displayed in the supply module LEDs (one green "in service" and one red "out of service"). For T1/E1 line cards, tests are designed to verify:

- **Clocking:** Internal (source timing) and receive clock recovery (loop timing).
- **Alarms:** Detection of loss of signal (LOS), loss of frame (LOF), alarm indication signal (AIS), and insertion of AIS.
- **Line Coding:** Alternate mark inversion (AMI) for both T1 and E1, bipolar 8 zero substitution (B8ZS) for T1 only, and high-density bipolar 3 (HDB3) for E1 only.
- **Framing:** Digital signal 1 (DS1) and E1 framing.
- **Channelization:** Ability to transfer user traffic across channels multiplexed from one or more contiguous or non contiguous time slots on a T1 or E1 link.

8.3.3 Logging and Tracing Tests

Logging and tracing tests are designed to verify the configurations and operations of logging and tracing. This also includes verification of "flight data recorder: non-volatile flash memory" logs when the system crashes. Tests may be designed to calculate the impact on system performance when all the logs are enabled.

8.3.4 Element Management Systems Tests

The EMS tests verify the main functions which are to manage, monitor, and upgrade the communication system network elements (NEs). Table 8.1 summarizes the functionalities of an EMS. An EMS communicates with its NEs by using, for example, the Simple Network Management Protocol (SNMP) [2] and a variety of proprietary protocols and mechanisms.

An EMS is a valuable component of a communication systems network. Not all EMSs will perform all of the tasks listed in Table 8.1. An EMS can support a subset of the tasks. A user working through the EMS graphical user interface (GUI) may accomplish some or all of the tasks. Remote access to an EMS allows the operators to access management and control information from any location. This facilitates the deployment of a distributed workforce that can rapidly respond to failure notifications. This means that thin client workstations can operate over the Internet and service provider intranets. In this subgroup, tests are designed to

TABLE 8.1 EMS Functionalities

Fault Management	Configuration Management	Accounting Management	Performance Management	Security Management
Alarm handling	System turn-up	Track service usage	Data collection	Control NE access
Trouble detection	Network provisioning	Bill for services	Report generation	Enable NE functions
Trouble correction	Autodiscovery		Data analysis	Access logs
Test and acceptance	Back-up and restore			
Network recovery	Database handling			

verify the five functionalities of an EMS (Table 8.1). This includes both the EMS client and the EMS server. Examples of EMS tests are given below.

- **Auto-Discovery:** EMS discovery software can be installed on the server to discover elements attached to the EMS through the IP network.
- **Polling and Synchronization:** An EMS server detects a system unreachable condition within a certain time duration. The EMS server synchronizes alarm status, configuration data, and global positioning system (GPS) time from the NE.
- **Audit Operations:** An audit mechanism is triggered whenever an out-of-service network element comes back. The mechanism synchronizes alarms between out-of-service NEs coming back online and the EMS.
- **Fault Manager:** Generation of critical events, such as reboot and reset, are converted to an alert and stored in the EMS database. The EMS can send an email/page to a configured address when an alarm is generated.
- **Performance Manager:** Data are pushed to the EMS server when the data buffer is full in the NE. Data in the server buffer are stored in the backup files once it is full.
- **Security Manager:** It supports authentication and authorization of EMS clients and NEs. The EMS server does the authorization based on user privileges.
- **Policies:** An EMS server supports schedulable log file transfer from the system. The EMS database is periodically backed up to a disk.
- **Logging:** An EMS server supports different logging levels for every major module to debug. An EMS server always logs errors and exceptions.
- **Administration and Maintenance:** This test configures the maximum number of simultaneous EMS clients. The EMS server backs up and restores database periodically.

- **Invoke Clients:** Several clients can be invoked to interact with the EMS server.
- **Live Data View:** A client can provide a live data viewer to monitor performance and show statistics of the system.
- **System Administration Task:** A client can shut down the server, configure logging levels, display server status, and enable auto-discovery.
- **File Transfer:** A client can check on-demand file transfer with a progress bar to indicate the progress and abort an on-demand file transfer operation.

SNMP Example The SNMP is an application layer protocol that facilitates the exchange of management information between network elements. The SNMP is a part of the Internet network management architecture consisting of three components: *network elements*, *agents*, and *network management stations* (NMSs). A NE is a network node that contains an SNMP agent and that resides on a managed network. Network elements collect and store management information and make this information available to the NMS over the SNMP protocol. Network elements can be routers, servers, radio nodes, bridges, hubs, computer hosts, printers, and modems. An agent is a network management software module that (i) resides on a NE, (ii) has the local knowledge of management information, and (iii) translates that information into a form compatible with the SNMP. An NMS, sometimes referred to as a console, executes management applications to monitor and control network elements. One or more NMSs exist on each managed networks. An EMS can act as an NMS.

A management information base (MIB) is an important component of a network management system. The MIB identifies the network elements (or managed objects) that are to be managed. Two types of managed objects exist:

- Scalar objects define a single object instance.
- Tabular objects define multiple related object instances that are grouped in MIB tables.

Essentially, a MIB is a virtual store providing a model of the managed information. For example, a MIB can contain information about the number of packets that have been sent and received across an interface. It contains statistics on the number of connections that exist on a Transmission Control Protocal (TCP) port as well as information that describes each user's ability to access elements of the MIB. The SNMP does not operate on the managed objects directly; instead, the protocol operates on a MIB. In turn, a MIB is the reflection of the managed objects, and its management mechanism is largely proprietary, perhaps through the EMS.

The important aspect of a MIB is that it defines (i) the elements that are managed, (ii) how a user accesses them, and (iii) how they can be reported. A MIB can be depicted as an abstract tree with an unnamed root; individual items are represented as leaves of the tree. An object identifier uniquely identifies a MIB object in the tree. The organization of object identifiers is similar to a telephone number hierarchy; they are organized hierarchically with specific digits assigned by different organizations. The Structure of Management Information (SMI) defines

the rules for describing the management information. The SMI specifies that all managed objects have a name, a syntax, and an encoding mechanism. The name is used as the object identifier. The syntax defines the data type of the object. The SMI syntax uses a subset of the Abstract Syntax Notation One (ASN.1) definitions. The encoding mechanism describes how the information associated with the managed object is formatted as a series of data items for transmission on the network.

Network elements are monitored and controlled using four basic SNMP commands: *read*, *write*, *trap*, and traversal operations. The read command is used by an NMS to monitor NEs. An NMS examines different variables that are maintained by NEs. The write command is used by an NMS to control NEs. With the write command, an NMS changes the values of variables stored within NEs. The trap command is used by NEs to asynchronously report events to an NMS. When certain types of events occur, an NE sends a trap to an NMS. The traversal operations are used by the NMS to determine the variables an NE supports and to sequentially gather information in a variable table, such as a routing table.

The SNMP is a simple request/response protocol that can send multiple requests. Six SNMP operations are defined. The Get operation allows an NMS to retrieve an object instant from an agent. The GetNext operation allows an NMS to retrieve the next object instance from a table or list within an agent. The GetBulk operation allows an NMS to acquire large amounts of related information without repeatedly invoking the GetNext operation. The Set operation allows an NMS to set values for object instances within an agent. The Trap operation is used by an agent to asynchronously inform an NMS of some event. The Inform operation allows one NMS to send trap information to another.

The SNMP messages consist of two parts: a header and a protocol data unit (PDU). The message header contains two fields, namely, a version number and a community name. A PDU has the following fields: PDU type, request ID, error status, error index, and variable bindings. The following descriptions summarize the different fields:

- **Version Number:** The version number specifies the version of the SNMP being used.
- **Community Name:** This defines an access environment for a group of NMSs. NMSs within a community are said to exist within the same administrative domain. Community names serve as a weak form of authentication because devices that do not know the proper community name are precluded from SNMP operations.
- **PDU Type:** This specifies the type of PDU being transmitted.
- **Request ID:** A request ID is associated with the corresponding response.
- **Error Status:** This indicates an error and shows an error type. In GetBulk operations, this field becomes a nonrepeater field by defining the number of requested variables listed that should be retrieved no more than once from the beginning of the request. The field is used when some of the variables are scalar objects with one variable.

- **Error Index:** An error index associates the error with a particular object instance. In GetBulk operations, this field becomes a Max-repetitions field. This field defines the maximum number of times that other variables beyond those specified by the nonrepeater field should be retrieved.
- **Variable Bindings (varbinds):** This comprises the data of an SNMP PDU. Variables bindings associate particular object instances with their current values (with the exception of Get and GetNext requests, for which the value is ignored).

8.3.5 Management Information Base Tests

The MIB tests are designed to verify (i) standard MIBs including MIB II and (ii) enterprise MIBs specific to the system. Tests may include verification of the agent within the system that implements the objects it claims to. Every MIB object is tested with the following primitives: Get, GetNext, GetBulk, and Set. It should be verified that all the counters related to the MIBs are incremented correctly and that the agent is capable of generating (i) well known traps (e.g., coldstart, warmstart, linkdown, linkup) and (ii) application-specific traps (X.25 restart and X.25 reset).

8.3.6 Graphical User Interface Tests

In modern-day software applications, users access functionalities via GUIs. Users of the client-server technology find it convenient to use GUI based applications. The GUI tests are designed to verify the interface to the users of an application. These tests verify different components (objects) such as icons, menu bars, dialogue boxes, scroll bars, list boxes, and radio buttons. Ease of use (usability) of the GUI design and output reports from the viewpoint of actual system users should be checked. Usefulness of the online help, error messages, tutorials, and user manuals are verified. The GUI can be utilized to test the functionality behind the interface, such as accurate response to database queries. GUIs need to be compatible, as discussed in Section 8.5, and consistent across different operating systems, environments, and mouse-driven and keyboard-driven inputs.

Similar to GUI testing, another branch of testing, called *usability testing*, has been evolving over the past several years. The usability characteristics which can be tested include the following:

- **Accessibility:** Can users enter, navigate, and exit with relative ease?
- **Responsiveness:** Can users do what they want and when they want in a way that is clear? It includes ergonomic factors such as color, shape, sound, and font size.
- **Efficiency:** Can users do what they want with a minimum number of steps and time?
- **Comprehensibility:** Do users understand the product structure with a minimum amount of effort?

8.3.7 Security Tests

Security tests are designed to verify that the system meets the security requirements: *confidentiality*, *integrity*, and *availability*. Confidentiality is the requirement that data and the processes be protected from unauthorized disclosure. Integrity is the requirement that data and process be protected from unauthorized modification. Availability is the requirement that data and processes be protected from the denial of service to authorized users. The security requirements testing approach alone demonstrates whether the stated security requirements have been satisfied regardless of whether or not those requirements are adequate. Most software specifications do not include negative and constraint requirements. Security testing should include negative scenarios such as misuse and abuse of the software system. The objective of security testing is to demonstrate [3] the following:

- The software behaves securely and consistently under all conditions—both expected and unexpected.
- If the software fails, the failure does not leave the software, its data, or its resources to attack.
- Obscure areas of code and dormant functions cannot be compromised or exploited.
- Interfaces and interactions among components at the application, framework/middleware, and operating system levels are consistently secure.
- Exception and error handling mechanisms resolve all faults and errors in ways that do not leave the software, its resources, its data, or its environment vulnerable to unauthorized modification or denial-of-service attack.

The popularity of the Internet and wireless data communications technologies have created new types security threats, such as un-authorized access to the wireless data networks, eavesdropping on transmitted data traffic, and denial of service attack [4]. Even within an enterprise, wireless local area network intruders can operate inconspicuously because they do not need a physical connection to the network [5]. Several new techniques are being developed to combat these kinds of security threats. Tests are designed to ensure that these techniques work—and this is a challenging task. Useful types of security tests include the following:

- Verify that only authorized accesses to the system are permitted. This may include authentication of user ID and password and verification of expiry of a password.
- Verify the correctness of both encryption and decryption algorithms for systems where data/messages are encoded.
- Verify that illegal reading of files, to which the perpetrator is not authorized, is not allowed.
- Ensure that virus checkers prevent or curtail entry of viruses into the system.
- Ensure that the system is available to authorized users when a zero-day attack occurs.

- Try to identify any "backdoors" in the system usually left open by the software developers. Buffer overflows are the most commonly found vulnerability in code that can be exploited to compromise the system. Try to break into the system by exploiting the backdoors.
- Verify the different protocols used by authentication servers, such as Remote Authentication Dial-in User Services (RADIUS), Lightweight Directory Access Protocol (LDAP), and NT LAN Manager (NTLM).
- Verify the secure protocols for client–server communications, such as the Secure Sockets Layer (SSL). The SSL provides a secure channel between clients and servers that choose to use the protocol for web sessions. The protocol serves two functions: (i) authenticate the web servers and/or clients and (ii) encrypt the communication channel.
- Verify the IPSec protocol. Unlike the SSL, which provides services at layer 4 and secures the communications between two applications, IPSec works at layer 3 and secures communications happening on the network.
- Verify different wireless security protocols, such as the Extensible Authentication Protocol (EAP), the Transport Layer Security (TLS) Protocol, the Tunneled Transport Layer Security (TTLS) Protocol, and the Protected Extensible Authentication Protocol (PEAP).

8.3.8 Feature Tests

Feature tests are designed to verify any additional functionalities which are defined in the requirement specifications but not covered in the above categories. Examples of those tests are data conversion and cross-functionality tests. Data conversion testing is testing of programs or procedures that are used to convert data from an existing system to a replacement system. An example is testing of a migration tool that converts a Microsoft Access database to MySQL format. Cross-functionality testing provides additional tests of the interdependencies among functions. For example, the verification of the interactions between NEs and an element management system in a 1xEV-DO wireless data network, as illustrated later in Figure 8.5, is considered as cross-functionality testing.

8.4 ROBUSTNESS TESTS

Robustness means how sensitive a system is to erroneous input and changes in its operational environment. Tests in this category are designed to verify how gracefully the system behaves in error situations and in a changed operational environment. The purpose is to deliberately break the system, not as an end in itself, but as a means to find error. It is difficult to test for every combination of different operational states of the system or undesirable behavior of the environment. Hence, a reasonable number of tests are selected from each group illustrated in Figure 8.4 and discussed below.

Figure 8.4 Types of robustness tests.

8.4.1 Boundary Value Tests

Boundary value tests are designed to cover boundary conditions, special values, and system defaults. The tests include providing invalid input data to the system and observing how the system reacts to the invalid input. The system should respond with an error message or initiate an error processing routine. It should be verified that the system handles boundary values (below or above the valid values) for a subset of configurable attributes. Examples of such tests for the SNMP protocol are as follows:

- Verify that an error response wrong_type is generated when the Set primitive is used to provision a variable whose type does not match the type defined in the MIB.
- Verify that an error response wrong_value is generated when the Set primitive is used to configure a varbind list with one of the varbinds set to an invalid value. For example, if the varbind can have values from set 0,1,2,3, then the input value can be − 1 to generate a wrong_value response.
- Verify that an error response too_big is generated when the Set primitive is used to configure a list of 33 varbinds. This is because the Set primitive accepts 32 varbinds at a time.
- Verify that an error response not_writable is generated when the Set primitive is used to configure a variable as defined in the MIB.
- Assuming that the SNMP protocol can support up to 1024 communities, verify that it is not possible to create the 1025th community.

Examples of robustness tests for the 1xEV-DO network, as shown in Figure 8.5, are as follows:

- Assuming that an EMS can support up to 300 NEs, verify that the EMS cannot support the 301st NE. Check the error message from the EMS when the user tries to configure the 301st network element.

Figure 8.5 Typical 1xEV-DO radio access network. (Courtesy of Airvana, Inc.)

- Assuming that an RNC can support 160,000 simultaneous sessions, verify that the 160,001th session cannot be established on an RNC. Check the error message from the EMS when the user tries to establish the 160,001th session.
- Assuming that a base transceiver station (BTS) can support up to 93 simultaneous users, verify that the 94th user cannot be connected to a BTS.

8.4.2 Power Cycling Tests

Power cycling tests are executed to ensure that, when there is a power glitch in a deployment environment, the system can recover from the glitch to be back in normal operation after power is restored. As an example, verify that the boot test is successful every time it is executed during power cycling.

8.4.3 On-Line Insertion and Removal Tests

On-line insertion and removal (OIR) tests are designed to ensure that on-line insertion and removal of modules, incurred during both idle and heavy load operations, are gracefully handled and recovered. The system then returns to normal operation after the failure condition is removed. The primary objective is to ensure that the system recovers from an OIR event without rebooting or crashing any other components. OIR tests are conducted to ensure the fault-free operation of the system while a faulty module is replaced. As an example, while replacing an Ethernet card, the system should not crash.

8.4.4 High-Availability Tests

High-availability tests are designed to verify the redundancy of individual modules, including the software that controls these modules. The goal is to verify that the

system gracefully and quickly recovers from hardware and software failures without adversely impacting the operation of the system. The concept of high availability is also known as fault tolerance. High availability is realized by means of proactive methods to maximize service up-time and to minimize the downtime. One module operates in the active mode while another module is in the standby mode to achieve $1+1$ redundancy. For this mode of operation, tests are designed to verify the following:

- A standby module generates an OIR event, that is, hot swapped, without affecting the normal operation of the system.
- The *recovery time* does not exceed a predefined limit while the system is operational. Recovery time is the time it takes for an operational module to become a standby module and the standby module to become operational.
- A server can automatically switch over from an active mode to a standby mode in case a fail-over event occurs. A fail-over is said to occur when a standby server takes over the workload of an active server.

Tests can be designed to verify that a fail-over does not happen without any *observable failure*. A fail-over without an observable failure is called *silent fail-over*. This can only be observed during the load and stability tests described in Section 8.9. Whenever a silent fail-over occurs, a causal analysis must be conducted to determine its cause.

8.4.5 Degraded Node Tests

Degraded node (also known as failure containment) tests verify the operation of a system after a portion of the system becomes nonoperational. It is a useful test for all mission-critical applications. Examples of degraded node tests are as follows:

- Cut one of the four T1 physical connections from one router to another router and verify that load balancing occurs among the rest of the three T1 physical connections. Confirm that packets are equally distributed among the three operational T1 connections.
- Disable the primary port of a router and verify that the message traffic passes through alternative ports with no discernible interruption of service to end users. Next, reactivate the primary port and verify that the router returns to normal operation.

Example of 1xEV-DO Wireless Data Networks The code division multiple-access (CDMA) 2000 1xEV-DO (one-time evolution, data only) is a standardized technology to deliver high data rate at the air interface between an access terminal (AT) and a base transceiver station (BTS), also known as a radio node (RN) [6–8]. The 1xEV-DO Revision 0 delivers a peak data rate of 2.54 Mbits/s on the forward link (from a BTS to an AT) using only 1.25 MHz of spectrum width and a peak data rate of 153.6 kbits/s on the reverse link. We show an architecture for connecting all the BTS with the Internet (IP core network) in Figure 8.5. In this architecture, a base station controller (BSC), also known as

a radio network controller (RNC), need not be directly connected by dedicated, physical links with a set of BTSs. Instead, the BTSs are connected to the RNCs via an IP back-haul network. Such an interconnection results in flexible control of the BTSs by the RNCs. The RNCs are connected with the Internet (IP core network) via one or more packet data serving nodes (PDSNs). Finally, the EMS allows the operator to manage the 1xEV-DO network.

The ATs (laptop, PDA, mobile telephone) implement the end-user side of the 1xEV-DO and TCP/IP. The ATs communicate with the RNs over the 1xEV-DO airlink. The RNs are the components that terminate the airlink to/from the ATs. The functions performed by the RNs are (i) control and processing of the physical airlink, (ii) processing of the 1xEV-DO media access control (MAC) layer, and (iii) communication via a back-haul network to the RNC. In addition, RNs in conjunction with the RNCs perform the softer handoff mobility function of the 1xEV-DO protocol, where an AT is in communication with multiple sector antennas of the same RN.

An RNC is an entity that terminates the higher layer components of the 1xEV-DO protocol suite. An RNC has logical interfaces to RNs, the authentication, authorization, and accounting (AAA) servers, other RNCs, and the PDSN. An RNC terminates 1xEV-DO signaling interactions from the ATs and processes the user traffic to pass it on to the PDSN. It manages radio resources across all the RNs in its domain and performs mobility management in the form of softer and soft handoffs. The AAA servers are carrier-class computing devices running RADIUS protocol and having an interface to a database. These servers may be configured for two AAA functions, namely, access network AAA and core network AAA. The access network AAA is connected to the RNCs, which perform the terminal authentication function. The core network AAA is connected to the PDSN, which performs user authentication at the IP level. The PDSN is a specialized router implementing IP and mobile IP. The PDSN may be implemented as a single, highly available device or as multiple devices clustered together to form a high-availability device. The PDSN is the edge of the core IP network with respect to the AT, that is, the point where (i) the Point-to-Point Protocol (PPP) traffic of the AT is terminated, (ii) the user is authenticated, and (iii) the IP service options are determined. The core IP network is essentially a network of routers. The EMS server is a system that directly controls the NEs. It is responsible for fault handling, network configuration, and statistics management of the network interfaces. The EMS server interfaces with the NEs via TCP/IP. The EMS server is accessed via a client—and not directly—by the network management staff of a wireless operator. An EMS client is a workstation from which a network operator manages the radio access network.

8.5 INTEROPERABILITY TESTS

In this category, tests are designed to verify the ability of the system to interoperate with third-party products. An interoperability test typically combines different network elements in one test environment to ensure that they work together. In other words, tests are designed to ensure that the software can be connected with other

systems and operated. In many cases, during interoperability tests, users may require the hardware devices to be interchangeable, removable, or reconfigurable. Often, a system will have a set of commands or menus that allow users to make the configuration changes. The reconfiguration activities during interoperability tests are known as *configuration testing* [9]. Another kind of interoperability test is called a (*backward*) *compatibility test*. Compatibility tests verify that the system works the same way across different platforms, operating systems, and database management systems. Backward compatibility tests verify that the current software build flawlessly works with older version of platforms. As an example, let us consider a 1xEV-DO radio access network as shown in Figure 8.5. In this scenario, tests are designed to ensure the interoperability of the RNCs with the following products from different vendors: (i) PDSN, (ii) PDA with 1xEV-DO card, (iii) AAA server, (iv) PC with 1xEV-DO card, (v) laptop with 1xEV-DO card, (vi) routers from different vendors, (vii) BTS or RNC, and (viii) switches.

8.6 PERFORMANCE TESTS

Performance tests are designed to determine the performance of the actual system compared to the expected one. The performance metrics needed to be measured vary from application to application. An example of expected performance is: The response time should be less than 1 millisecond 90% of the time in an application of the "push-to-talk" type. Another example of expected performance is: A transaction in an on-line system requires a response of less than 1 second 90% of the time. One of the goals of router performance testing is to determine the system resource utilization, for maximum aggregation throughput rate considering zero drop packets. In this category, tests are designed to verify response time, execution time, throughput, resource utilization, and traffic rate.

For performance tests, one needs to be clear about the specific data to be captured in order to evaluate performance metrics. For example, if the objective is to evaluate the response time, then one needs to capture (i) end-to-end response time (as seen by external user), (ii) CPU time, (iii) network connection time, (iv) database access time, (v) network connection time, and (vi) waiting time.

Some examples of performance test objectives for an EMS server are as follows:

- Record the CPU and memory usage of the EMS server when 5, 10, 15, 20, and 25 traps per second are generated by the NEs. This test will validate the ability of the EMS server to receive and process those number of traps per second.
- Record the CPU and memory usage of the EMS server when log files of different sizes, say, 100, 150, 200, 250 and 300 kb, are transferred from NEs to the EMS server once every 15 minutes.

Some examples of performance test objectives of SNMP primitives are as follows:

- Calculate the response time of the Get primitive for a single varbind from a standard MIB or an enterprise MIB.

- Calculate the response time of the GetNext primitive for a single varbind from a standard MIB or an enterprise MIB.
- Calculate the response time of the GetBulk primitive for a single varbind from a standard MIB or an enterprise MIB.
- Calculate the response time of the Set primitive for a single varbind from a standard MIB or an enterprise MIB.

Some examples of performance test objectives of a 1xEV-DO Revision 0 are as follows:

- Measure the maximum BTS forward-link throughput.
- Measure the maximum BTS reverse-link throughput.
- Simultaneously generate maximum-rate BTS forward- and reverse-link data capacities.
- Generate the maximum number of permissible session setups per hour.
- Measure the AT-initiated connection setup delay.
- Measure the maximum BTS forward-link throughput per sector carrier for 16 users in the 3-km/h mobility model.
- Measure the maximum BTS forward-link throughput per sector carrier for 16 users in the 30-km/h mobility model.

The results of performance are evaluated for their acceptability. If the performance metric is unsatisfactory, then actions are taken to improve it. The performance improvement can be achieved by rewriting the code, allocating more resources, and redesigning the system.

8.7 SCALABILITY TESTS

All man-made artifacts have engineering limits. For example, a car can move at a certain maximum speed in the best of road conditions, a telephone switch can handle a certain maximum number of calls at any given moment, a router has a certain maximum number of interfaces, and so on. In this group, tests are designed to verify that the system can scale up to its engineering limits. A system may work in a limited-use scenario but may not scale up. The run time of a system may grow exponentially with demand and may eventually fail after a certain limit. The idea is to test the limit of the system, that is, the magnitude of demand that can be placed on the system while continuing to meet latency and throughput requirements. A system which works acceptably at one level of demand may not scale up to another level. Scaling tests are conducted to ensure that the system response time remains the same or increases by a small amount as the number of users are increased. Systems may scale until they reach one or more engineering limits. There are three major causes of these limitations:

i. Data storage limitations—limits on counter field size and allocated buffer space

ii. Network bandwidth limitations—Ethernet speed 10 Mbps and T1 card line rate 1.544 Mbps

iii. Speed limit—CPU speed in megahertz

Extrapolation is often used to predict the limit of scalability. The system is tested on an increasingly larger series of platforms or networks or with an increasingly larger series of workloads. Memory and CPU utilizations are measured and plotted against the size of the network or the size of the load. The trend is extrapolated from the measurable and known to the large-scale operation. As an example, for a database transaction system calculate the system performance, that is, CPU utilization and memory utilization for 100, 200, 400, and 800 transactions per second, then draw graphs of number of transactions against CPU and memory utilization. Extrapolate the measured results to 20,0000 transactions. The drawback in this technique is that the trend line may not be accurate. The system behavior may not degrade gradually and gracefully as the parameters are scaled up. Examples of scalability tests for a 1xEV-DO network are as follows:

- Verify that the EMS server can support the maximum number of NEs, say, 300, without any degradation in EMS performance.
- Verify that the maximum number of BTS, say, 200, can be homed onto one BSC.
- Verify that the maximum number of EV-DO sessions, say, 16,000, can be established on one RNC.
- Verify that the maximum number of EV-DO connections, say, 18,400, can be established on one RNC.
- Verify that the maximum BTS capacity for the three-sector configuration is 93 users per BTS.
- Verify the maximum softer handoff rate with acceptable number of call drops per BTS. Repeat the process every hour for 24 hours.
- Verify the maximum soft handoff rate with no call drops per BTS. Repeat the process for every hour for 24 hours.

8.8 STRESS TESTS

The goal of stress testing is to evaluate and determine the behavior of a software component while the offered load is in excess of its designed capacity. The system is deliberately stressed by pushing it to and beyond its specified limits. Stress tests include deliberate contention for scarce resources and testing for incompatibilities. It ensures that the system can perform acceptably under worst-case conditions under an expected peak load. If the limit is exceeded and the system does fail, then the recovery mechanism should be invoked. Stress tests are targeted to bring out the problems associated with one or more of the following:

- Memory leak
- Buffer allocation and memory carving

One way to design a stress test is to impose the maximum limits on all system performance characteristics at the same time, such as the response time, availability, and throughput thresholds. This literally provides the set of worst-case conditions under which the system is still expected to operate acceptably.

The best way to identify system bottlenecks is to perform stress testing from different locations inside and outside the system. For example, individually test each component of the system, starting with the innermost components that go directly to the core of the system, progressively move outward, and finally test from remote locations far outside the system. Testing each link involves pushing it to its full-load capacity to determine the correct operation. After all the individual components are tested beyond their highest capacity, test the full system by simultaneously testing all links to the system at their highest capacity. The load can be deliberately and incrementally increased until the system eventually does fail; when the system fails, observe the causes and locations of failures. This information will be useful in designing later versions of the system; the usefulness lies in improving the robustness of the system or developing procedures for a disaster recovery plan. Some examples of stress tests of a 1xEV-DO network are as follows:

- Verify that repeated establishment and teardown of maximum telnet sessions to the BSC and BTS executed over 24 hours do not result in (i) leak in the number of buffers or amount of memory or (ii) significant increase in the degree of fragmentation of available memory. Tests should be done for both graceful and abrupt teardowns of the telnet session.
- Stress the two Ethernet interfaces of a BTS by sending Internet traffic for 24 hours and verify that no memory leak or crash occurs.
- Stress the four T1/E1 interfaces of a BTS by sending Internet traffic for 24 hours and verify that no memory leak or crash occurs.
- Verify that repeated establishment and teardown of AT connections through a BSC executed over 24 hours do not result in (i) leaks in the number of buffers or amount of memory and (ii) significant increase in the degree of fragmentation of available memory. The sessions remain established for the duration of the test.
- Verify that repeated soft and softer handoffs executed over 24 hours do not result in leaks in the number of buffers or amount of memory and do not significantly increase the degree of fragmentation of available memory.
- Verify that repeated execution of all CLI commands over 24 hours do not result in leaks in the number of buffers or amount of memory and do not significantly increase the degree of fragmentation of available memory.

Examples of stress tests of an SNMP agent are as follows:

- Verify that repeated walking of the MIBs via an SNMP executed over 24 hours do not result in leaks in number of buffers or amount of memory and do not significantly increase the degree of fragmentation of available memory

- Verify that an SNMP agent can successfully respond to a GetBulk request that generates a large PDU, preferably of the maximum size, which is 8 kbytes under the following CPU utilization: 0, 50, and 90%.
- Verify that an SNMP agent can simultaneously handle multiple GetNext and GetBulk requests over a 24-hour testing period under the following CPU utilizations: 0, 50, and 90%.
- Verify that an SNMP agent can handle multiple Get requests containing a large number of varbinds over a 24-hour testing period under the following CPU utilizations: 0, 50, and 90%.
- Verify that an SNMP agent can handle multiple Set requests containing a large number of varbinds over a 24-hour testing period under the following CPU utilizations: 0, 50, and 90%.

8.9 LOAD AND STABILITY TESTS

Load and stability tests are designed to ensure that the system remains stable for a long period of time under full load. A system might function flawlessly when tested by a few careful testers who exercise it in the intended manner. However, when a large number of users are introduced with incompatible systems and applications that run for months without restarting, a number of problems are likely to occur: (i) the system slows down, (ii) the system encounters functionality problems, (iii) the system silently fails over, and (iv) the system crashes altogether. Load and stability testing typically involves exercising the system with virtual users and measuring the performance to verify whether the system can support the anticipated load. This kind of testing helps one to understand the ways the system will fare in real-life situations. With such an understanding, one can anticipate and even prevent load-related problems. Often, operational profiles are used to guide load and stability testing [10]. The idea is to test the system the way it will be actually used in the field. The concept of *operation profile* is discussed in Chapter 15 on software reliability.

Examples of load and stability test objectives for an EMS server are as follows:

- Verify the EMS server performance during quick polling of the maximum number of nodes, say, 300. Document how long it takes to quick poll the 300 nodes. Monitor the CPU utilization during quick polling and verify that the results are within the acceptable range. The reader is reminded that quick polling is used to check whether or not a node is reachable by doing a ping on the node using the SNMP Get operation.
- Verify the EMS performance during full polling of the maximum number of nodes, say, 300. Document how long it takes to full poll the 300 nodes. Monitor the CPU utilization during full polling and verify that the results are within the acceptable range. Full polling is used to check the status and any configuration changes of the nodes that are managed by the server.

- Verify the EMS server behavior during an SNMP trap storm. Generate four traps per second from each of the 300 nodes. Monitor the CPU utilization during trap handling and verify that the results are within an acceptable range.
- Verify the EMS server's ability to perform software downloads to the maximum number of nodes, say, 300. Monitor CPU utilization during software download and verify that the results are within an acceptable range.
- Verify the EMS server's performance during log file transfers of the maximum number of nodes. Monitor the CPU utilization during log transfer and verify that the results are within an acceptable range.

In load and stability testing, the objective is to ensure that the system can operate on a large scale for several months, whereas, in stress testing, the objective is to break the system by overloading it to observe the locations and causes of failures.

8.10 RELIABILITY TESTS

Reliability tests are designed to measure the ability of the system to remain operational for long periods of time. The reliability of a system is typically expressed in terms of mean time to failure (MTTF). As we test the software and move through the system testing phase, we observe failures and try to remove the defects and continue testing. As this progresses, we record the time durations between successive failures. Let these successive time intervals be denoted by $t_1, t_2, \ldots, t_i$. The average of all the i time intervals is called the MTTF. After a failure is observed, the developers analyze and fix the defects, which consumes some time—let us call this interval the *repair* time. The average of all the repair times is known as the mean time to repair (MTTR). Now we can calculate a value called mean time between failures (MTBF) as $\text{MTBF} = \text{MTTF} + \text{MTTR}$. The random testing technique discussed in Chapter 9 is used for reliability measurement. Software reliability modeling and testing are discussed in Chapter 15 in detail.

8.11 REGRESSION TESTS

In this category, new tests are not designed. Instead, test cases are selected from the existing pool and executed to ensure that nothing is broken in the new version of the software. The main idea in regression testing is to verify that no defect has been introduced into the unchanged portion of a system due to changes made elsewhere in the system. During system testing, many defects are revealed and the code is modified to fix those defects. As a result of modifying the code, one of four different scenarios can occur for each fix [11]:

- The reported defect is fixed.
- The reported defect could not be fixed in spite of making an effort.

- The reported defect has been fixed, but something that used to work before has been failing.
- The reported defect could not be fixed in spite of an effort, and something that used to work before has been failing.

Given the above four possibilities, it appears straightforward to reexecute every test case from version $n - 1$ to version n before testing anything new. Such a full test of a system may be prohibitively expensive. Moreover, new software versions often feature many new functionalities in addition to the defect fixes. Therefore, regression tests would take time away from testing new code. Regression testing is an expensive task; a subset of the test cases is carefully selected from the existing test suite to (i) maximize the likelihood of uncovering new defects and (ii) reduce the cost of testing. Methods for test selection for regression testing are discussed in Chapter 13.

8.12 DOCUMENTATION TESTS

Documentation testing means verifying the technical accuracy and readability of the user manuals, including the tutorials and the on-line help. Documentation testing is performed at three levels as explained in the following:

Read Test: In this test a documentation is reviewed for clarity, organization, flow, and accuracy without executing the documented instructions on the system.

Hands-On Test: The on-line help is exercised and the error messages verified to evaluate their accuracy and usefulness.

Functional Test: The instructions embodied in the documentation are followed to verify that the system works as it has been documented.

The following concrete tests are recommended for documentation testing:

- Read all the documentations to verify (i) correct use of grammar, (ii) consistent use of the terminology, and (iii) appropriate use of graphics where possible.
- Verify that the glossary accompanying the documentation uses a standard, commonly accepted terminology and that the glossary correctly defines the terms.
- Verify that there exists an index for each of the documents and the index block is reasonably rich and complete. Verify that the index section points to the correct pages.
- Verify that there is no internal inconsistency within the documentation.
- Verify that the on-line and printed versions of the documentation are same.
- Verify the installation procedure by executing the steps described in the manual in a real environment.

- Verify the troubleshooting guide by inserting error and then using the guide to troubleshoot the error.
- Verify the software release notes to ensure that these accurately describe (i) the changes in features and functionalities between the current release and the previous ones and (ii) the set of known defects and their impact on the customer.
- Verify the on-line help for its (i) usability, (ii) integrity, (iii) usefulness of the hyperlinks and cross-references to related topics, (iv) effectiveness of table look-up, and (v) accuracy and usefulness of indices.
- Verify the configuration section of the user guide by configuring the system as described in the documentation.
- Finally, use the document while executing the system test cases. Walk through the planned or existing user work activities and procedures using the documentation to ensure that the documentation is consistent with the user work.

8.13 REGULATORY TESTS

In this category, the final system is shipped to the regulatory bodies in those countries where the product is expected to be marketed. The idea is to obtain compliance marks on the product from those bodies. The regulatory approval bodies of various countries have been shown in Table 8.2. Most of these regulatory bodies issue safety and EMC (electromagnetic compatibility)/EMI (electromagnetic interference) compliance certificates (emission and immunity). The regulatory agencies are interested in identifying flaws in software that have potential safety consequences. The safety requirements are primarily based on their own published standards. For example, the CSA (Canadian Standards Association) mark is one of the most recognized, accepted, and trusted symbols in the world. The CSA mark on a product means that the CSA has tested a representative sample of the product and determined that the product meets the CSA's requirements. Safety-conscious and concerned consumers look for the CSA mark on products they buy. Similarly, the CE (Conformité Européenne) mark on a product indicates conformity to the European Union directive with respect to safety, health, environment, and consumer protection. In order for a product to be sold in the United States, the product needs to pass certain regulatory requirements of the Federal Communications Commission (FCC).

Software safety is defined in terms of *hazards*. A hazard is a state of a system or a physical situation which when combined with certain environmental conditions could lead to an accident or mishap. An *accident* or *mishap* is an unintended event or series of events that results in death, injury, illness, damage or loss of property, or harm to the environment [12]. A hazard is a logical precondition to an accident. Whenever a hazard is present, the consequence can be an accident. The existence of a hazard state does not mean that an accident will happen eventually. The concept of safety is concerned with preventing hazards.

TABLE 8.2 Regulatory Approval Bodies of Different Countries

Country	Regulatory Certification Approval Body
Argentina	IRAM is a nonprofit private association and is the national certification body of Argentina for numerous product categories. The IRAM safety mark is rated based on compliance with the safety requirements of a national IRAM standard.
Australia and New Zealand	The Australian Communications Authority (ACA) and the Radio Spectrum Management Group (RSM) of New Zealand have agreed upon a harmonized scheme in producing the C-tick mark that regulates product EMC compliance.
Canada	Canadian Standards Association
Czech Republic	The Czech Republic is the first European country to adopt conformity assessment regulations based on the European Union CE mark without additional approval certification or testing.
European Union	Conformité Européenne
Japan	The VCCI mark (Voluntary Control Council for Interference by Information Technology Equipment) is administered by VCCI for information technology equipment (ITE) sold in Japan.
Korea	All products sold in Korea are required to be compliant and subject to the MIC (Ministry of Information and Communication) mark certification. EMC and safety testing are both requirements.
Mexico	The products must be tested in Mexico for the mandatory NOM (Normality of Mexico) mark.
Peoples Republic of China	The CCC (China Compulsory Certification) mark is required for a wide range of products sold in the Peoples Republic of China.
Poland	The Polish Safety B-mark (B for *bezpieczny*, which means "safe") must be shown on all hazardous domestic and imported products.
	Poland does not accept the CE mark of the European Union.
Russia	GOST-R certification. This certification system is administered by the Russian State Committee on Standardization, Metrology, and Certification (Gosstandart). Gosstandart oversees and develops industry mandatory and voluntary certification programs.
Singapore	The PSB mark is issued by the Singapore Productivity and Standards Board. The Safety Authority (PSB) is the statutory body appointed by the Ministry of Trade and Industry to administer the regulations.
South Africa	The safety scheme for electrical goods is operated by the South African Bureau of Standards (SABS) on behalf of the government. Compliance can be provided by the SABS based on the submission of the test report from any recognized laboratory.
Taiwan	Most products sold in Taiwan must be approved in accordance with the regulations as set forth by the BSMI (Bureau of Standards, Metrology and Inspection).
United States	Federal Communications Commission

A software in isolation cannot do physical damage. However, a software in the context of a system and an embedding environment could be vulnerable. For example, a software module in a database application is not hazardous by itself, but when it is embedded in a missile navigation system, it could be hazardous. If a missile takes a U-turn because of a software error in the navigation system and destroys the submarine that launched it, then it is not a safe software. Therefore, the manufacturers and the regulatory agencies strive to ensure that the software is safe the first time it is released.

The organizations developing safety-critical software systems should have a safety assurance (SA) program to eliminate hazards or reduce their associated risk to an acceptable level [13]. Two basic tasks are performed by an SA engineering team as follows:

- Provide methods for identifying, tracking, evaluating, and eliminating hazards associated with a system.
- Ensure that safety is embedded into the design and implementation in a timely and cost-effective manner such that the risk created by the user/operator error is minimized. As a consequence, the potential damage in the event of a mishap is minimized.

8.14 SUMMARY

In this chapter we presented a taxonomy of system tests with examples from various domains. We explained the following categories of system tests:

- *Basic tests* provide an evidence that the system can be installed, configured, and brought to an operational state. We described five types of basic tests: boot, upgrade/downgrade, light emitting diode, diagnostic, and command line interface tests.
- *Functionality tests* provide comprehensive testing over the full range of the requirements within the capabilities of the system. In this category, we described eight types of tests: communication systems, module, logging and tracing, element management systems, management information base, graphical user interface, security, and feature tests.
- *Robustness tests* determine the system recovery process from various error conditions or failure situations. In this category, we described five types of robustness tests: boundary value, power cycling, on-line insertion and removal, high availability, degraded node tests.
- *Interoperability tests* determine if the system can interoperate with other third-party products.
- *Performance tests* measure the performance characteristics of the system, for example, throughput and response time, under various conditions.
- *Scalability tests* determine the scaling limits of the system.

- *Stress tests* stress the system in order to determine the limitations of the system and determine the manner in which failures occur if the system fails.
- *Load and stability tests* provide evidence that, when the system is loaded with a large number of users to its maximum capacity, the system is stable for a long period of time under heavy traffic.
- *Reliability tests* measure the ability of the system to keep operating over long periods of time.
- *Regression tests* determine that the system remains stable as it cycles through the integration with other projects and through maintenance tasks.
- *Documentation tests* ensure that the system's user guides are accurate and usable.
- *Regulatory tests* ensure that the system meets the requirements of government regulatory bodies. Most of these regulatory bodies issue safety, emissions, and immunity compliance certificates.

LITERATURE REVIEW

A good discussion of software safety concepts, such as *mishap, hazard, hazard analysis, fault tree analysis, event tree analysis, failure modes and effects analysis, and firewall*, can be found in Chapter 5 of the book by Freidman and Voas [13]. The book presents useful examples and opinions concerning the relationship of these concepts to software reliability.

For those readers who are actively involved in usability testing or are interested in a more detailed treatment of the topic, Jeffrey Rubin's book (*Handbook of Usability Testing—How to Plan, Design, and Conduct Effective Tests*, Wiley, New York, 1994) provides an excellent guide. Rubin describes four types of usability tests in great detail: (i) exploratory, (ii) assessment, (iii) validation, and (iv) comparison. The book lists several excellent references on the subject in its bibliography section. Memon et al. have conducted innovative research on test adequacy criteria and automated test data generation algorithms that are specifically tailored for programs with graphical user interfaces. The interested readers are recommended to study the following articles:

A. M. Memon, "GUI Testing: Pitfalls and Process," *IEEE Computer*, Vol. 35, No. 8, 2002, pp. 90–91.

A. M. Memon, M. E. Pollock, and M. L. Soffa, "Hierarchical GUI Test Case Generation Using Automated Planning," *IEEE Transactions on Software Engineering*, Vol. 27, No. 2, 2001, pp. 144–155.

A. M. Memon, M. L. Soffa, and M. E. Pollock, "Coverage Criteria for GUI Testing," in *Proceedings of the 9th ACM SIGSOFT International Symposium on Foundation of Software Engineering*, ACM Press, New York 2001, pp. 256–267.

In the above-mentioned work, a GUI is represented as a series of operators that have preconditions and postconditions related to the state of the GUI. This representation classifies the GUI events into four categories: menu-open events, unrestricted-focused events, restricted-focus events, and system interaction events. Menu-open events are normally associated with the usage of the pull-down menus in a GUI. The unrestricted-focus events simply expand the interaction options available to a GUI user, whereas the restricted-focus events require the attention of the user before additional interactions can occur. Finally, system interaction events require the GUI to interact with the actual application.

An excellent collection of essays on the usability and security aspects of a system appears in the book edited by L. F. Cranor and S. Garfinkel (*Security and Usability*, O'Reilly, Sebastopol, CA, 2005). This book contains 34 groundbreaking articles that discuss case studies of usable secure system design. This book is useful for researchers, students, and practitioners in the fields of security and usability.

Mathematically rigorous treatments of performance analysis and concepts related to software performance engineering may be found in the following books:

R. Jain, *The Art of Computer Systems Performance Analysis*, John, New York, 1991.

C. U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, Reading, MA, 1990.

Each of these books provides a necessary theoretical foundation for our understanding of performance engineering.

REFERENCES

1. D. Rayner. OSI Conformance Testing. *Computer Networks and ISDN Systems*, Vol. **14**, 1987, pp. 79–98.
2. D. K. Udupa. *TMN: Telecommunications Management Network*. McGraw-Hill, New York, 1999.
3. J. Jarzombek and K. M. Goertzel. Security in the Software Cycle. *Crosstalk, Journal of Defense Software Engineering*, September 2006, pp. 4–9.
4. S. Northcutt, L. Zeltser, S. Winters, K. Kent, and R. W. Ritchey. *Inside Network Perimeter Security*, 2nd ed. Sams Publishing, Indianapolis, IN, 2005.
5. K. Sankar, S. Sundaralingam, A. Balinsky, and D. Miller. *Cisco Wireless LAN Security*. Cisco Press, Indianapolis, IN, 2004.
6. IOS for 1 × EV, IS-878, 3Gpp2, http://www.3gpp2.org, June 2001.
7. CDMA2000 IOS Standard, IS-2001, 3Gpp2, http://www.3gpp2.org, Nov. 2001.
8. CDMA2000 High Rate Packet Data Air Interface, IS-856–1, 3Gpp2, http://www.3gpp2.org, Dec. 2001.
9. B. Beizer. *Software Testing and Quality Assurance*. Von Nostrand Reinhold, New York, 1984.
10. A. Avritzer and E. J. Weyuker. The Automatic Generation of Load Test Suites and Assessment of the Resulting Software. *IEEE Transactions on Software Engineering*, September 1995, pp. 705–716.
11. J. A. Whittaker. What Is Software Testing? And Why Is It So Hard? *IEEE Software*, January/February 2000, pp. 70–79.
12. N. G. Leveson. Software Safety: Why, What, and How. *ACM Computing Surveys*, June 1986, pp. 125–163.
13. M. A. Friedman and J. M. Voas. *Software Assessment: Reliability, Safety, Testability*. Wiley, New York, 1995.

Exercises

1. What is an element management system (EMS)? How is it different from a network management station (NMS)?
2. What are the differences between configuration, compatibility, and interoperability testing?
3. What are the differences between performance, stress, and scalability testing? What are the differences between load testing and stress testing?
4. What is the difference between performance and speed?
5. Buffer overflow is the most commonly found vulnerability in network-aware code that can be exploited to compromise a system. Explain the reason.
6. What are zero-day attacks? Discuss its significance with respect to security testing.
7. Discuss the importance of regression testing when developing a new software release. What test cases from the test suite would be more useful in performing a regression test?
8. What are the differences between safety and reliability? What are the differences between safety testing and security testing?
9. What is the similarity between software safety and fault tolerance?
10. For each of the following situations, explain whether it is a hazard or a mishap:
    (a) Water in a swimming pool becomes electrified.
    (b) A room fills with carbon dioxide.
    (c) A car stops abruptly.
    (d) A long-distance telephone company suffers an outage.
    (e) A nuclear weapon is destroyed in an unplanned manner.
11. What are the similarities and differences between quality assurance (QA) and safety assurance (SA)?
12. For your current test project, develop a taxonomy of system tests that you plan to execute against the implementation.

CHAPTER 9

Functional Testing

The test of a first-rate intelligence is the ability to hold two opposed ideas in the mind at the same time, and still retain the ability to function.
— *F. Scott Fitzgerald*

9.1 FUNCTIONAL TESTING CONCEPTS OF HOWDEN

William E. Howden developed the idea of functional testing of programs while visiting the International Mathematics and Statistics Libraries (IMSL) in Houston in 1977–1978. IMSL is presently known as Visual Numerics (http://www.vni.com/). The IMSL libraries are a comprehensive set of mathematical and statistical functions that programmers can embed into their software applications. IMSL uses proven technology that has been thoroughly tested, well documented, and continuously maintained. Howden applied the idea of functional testing to programs from edition 5 of the IMSL package. The errors he discovered can be considered to be of some subtlety to have survived to edition 5 status [1].

A *function* in mathematics is defined to be a set of ordered pairs (X_i, Y_i), where X_i is a vector of input values and Y_i is a vector of output values. In functional testing, a program P is viewed as a *function* that transforms the input vector X_i into an output vector Y_i such that $Y_i = P(X_i)$.

Examples

1. Let $Y_1 = \sqrt{X_1}$. Here, P is a square-root computing function which calculates the squareroot Y_1 of nonnegative integer X_1. The result is assigned to Y_1.
2. Let $Y_2 = \text{C_compiler}(X_2)$. The program P is viewed as a C_compiler function that produces object code from C program X_2. The object code is held in Y_2.

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

3. Let $Y_3 = \text{TelephoneSwitch}(X_3)$. A telephone switch program P produces a variety of tones and voice signals represented by the vector

$$Y_3 = \{\text{idle, dial, ring, fast busy, slow busy tone, voice}\}$$

by processing input data represented by the vector

$$X_3 = \{\text{off hook, on hook, phone number, voice}\}.$$

4. Let $Y_4 = \text{sort}(X_4)$. The program P in this example is an implementation of a sorting algorithm which produces a sorted array Y_4 from the input vector $X_4 = \{A, N\}$, where A is the array to be sorted and N is the number of elements in A.

The above four examples suggest that sometimes it is easy to view a program as a function in the mathematical sense and sometimes it is more difficult. It is easier to view a program as a function when the input values are algorithmically, or mathematically, transformed into output values, such as in the first and the fourth examples above. In the fourth example, Y_4 is a certain permutation of the input array A. It is more difficult to view a program as a function when the input values are not directly transformed into the output values. For instance, in the third example above, an *off-hook* input is not mathematically transformed into a *dial tone* output. In functional testing we are not concerned with the details of the mechanism by which an input vector is transformed into an output vector. Instead, a program is treated as a function in the general sense.

Three key elements of a function are its input, output, and expected transformation of input to output. Ignoring the details of the actual transformation of input to output, we analyze the domains of the input and the output variables of programs to generate test data. The four key concepts in functional testing [2] are as follows:

- Precisely identify the domain of each input and each output variable.
- Select values from the data domain of each variable having *important* properties.
- Consider combinations of special values from different input domains to design test cases.
- Consider input values such that the program under test produces special values from the domains of the output variables.

One can identify the domain of an input or an output variable by analyzing the requirements specification and the design documents. In the following sections, we discuss Howden's method for selecting test data from the domains of input and output variables.

9.1.1 Different Types of Variables

In this section, we consider numeric variables, arrays, substructures, and subroutine arguments and their *important* values. These types of variables are commonly used as input to and output from a large number of systems for numeric calculations. The MATLAB package and a number of tax filing software systems are examples of such systems.

Numeric Variable The domain of a numeric variable is specified in one of two ways as follows:

- **A set of discrete values:** An example of this type of domain is MODE = $\{23, 79\}$ from the Bluetooth specification. The variable MODE is a numeric variable which takes one of the two values from the set $\{23, 79\}$. The MODE value is used in a modulo operation to determine the channel frequency for packet transmission.
- **A few contiguous segments of values:** As an example, the gross income input to a tax filing software for a person or company is specified as a value from the range $\{0, \ldots, \infty\}$. Each contiguous segment is characterized by a *minimum (MIN)* value and a *maximum (MAX)* value.

Example: The inputs and the output variables of the frequency selection box (FSB) module of the Bluetooth wireless communication system are shown in Figure 9.1. Bluetooth communication technology uses a frequency hopping spread-spectrum technique for accessing the wireless medium. A piconet *channel* is viewed as a possibly infinite sequence of *slots*, where one slot is 625 μs long. The frequency on which a data packet will be transmitted during a given slot is computed by the FSB module illustrated in Figure 9.1. The FSB module accepts three input variables MODE, CLOCK, and ADDRESS and generates values of the output variable INDEX. All four are numeric variables. The domains of these variables are characterized as follows:

MODE: The domain of variable MODE is the discrete set $\{23, 79\}$.

CLOCK: The CLOCK variable is represented by a 28-bit unsigned number with MIN = 0x0000000 and MAX = 0xFFFFFFF. The smallest increment

Figure 9.1 Frequency selection box of Bluetooth specification.

in CLOCK represents the elapse of 312.5 μs. The FSB module uses the *upper* 27 bits of the 28-bit CLOCK in frequency calculations.

ADDRESS: The ADDRESS variable is represented by a 48-bit unsigned number. The FSB module uses the *lower* 28 bits of the 48-bit ADDRESS. Therefore, the range of ADDRESS from the viewpoint of the FSB module is specified as follows:

- MIN = 0xyyyyy0000000, where yyyyy is a 20-bit arbitrary value.
- MAX = 0xzzzzzFFFFFFF, where zzzzz is a 20-bit arbitrary value.

INDEX: This variable assumes values in a given range as specified in the following:

- MIN = 0.
- MAX = 22 if MODE = 23.
- MAX = 78 if MODE = 79.

Having characterized the domain of a numeric variable as a discrete set of values or as a set of contiguous segments, test data are chosen by applying different *selection criteria* depending upon the use of those variables, namely, (i) input, (ii) output, (iii) dualuse, and (iv) multipletype. A dual-use variable is one that holds an input to the system under test at the beginning of system execution and receives an output value from the system at some point thereafter. A multiple-type variable is one that can hold input (or even output) values of different types, such as numeric and string, at different times. In this section we give an example of such a variable from a real-life system. Next, we explain the four selection criteria.

1. *Selection Criteria for Input Variables*: If the input domain is a discrete set of values, then tests involving each value are performed. The domain of the input variable MODE in Figure 9.1 consists of the set $\{23, 79\}$. The FSB module is tested at least once with MODE = 23 and at least once with MODE = 79. If the domain of a variable consists of one or more segments of values, then test data are selected as follows:
   - Consider the minimum value of a segment.
   - Consider the maximum value of a segment.
   - Consider a typical representative value in a segment.
   - Consider certain values which have special mathematical properties. These values include 0, 1, and real numbers with *small* absolute values.
   - Consider, if possible, values lying outside a segment. Here the idea is to observe the behavior of the program in response to invalid input.
   - In case the conceptual minimum (maximum) of a variable is $-\infty$ ($+\infty$), then a large negative (positive) value is chosen to represent $-\infty$ ($+\infty$).
2. *Selection Criteria for Output Variables*: If the domain of an output variable consists of a *small* set of discrete values, then the program is tested with input which results in the generation of each of the output values. The output variable of the frequency hopping box in Figure 9.1 has a

domain of two discrete sets $\{0, \ldots, 22\}$ and $\{0, \ldots, 78\}$ for MODE = 23 and MODE = 79, respectively. The frequency hopping box must be adequately tested so that it produces each of the output values as desired. For a *large* set of discrete values, the program is tested with many different inputs that cause the program to produce many different output values. If an output variable has a domain consisting of one or more segments of numbers, then the program is tested as follows:

- Test the program with different inputs so that it produces the *minimum* values of the segments.
- Test the program with different inputs so that it produces the *maximum* values of the segments.
- Test the program with different inputs so that it produces some *interior* values in each segment.

3. *Selection Criteria for Dual-Use Variables*: A variable often serves as an input to a program (or function) and holds the output from the program (or function) at the end of the desired computation. Such a variable is called a *dual-use* variable. Additional test cases are designed to meet the following selection criteria for dual-use variables:

- Consider a test case such that the program produces an output value which is *different* from the input value of the same dual-use variable. The idea behind this test is to avoid coincidental correctness of the output.
- Consider a test case such that the program produces an output value which is *identical* to the input value of the same dual-use variable.

4. *Selection Criteria for Multiple-Type Variables*: Sometimes an input variable can take on values of different types. For example, a variable may take on values of type *integer* in one program invocation and of type *string* in another invocation. It may be unlikely for a programmer to define a single storage space to hold values of different types. The program will read the input value into different locations (i.e., variables) depending upon the type of value provided by the user. This scenario requires us to test that the program correctly reads an input value depending upon its type and, subsequently, correctly processes the value. Such multiple-type variables may arise in real-life systems, and programs must take necessary actions to handle them. If an input or output variable can take on values of different types, then the following criteria are used:

- For an input variable of multipletype, the program is tested with input values of all the types.
- For an output variable of multipletype, the program is tested with different inputs so that the variable holds values of all different types.

Example: We show a part of the tax forms prepared by Canada Customs and Revenue Agency (CCRA) in Figure 9.2. By analyzing this specification we conclude that a taxpayer (user) inputs a *real number* or a *blank* in *line 2*. The

Enter your net income from line 236 of your return			1
Enter your spouse or common-law partner's net income from page 1 of your return		+	2
Add lines 1 and 2	**Income for Ontario credits**	=	3

Involuntary separation 6089

If, on December 31, 2001, you and your spouse or common-law partner occupied separate principal residences for medical, educational, or business reasons, **leave line 2 blank** and enter his or her address in the area beside box 6089.

Figure 9.2 Part of form ON479 of T1 general—2001, published by the CCRA.

value input in line 2 is a real number representing the net income of the spouse or common-law partner of the user if both of them occupied the same residence on December 31, 2001. Otherwise, if they occupied separate principal residences for the specified reasons, then line 2 must be left blank and the address of the spouse or common-law partner must be provided in box 6089.

Clearly, line 2 is an input to a tax filing software system which must be able to handle different *types* of values input in *line 2* —namely *real* values and *blank*, and a blank is not the same as 0.0. This is because, if we equate a blank with 0.0, then the software system may not know when and how to interpret the information given in box 6089. Referring to line 2 input of Figure 9.2, a tax filing program must be tested as follows:

1. Interpret line 2 as a *numeric* variable taking on values from an interval and apply selection criteria such as selecting the minimum value, the maximum value, and an interior value of the defined interval.
2. Interpret line 2 as a *string* variable taking on values from a discrete set of values, where the discrete set consists of just one member, namely a blank.

Arrays An array holds values of the same type, such as integer and real. Individual elements of an array are accessed by using one or more indices. In some programming languages, such as MATLAB, an array can hold values of both integer and real types. An array has a more complex structure than an individual numeric variable. This is because of the following three distinct properties of an array. The three properties are individually and collectively considered while testing a program.

- An array can have one or more dimensions. For example, $A[i][j]$ is the element at row i and column j of a two-dimensional array A. Array dimensions are considered in testing because their values are likely to be used in controlling *for* and *while* loops. Just as we select extremal—both minimum and maximum—values and an intermediate value of a numeric variable, we need to consider arrays of different configurations, such as an array of

minimum size, an array of maximum size, an array with a minimum value for the first dimension and a maximum value for the second dimension, and so on.

- Individual array elements are considered as distinct numeric variables. Each value of an array element can be characterized by its minimum value, maximum value, and special values, such as 0, 1, and ϵ. All these values need to appear in tests in order to observe the program behavior while processing the extremal and special values.
- A portion of an array can be collectively interpreted as a distinct substructure with specific application-dependent properties. For example, in the field of numerical analysis a matrix structure is a common representation of a set of linear equations. The diagonal elements of a matrix, the lower triangular matrix, and the upper triangular matrix are substructures with significance in numerical analysis.

  Just as we consider special values of a numeric variable, such as 0, 1, and a small value ϵ, there exist special values of substructures of an array. For example, some well-known substructures of a two-dimensional array are individual rows and columns, diagonal elements, a lower triangular matrix, and an upper triangular matrix. These substructures are interpreted as a whole.

The selection criteria for array dimensions is based on the following three intuitive steps:

1. Completely specify the dimensions of an array variable. In programming languages such as C and Java, the dimensions of an array are statically defined. On the other hand, in the programming language MATLAB, there is no such concept as static definition of an array dimension. Instead, an array can be dynamically built without any predefined limit.
2. Construct different, special configurations of the array by considering special values of individual array dimensions and their combinations. Consider an array with k dimensions, where each dimension is characterized by a minimum value, a maximum value, and an intermediate value. These selections can be combined to form 3^k different sets of dimensions for a k-dimensional array.

   Let us take a concrete example with a two-dimensional array, where $k = 2$. The two dimensions are commonly known as row and column dimensions. The minimum number of rows of an array for an application can be an arbitrary value, such as 1. Similarly, the maximum number of rows can be 20. An intermediate row number is 10. The minimum number of columns of the array can be 2, the maximum number of columns can be 15, and an intermediate column number can be 8. By considering the three row values and three column values, one can enumerate the $3^k = 3^2 = 9$ combinations of different array configurations. Some examples of those 9 different configurations of arrays are 1×2, 1×15, 20×2, 20×15, and so on.

3. Apply the selection criteria of Section 9.1.1 to individual elements of the selected array configurations.

Substructure In general, a structure means a data type that can hold multiple data elements. In the field of numerical analysis, a matrix structure is commonly used. For example, a set of n linear equations in n unknown quantities is represented by an $n \times (n+1)$ matrix. The n rows and the first n columns of the matrix represent the coefficients of the n linear equations, whereas the $(n+1)$th column represents the constants of the n equations. The n rows and the first n columns can be considered as one substructure, whereas the $(n+1)$th column is another substructure. In addition, individual columns, rows, and elements of the matrix can be considered distinct substructures. In general, one can identify a variety of substructures of a given structure. Given the fact that there can be a large number of substructures of a structure, it is useful to find *functionally identifiable* substructures. In the above example of an $n \times (n+1)$ matrix, the $(n+1)$th column is one functionally identifiable substructure, whereas the rest of the matrix is another functionally identifiable substructure. Individual columns and individual rows are functionally identifiable substructures as well. For example, a row of identical values means that the coefficients of all the variables and the constant of an equation have identical values. Sometimes, pairs of columns can form substructures. For example, one may use a pair of columns to represent complex numbers—one column for the "real" part and another column for the "imaginary" part. It is useful to consider both the full structure and the substructures, such as rows, columns, and individual elements, in functional testing involving matrix structures. The following criteria are considered in identifying substructures:

- **Dimensions of Substructures:** Choose structure dimensions such that substructures can take on all possible dimension values. Examples of substructures taking on special values are as follows:(i) the number of elements in a row is 1 and (ii) the number of elements in a row is "large."
- **Values of Elements of Substructures:** Choose values of elements of structures such that substructures take on all possible special values. Examples of substructures taking on special values are as follows: (i) all elements of a row take on value 0, (ii) all elements take on value 1, and (iii) all elements take on value ϵ.
- **Combinations of Individual Elements with Array Dimensions:** Combine the *dimension* aspect with the *value* aspect. For example, one can select a *large* vector—a row or a column—of elements with identical special values, such as all 0's, all 1's, and so on.

Subroutine Arguments Some programs accept input variables whose values are the *names* of functions. Such programs are found in numerical analysis and statistical applications [1]. Functional testing requires that each value of such a variable be included in a test case. Consider a program P calling a function $f(g, \text{param_list})$, where $f()$ accepts a list of parameters denoted by param_list and another parameter

g of enumerated type. When $f()$ is invoked, it invokes the function held in g. Let the values of g be represented by the set $\{g_1, g_2, g_3\}$. We design three test cases to execute $f(g_1, \text{list}_1)$, $f(g_2, \text{list}_2)$, and $f(g_3, \text{list}_3)$ such that, eventually, $g_1()$, $g_2()$, and $g_3()$ are executed.

9.1.2 Test Vector

A *test vector*, also called test data, is an instance of the input to a program. It is a certain configuration of the values of all the input variables. Values of individual input variables chosen in the preceding sections must be combined to obtain a test vector. If a program has n input variables $\text{var}_1, \text{var}_2, \ldots, \text{var}_n$ which can take on $k_1, k_2, \ldots, k_n$ special values, respectively, then there are $k_1 \times k_2 \times \cdots \times k_n$ possible combinations of test data.

Example: We show the number of special values of different input variables of the FSB module of Figure 9.1 in Table 9.1. Variable MODE takes on values from a discrete set of size 2 and both the values from the discrete set are considered. Variables CLOCK and MODE take on values from one interval each and *three* special values for each of them are considered. Therefore, one can generate $2 \times 3 \times 3 = 18$ test vectors from Table 9.1.

Some programs accept a large number of input variables. Tax filing software systems are examples of such programs. Consider all possible combinations of a few special values of a large number of input variables is a challenging task. If a program has n input variables, each of which can take on k special values, then there are k^n possible combinations of test vectors. We know that k is a *small* number, but n may be large. We have more than one million test vectors even for $k = 3$ and $n = 20$. Therefore, there is a need to identify a method for reducing the number of test vectors obtained by considering all possible combinations of all the sets of special values of input variables.

Howden suggested that there is no need of combining values of *all* input variables to design a test vector if the variables are not *functionally related*. To reduce the number of input combinations, Howden suggested [3, 4] that we produce all possible combinations of special values of variables falling in the same functionally related subset. In this way, the total number of combinations of special values of the input variables is reduced. It is difficult to give a formal definition

TABLE 9.1 Number of Special Values of Inputs to FBS Module of Figure 9.1

Variable	Number of Special Values (k)	Special Values
MODE	2	{23, 79}
CLOCK	3	{0x0000000, 0x000FF00, 0xFFFFFFF}
ADDRESS	3	{0xFFFFF0000000, 0xFFFFF00FFF00, 0xFFFFFFFFFFFF}

Figure 9.3 Functionally related variables.

of the idea of functionally related variables, but it is easy to identify them. Let us consider the following examples:

- Variables appearing in the same assignment statement are functionally related.
- Variables appearing in the same branch predicate—the condition part of an if statement, for example—are functionally related.

Example: The program P in Figure 9.3*a* has *five* input variables such that $x_1, \ldots, x_4$ take on *three* special values and x_5 is a Boolean variable. The total number of combinations of the special values of the five input variables is $3^4 \times 2 = 162$. Let us assume that program P has an internal structure as shown in Figure 9.3*b*, where variables x_1 and x_2 are functionally related and variables x_3 and x_4 are functionally related. Function f_1 uses the input variables x_1 and x_2. Similarly, function f_2 uses the input variables x_3 and x_4. Input variable x_5 is used to decide whether the output of f_1 or the output of f_2 will be the output of P. We consider $3^2 = 9$ different combination of x_1 and x_2 as input to f_1, 9 different combinations of x_3 and x_4 as input to f_2, and two different values of x_5 to the decision box d in P. We need 36 $[(9 + 9) \times 2]$ combinations of the five input variables $x_1, \ldots, x_5$, which is much smaller than 162.

9.1.3 Testing a Function in Context

Let us consider a program P and a function f in P as shown in Figure 9.4. The variable x is an input to P and input to f as well. Suppose that x can take on values in the range $[-\infty, +\infty]$ and that f is called only when the predicate $x \geq 20$

Figure 9.4 Function in context.

holds. If we are unaware of the predicate $x \geq 20$, then we are likely to select the following set of test data to test P:

$$x = +k,$$
$$x = -k,$$
$$x = 0.$$

Here, k is a number with a large magnitude.

The reader may note that the function f will be invoked just once for $x = +k$, assuming that $k \geq 20$, and it will not be invoked when P is run with the other two test data because of the conditional execution of f. Testing function f in isolation will require us to generate the same test data as above. It may be noted that the latter two data points are invalid data because they fall outside the range of x for f in P. The valid range of x for f is $[20, +\infty]$, and functional testing *in context* requires us to select the following values of x:

$$x = k \quad \text{where } k \gg 20$$
$$x = y \quad \text{where } 20 < y \ll k$$
$$x = 20$$

where the symbols $\ll$ and $\gg$ are read as "much larger" and "much smaller," respectively.

9.2 COMPLEXITY OF APPLYING FUNCTIONAL TESTING

In order to have an idea of the difficulty in applying the concept of functional testing, let us summarize the main points in functional testing in the following:

- *Identify* the input and the output variables of the program and their data domains.
- *Compute* the expected outcomes as illustrated in Figure 9.5*a* for selected input values.
- *Determine* the input values that will cause the program to produce selected outputs as illustrated in Figure 9.5*b*.

Figure 9.5 (*a*) Obtaining output values from an input vector and (*b*) obtaining an input vector from an output value in functional testing.

Generating test data by analyzing the *input domains* has the following two characteristics.

- The number of test cases obtained from an analysis of the input domains is likely to be *too many* because of the need to design test vectors representing different combinations of special values of the input variables.
- Generation of the expected output for a certain test vector is relatively simple. This is because a test designer computes an expected output from an understanding and analysis of the specification of the system.

On the other hand, generating test data by analyzing the *output domains* has the following characteristics:

- The number of test cases obtained from an analysis of the output domains is likely to be *fewer* compared to the same number of input variables because there is no need to consider different combinations of special values of the output variables.
- Generating an input vector required to produce a chosen output value will require us to analyze the specification in the *reverse* direction, as illustrated in Figure 9.5*b*. Such reverse analysis will be a more challenging task than

computing an expected value in the forward direction, as illustrated in Figure 9.5*a*.

So far in this section we have discussed the ways to apply the idea of functional testing to an entire program. However, the underlying concept, that is, analyzing the input and output domains of a program, can as well be applied to individual modules, functions, or even lines of code. This is because every computing element can be described in terms of its input and output domains, and hence the idea of functional testing can be applied to such a computing element.

Referring to Figure 9.6, program P is decomposed into three *modules* M_1, M_2, and M_3. In addition, M_1 is composed of functions f_1 and f_5, M_2 is composed of functions f_2 and f_3, and M_3 is composed of functions f_4 and f_6. We can apply the idea of functional testing to the entire program P, individual modules M_1, M_2, and M_3, and individual functions $f_1, \ldots, f_6$ by considering their respective input and output domains as listed in Table 9.2.

Figure 9.6 Functional testing in general.

TABLE 9.2 Input and Output Domains of Functions of *P* in Figure 9.6

Entity Name	Input Variables	Output Variables
P	$\{x_1, x_2, x_3, x_4\}$	$\{z\}$
M_1	$\{x_1, x_2\}$	$\{y_5\}$
M_2	$\{x_3, x_4\}$	$\{y_4\}$
M_3	$\{y_4, y_5\}$	$\{z\}$
f_1	$\{x_1, x_2\}$	$\{y_1\}$
f_2	$\{x_3, x_4, y_3\}$	$\{y_2, y_4\}$
f_3	$\{y_2\}$	$\{y_3\}$
f_4	$\{y_4\}$	$\{y_6\}$
f_5	$\{y_1\}$	$\{y_5\}$
f_6	$\{y_5, y_6\}$	$\{z\}$

Conceptually, one can apply functional testing at any level of abstraction, from a single line of code at the lowest level to the entire program at the highest level. As we consider individual modules, functions, and lines of code, the task of accurately identifying the input and output domains of the computing element under test becomes more difficult.

The methodology for developing functional test cases is an analytical process that decomposes specification of a program into different classes of behaviors. The functional test cases are designed for each class separately by identifying the input and output domains of the class. Identification of input and output domains help in classifying the specification into different classes. However, often, in practice, the total number of input and output combinations can be very large. Several well-known techniques are available to tackle this issue, which we discuss in the following sections.

9.3 PAIRWISE TESTING

Pairwise testing is a special case of all-combination testing of a system with n input variables. Let us consider n input variables denoted by $\{v_1, v_2, \ldots, v_i, \ldots, v_n\}$. For simplicity, assume that for each variable v_i, $1 \leq i \leq n$, we choose k values of interest. An all-combination testing means considering all the k^n different test vectors. On the other hand, pairwise testing means that each possible combination of values for every pair of input variables is covered by at least one test case. Pairwise testing requires a subset of test cases (vectors) covering all pairwise combinations of values instead of all possible combinations of values of a set of variables.

Pairwise testing is also referred to as all-pair/two-way testing. One can also generate three- four-way or 4-way tests to cover all combinations of values of three or four variables. As all combinations of values of more and more variables (e.g., 2, 3, 4, ...) are considered, the size of the test suite grows rapidly. Empirical results [5] for medical devices and distributed database systems show that two-way testing would detect more that 90% of the defects, whereas, four-way testing would detect 100% of the defects. Pairwise testing would detect approximately 70% of the defects, whereas six-way testing would detect 100% of defects for browser and server applications.

Consider the system S in Figure 9.7, which has three input variables X, Y, and Z. Let the notation $D(w)$ denote the set of values for an arbitrary variable w. For the three given variables X, Y, and Z, their value sets are as follows: $D(X) = \{\text{True, False}\}$, $D(Y) = \{0, 5\}$, and $D(Z) = \{Q, R\}$. The total number of

Figure 9.7 System S with three input variables.

TABLE 9.3 Pairwise Test Cases for System *S*

Test Case ID	Input X	Input Y	Input Z
TC_1	True	0	Q
TC_2	True	5	R
TC_3	False	0	Q
TC_4	False	5	R

all-combination test cases is $2 \times 2 \times 2 = 8$. However, a subset of four test cases, as shown in Table 9.3, covers all pairwise combinations. Different test generation strategies for pairwise testing have been reported in the literature [6]. In this section, two popular techniques, namely, orthogonal array (OA) [7] and in parameter order (IPO) [8], are discussed.

9.3.1 Orthogonal Array

Orthogonal arrays were originally studied as a kind of numerical curiosity by monks [9]. It was further studied by C. R. Rao, a statistician, in the late 1940s, [10]. Genichi Taguchi first used the idea of orthogonal arrays in his experimental design of total quality management (TQM) [11]. The method, known as the Taguchi method, has been used in experimental design in the manufacturing field and provides an efficient and systematic way to optimize designs for performance, quality, and cost. It has been used successfully in Japan and the United States in designing reliable, high-quality products at low cost in the automobile and consumer electronics industries. Mandl was the first to use the concept of orthogonal array in designing test cases for pairwise testing of compilers [12].

Let us consider the two-dimensional array of integers shown in Table 9.4. The array has an interesting property: Choose any two columns at random and find all pairs (1,1), (1,2), (2,1), and (2,2); however, not all the combinations of 1's and 2's appear in the table. For example, (2,2,2) is a valid combination, but it is not in the table. Only four of the eight combinations can be found in the table. This is an example of $L_4(2^3)$ orthogonal array. The 4 indicates that the array has four rows, also known as *runs*. The 2^3 part indicates that the array has three columns, known

TABLE 9.4 $L_4(2^3)$ Orthogonal Array

	Factors		
Runs	1	2	3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

TABLE 9.5 Commonly Used Orthogonal Arrays

Orthogonal Array	Number of Runs	Maximum Number of Factors	Maximum Number of Columns at These Levels			
			2	3	4	5
L_4	4	3	3			
L_8	8	7	7			
L_9	9	4	—	4		
L_{12}	12	11	11			
L_{16}	16	15	15			
L'_{16}	16	5	—	—	5	
L_{18}	18	8	1	7		
L_{25}	25	6	—	—	—	6
L_{27}	27	13	—	13		
L_{32}	32	31	31			
L'_{32}	32	10	1	—	9	
L_{36}	36	23	11	12		
L'_{36}	36	16	3	13		
L_{50}	50	12	1	—	—	11
L_{54}	54	26	1	25		
L_{64}	64	63	63			
L'_{64}	64	21	—	—	21	
L_{81}	81	40	—	40		

as factors, and each cell in the array contains two different values, known as levels. *Levels* mean the maximum number of values that a single factor can take on. The maximum number of columns at levels 1 and 2 are 0 and 3, respectively. Orthogonal arrays are generally denoted by the pattern L_{Runs} (Levels^{Factors}). Commonly used orthogonal arrays are given in Table 9.5.

Let us consider our previous example of the system S, where S has three input variables X, Y, and Z. For the three given variables X, Y, and Z, their value sets are as follows: $D(X) = \{\text{True, False}\}$, $D(Y) = \{0, 5\}$, and $D(Z) = \{Q, R\}$. Let us map the variables to the factors and values to the levels onto the $L_4(2^3)$ orthogonal array (Table 9.4) with the resultant in Table 9.3. In the first column, let $1 = \text{True}, 2 = \text{False}$. In the second column, let $1 = 0, 2 = 5$. In the third column, let $1 = Q, 2 = R$. Note that not all combinations of all variables have been selected; instead combinations of all pairs of input variables have been covered with four test cases. It is clear from the above example that orthogonal arrays provide a technique for selecting a subset of test cases with the following properties:

- The technique guarantees testing the pairwise combinations of all the selected variables.
- The technique generates fewer test cases than the all-combination approach.

- The technique generates a test suite that has an even distribution of all pairwise combinations.
- The technique can be automated.

In the following, the steps of a technique to generate orthogonal arrays are presented. The steps are further explained by means of a detailed example.

Step 1: Identify the maximum number of independent input variables with which a system will be tested. This will map to the *factors* of the array—each input variable maps to a different factor.

Step 2: Identify the maximum number of values that each independent variable will take. This will map to the levels of the array.

Step 3: Find a suitable orthogonal array with the smallest number of runs $L_{\text{Runs}}(X^Y)$, where X is the number of levels and Y is the number of factors [7, 13]. A suitable array is one that has at least as many factors as needed from step 1 and has at least as many levels for each of those factors as identified in step 2.

Step 4: Map the variables to the factors and values of each variable to the levels on the array.

Step 5: Check for any "left-over" levels in the array that have not been mapped. Choose arbitrary valid values for those left-over levels.

Step 6: Transcribe the runs into test cases.

Web Example. Consider a website that is viewed on a number of browsers with various plug-ins and operating systems (OSs) and through different connections as shown in Table 9.6. The table shows the variables and their values that are used as elements of the orthogonal array. We need to test the system with different combinations of the input values.

Following the steps laid out previously, let us design an orthogonal array to create a set of test cases for pairwise testing:

Step 1: There are four independent variables, namely, Browser, Plug-in, OS, and Connection.

Step 2: Each variable can take at most three values.

TABLE 9.6 Various Values That Need to Be Tested in Combinations

Variables	Values
Browser	Netscape, Internet Explorer (IE), Mozilla
Plug-in	Realplayer, Mediaplayer
OS	Windows, Linux, Macintosh
Connection	LAN, PPP, ISDN

Note: LAN, local-area network; PPP, Point-to-Point Protocal; ISDN, Integrated Services Digital Network.

TABLE 9.7 $L_9(3^4)$ Orthogonal Array

	Factors			
Runs	1	2	3	4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Step 3: An orthogonal array $L_9(3^4)$ as shown in Table 9.7 is good enough for the purpose. The array has nine rows, three levels for the values, and four factors for the variables.

Step 4: Map the variables to the factors and values to the levels of the array: the factor 1 to Browser, the factor 2 to Plug-in, the factor 3 to OS, and the factor 4 to Connection. Let 1 = Netscape, 2 = IE, and 3 = Mozilla in the Browser column. In the Plug-in column, let 1 = Realplayer and 3 = Mediaplayer. Let 1 = Windows, 2 = Linux, and 3 = Macintosh in the OS column. Let 1 = LAN, 2 = PPP, and 3 = ISDN in the Connection column. The mapping of the variables and the values onto the orthogonal array is given in Table 9.8.

Step 5: There are left-over levels in the array that are not being mapped. The factor 2 has three levels specified in the original array, but there are only two possible values for this variable. This has caused a level (2) to be left

TABLE 9.8 $L_9(3^4)$ Orthogonal Array after Mapping Factors

Test Case ID	Browser	Plug-in	OS	Connection
TC_1	Netscape	Realplayer	Windows	LAN
TC_2	Netscape	2	Linux	PPP
TC_3	Netscape	Mediaplayer	Macintosh	ISDN
TC_4	IE	Realplayer	Linux	ISDN
TC_5	IE	2	Macintosh	LAN
TC_6	IE	Mediaplayer	Windows	PPP
TC_7	Mozilla	Realplayer	Macintosh	PPP
TC_8	Mozilla	2	Windows	ISDN
TC_9	Mozilla	Mediaplayer	Linux	LAN

TABLE 9.9 Generated Test Cases after Mapping Left-Over Levels

Test Case ID	Browser	Plug-in	OS	Connection
TC_1	Netscape	Realplayer	Windows	LAN
TC_2	Netscape	Realplayer	Linux	PPP
TC_3	Netscape	Mediaplayer	Macintosh	ISDN
TC_4	IE	Realplayer	Linux	ISDN
TC_5	IE	Mediaplayer	Macintosh	LAN
TC_6	IE	Mediaplayer	Windows	PPP
TC_7	Mozilla	Realplayer	Macintosh	PPP
TC_8	Mozilla	Realplayer	Windows	ISDN
TC_9	Mozilla	Mediaplayer	Linux	LAN

over for variable Plug-in after mapping the factors. One must provide a value in the cell. The choice of this value can be arbitrary, but to have a coverage, start at the top of the Plug-in column and cycle through the possible values when filling in the left-over levels. Table 9.9 shows the mapping after filling in the remaining levels using the cycling technique mentioned.

Step 6: We generate nine test cases taking the test case values from each run. Now let us examine the result:

- Each Browser is tested with every Plug-in, with every OS, and with every Connection.
- Each Plug-in is tested with every Browser, with every OS, and with every Connection.
- Each OS is tested with every Browser, with every Plug-in, and with every Connection.
- Each Connection is tested with every Browser, with every Plug-in, and with every OS.

9.3.2 In Parameter Order

Tai and Lei [8] have given an algorithm called in parameter order (IPO) to generate a test suite for pairwise coverage of input variables. The algorithm generates a test suite that satisfies pairwise coverage for the values of the first two parameters. Then the test suite is extended by the algorithm to satisfy pairwise coverage for the values of the third parameters and continues to do so for the values of each additional parameter until all parameters are included in the test suite.

The algorithm runs in three phases, namely, *initialization*, *horizontal growth*, and *vertical growth*, in that order. In the initialization phase, test cases are generated to cover two input variables. In the horizontal growth phase, the existing test cases are extended with the values of the other input variables. In the vertical growth phase, additional test cases are created such that the test suite satisfies pairwise

coverage for the values of the new variables. In order to use the IPO test generation techniques, one can follow the steps described below. Assume that there are n number of variables denoted by $\{p_i | 1 \leq i \leq n\}$ and a dash denotes an unspecified value of a variable.

Algorithm: In Parameter Order

Input: Parameter p_i and its domain $D(p_i) = \{v_1, v_2, \ldots, v_q\}$, where $i = 1, \ldots, n$.

Output: A test suite T satisfying pairwise coverage.

Initialization Phase:

Step 1: For the first two parameters p_1 and p_2, generate the test suite

$$T := \{(v_1, v_2) | v_1 \text{ and } v_2 \text{ are values of } p_1 \text{ and } p_2, \text{respectively}\}$$

Step 2: If $i = 2$, stop. Otherwise, for $i = 3, 4, \ldots, n$ repeat **steps 3** and **4**.

Horizontal Growth Phase:

Step 3: Let $D(p_i) = \{v_1, v_2, \ldots, v_q\}$. Create a set

$$\pi_i := \{\text{pairs between values of } p_i \text{ and all values of } p_1, p_2, \ldots, p_{i-1}\}$$

```
If |T| ≤ q, then
{ for 1 ≤ j ≤ |T|, extend the jth test in T by adding
   values vj and remove from πi pairs covered by the
   extended test}
else
{ for 1 ≤ j ≤ q, extend the jth test in T by adding
   value vj and remove from πi pairs covered by the
   extended test;
for q < j ≤ |T|, extend the jth test in T by adding
   one value of pi such that the resulting test covers
   the most numbers of pairs in πi, and remove from
   πi pairs covered by the extended test };
```

Vertical Growth Phase:

Step 4: Let $T' := \Phi$ (empty set) and $|\pi_i| > 0$;

```
for each pair in πi (let the pairs contain value w
    of pk, 1 ≤ k < i, and values u of pi)
{
if (T′ contains a test with − as the value of pk and u
    as the value of pi)
```

```
modify this test by replacing the — with w;
else
  add a new test to T′ that has w as the value of pk,
  u as the value of pi, and — as the value of every
  other parameter;
};
T := T ∪ T′;
```

The test cases may contain — values after the generation of test suite T. If p_i is the last parameter, each — value p_k, $1 \leq k \leq i$, is replaced with any value of p_k. Otherwise, these — values are replaced with parameter values in the horizontal growth phase for p_{i+1} as follows: Assuming that value v of p_{i+1} is chosen for the horizontal growth of a test that contains — as the value for p_k, $1 \leq k \leq i$. If there are uncovered pairs involving v and some values of p_k, the — for p_k is replaced with one of these values of p_k. Otherwise, the — for p_k is replaced with any value of p_k.

Example: Consider the system S in Figure 9.7 which has three input parameters X, Y, and Z. Assume that a set D, a set of input test data values, has been selected for each input variable such that $D(X) = \{\text{True, False}\}$, $D(Y) = \{0, 5\}$, and $D(Z) = \{P, Q, R\}$. The total number of possible test cases is $2 \times 2 \times 3 = 12$, but the IPO algorithm generates six test cases. Let us apply step 1 of the algorithm.

Step 1: Generate a test suite consisting of four test cases with pairwise coverage for the first two parameters X and Y:

$$T = \begin{bmatrix} (\text{True}, & 0) \\ (\text{True}, & 5) \\ (\text{False}, & 0) \\ (\text{False}, & 5) \end{bmatrix}$$

Step 2: $i = 3 > 2$; therefore, steps 2 and 3 must be executed.

Step 3: Now $D(Z) = \{P, Q, R\}$. Create a set $\pi_3 := \{$ pairs between values of Z and values of $X, Y\}$, which is

$$\pi_3 = \begin{bmatrix} (\text{True}, P), & (\text{True}, Q), & (\text{True}, R) \\ (\text{False}, P), & (\text{False}, Q), & (\text{False}, R) \\ (0, P), & (0, Q), & (0, R) \\ (5, P), & (5, Q), & (5, R) \end{bmatrix}$$

Since Z has three values, we have $q = 3$ and $|T| = 4 > q = 3$. We extended (True, 0), (True, 5), and (False, 0) by adding P, Q, and R, respectively. Next, we remove the pairs (True, P), (True, Q), (False, R), (0, P), (5, Q), and (0, R) from the set π_3 because these pairs are covered

by the partially extended test suite. The extended test suite T and π_3 become

$$T = \begin{bmatrix} (\text{True}, & 0, & P) \\ (\text{True}, & 5, & Q) \\ (\text{False}, & 0, & R) \\ (\text{False}, & 5, &) \end{bmatrix}$$

$$\pi_3 = \begin{bmatrix} & & (\text{True}, R) \\ (\text{False}, P), & (\text{False}, Q) & \\ & (0, Q) & \\ (5, P), & (5, R) & \end{bmatrix}$$

Now we need to select one of P, Q, and R for (False, 5). If we add P to (False, 5), the extended test (False, 5, P) covers two missing pairs (False, P) and (5, P). If we add Q to (False, 5), the extended test (False, 5, Q) covers only one missing pair (False, Q). If we add R to (False, 5), the extended test (False, 5, R) covers only one missing pair (5, R). Therefore, the algorithm will choose (False, 5, P) as the fourth test case. Remove the pairs (False, P) and (5, P) from the set π_3 because these pairs are covered by the partial extended test suite. Now the extended test suite T and π_3 become

$$T = \begin{bmatrix} (\text{True}, & 0, & P) \\ (\text{True}, & 5, & Q) \\ (\text{False}, & 0, & R) \\ (\text{False}, & 5, & P) \end{bmatrix} \qquad \pi_3 = \begin{bmatrix} (\text{True}, R) \\ (\text{False}, Q) \\ (0, Q) \\ (5, R) \end{bmatrix}$$

So far the tests in T have not yet covered the four tests in π_3, namely, (True, R), (False, Q), (0, Q), and (5, R).

Step 4: The algorithm will generate a set $T' = \{(\text{True}, —, R), (\text{False}, —, Q)\}$ from the first two pairs of π_3, that is, (True, R) and (False, Q). The algorithm changes the test case (False, —, Q) to (False, 0, Q) without adding a new test case to cover the next pair (0, Q). The algorithm modifies the test case (True, —, R) to (True, 5, R) without adding any new test case to be able to cover the pair (5, R). The union $T \cup T'$ generates the six pairwise test cases as follows:

$$T = \begin{bmatrix} (\text{True}, & 0, & P) \\ (\text{True}, & 5, & Q) \\ (\text{False}, & 0, & R) \\ (\text{False}, & 5, & P) \\ (\text{False}, & 0, & Q) \\ (\text{True}, & 5, & R) \end{bmatrix}$$

9.4 EQUIVALENCE CLASS PARTITIONING

An input domain may be too large for all its elements to be used as test input (Figure 9.8*a*). However, the input domain can be partitioned into a finite number of subdomains for selecting test inputs. Each subdomain is known as an equivalence class (EC), and it serves as a source of at least one test input (Figure 9.8*b*). The objective of equivalence partitioning is to divide the input domain of the system under test into classes, or groups, of inputs. All the inputs in the same class have a similar effect on the system under test [14, 15]. An EC is a set of inputs that the system treats identically when the system is tested. It represents certain conditions, or predicates, on the input domain. An input condition on the input domain is a predicate over the values of the input domain. A valid input to a system is an element of the input domain that is expected to return a nonerror value. An invalid input is an input that is expected to return an error value. Input conditions are used to partition the input domain into ECs for the purpose of selecting inputs.

Guidelines for EC Partitioning Equivalence classes can be derived from an input domain by a heuristic technique. One can approximate the ECs by identifying classes for which different program behaviors are specified. Identification of ECs becomes easier with experience. Myers suggests the following guidelines to identify ECs [16].

1. *An input condition specifies a range [a, b]*: Identify one EC for $a \leq X \leq b$ and two other classes for $X < a$ and $X > b$ to test the system with invalid inputs.
2. *An input condition specifies a set of values*: Create one EC for each element of the set and one EC for an invalid member. For example, if the input is selected from a set of N items, then $N + 1$ ECs are created: (i) one EC for each element of the set $\{M_1\}, \{M_2\}, \ldots, \{M_N\}$ and (ii) one EC for elements outside the set $\{M_1, M_2, \ldots, M_N\}$.
3. *Input condition specifies for each individual value*: If the system handles each valid input differently, then create one EC for each valid input. For

(*a*) Input domain

(*b*) Input domain partitioned into four subdomains

Figure 9.8 (a) Too many test inputs; (b) one input selected from each subdomain.

example, if the input is from a menu, then create one EC for each menu item.

4. *An input condition specifies the number of valid values (say N)*: Create one EC for the correct number of inputs and two ECs for invalid inputs—one for zero values and one for more than *N* values. For example, if a program can accept 100 natural numbers for sorting, then three ECs are created: (i) one for 100 valid input of natural numbers, (ii) one for no input value, and (iii) one for more than 100 natural numbers.
5. *An input condition specifies a "must-be" value*: Create one EC for a must-be value and one EC for something that is not a must-be value. For example, if the first character of a password must be a numeric character, then we are required to generate two ECs: (i) one for valid values, {pswd | the first character of pswd has a numeric value}, and (ii) one for invalid values, {pswd | the first character of pswd is not numeric}.
6. *Splitting of EC*: If elements in a partitioned EC are handled differently by the system, then split the EC into smaller ECs.

Identification of Test Cases from ECs Having identified the ECs of an input domain of a program, test cases for each EC can be identified by the following:

Step 1: Assign a unique number to each EC.

Step 2: For each EC with valid input that has not been covered by test cases yet, write a new test case covering as many uncovered ECs as possible.

Step 3: For each EC with invalid input that has not been covered by test cases, write a new test case that covers one and only one of the uncovered ECs.

In summary, the advantages of EC partitioning are as follows:

- A small number of test cases are needed to adequately cover a large input domain.
- One gets a better idea about the input domain being covered with the selected test cases.
- The probability of uncovering defects with the selected test cases based on EC partitioning is higher than that with a randomly chosen test suite of the same size.
- The EC partitioning approach is not restricted to input conditions alone; the technique may also be used for output domains.

Example: Adjusted Gross Income. Consider a software system that computes income tax based on adjusted gross income (AGI) according to the following rules:

If AGI is between \$1 and \$29,500, the tax due is 22% of AGI.

If AGI is between \$29,501 and \$58,500, the tax due is 27% of AGI.

If AGI is between \$58,501 and \$100 billion, the tax due is 36% of AGI.

TABLE 9.10 Generated Test Cases to Cover Each Equivalence Class

Test Case Number	Test Value	Expected Result	Equivalence Class Being Tested
TC_1	\$22,000	\$4,840	**EC1**
TC_2	\$46,000	\$12,420	**EC3**
TC_3	\$68,000	\$24,480	**EC4**
TC_4	\$-20,000	Rejected with an error message	**EC2**
TC_5	\$150 billion	Rejected with an error message	**EC5**

In this case, the input domain is from \$1 to \$100 billion. There are three input conditions in the example:

1. $\$1 \leq \text{AGI} \leq \$29{,}500$.
2. $\$29{,}501 \leq \text{AGI} \leq \$58{,}500$.
3. $\$58{,}501 \leq \text{AGI} \leq \$100$ billion.

First we consider condition 1, namely, $\$1 \leq \text{AGI} \leq \$29{,}500$, to derive two ECs:

EC1: $\$1 \leq \text{AGI} \leq \$29{,}500$; valid input.

EC2: $\text{AGI} < 1$; invalid input.

Then, we consider condition 2, namely, $\$29{,}501 \leq \text{AGI} \leq \$58{,}500$, to derive one EC:

EC3: $\$29{,}501 \leq \text{AGI} \leq \$58{,}500$; valid input.

Finally, we consider condition 3, namely, $\$58{,}501 \leq \text{AGI} \leq \$100$ billion, to derive two ECs:

EC4: $\$58{,}501 \leq \text{AGI} \leq \$100$ billion; valid input.

EC5: $\text{AGI} > \$100$ billion; invalid input.

Note that each condition was considered separately in the derivation of ECs. Conditions are not combined to select ECs. Five test cases are generated to cover the five ECs, as shown in Table 9.10.

In the EC partition technique, a single test input is arbitrarily selected to cover a specific EC. We need to generate specific test input by considering the extremes, either inside or outside of the defined EC partitions. This leads us to the next technique, known as boundary value analysis, which focuses on the boundary of the ECs to identify test inputs.

9.5 BOUNDARY VALUE ANALYSIS

The central idea in boundary value analysis (BVA) is to select test data near the boundary of a data domain so that data both within and outside an EC are selected.

It produces test inputs near the boundaries to find failures caused by incorrect implementation of the boundaries. Boundary conditions are predicates that apply directly on and around the boundaries of input ECs and output ECs. In practice, designers and programmers tend to overlook boundary conditions. Consequently, defects tend to be concentrated near the boundaries between ECs. Therefore, test data are selected on or near a boundary. In that sense, the BVA technique is an extension and refinement of the EC partitioning technique [17]. In the BVA technique, the boundary conditions for each EC are analyzed in order to generate test cases.

Guidelines for BVA As in the case of EC partitioning, the ability to develop high-quality effective test cases using BVA requires experience. The guidelines discussed below are applicable to both input conditions and output conditions. The conditions are useful in identifying high-quality test cases. By high-quality test cases we mean test cases that can reveal defects in a program.

1. *The EC specifies a range*: If an EC specifies a range of values, then construct test cases by considering the boundary points of the range and points just beyond the boundaries of the range. For example, let an EC specify the range of $-10.0 \leq X \leq 10.0$. This would result in test data $\{-9.9 - 10.0, -10.1\}$ and $\{9.9, 10.0, 10.1\}$.
2. *The EC specifies a number of values*: If an EC specifies a number of values, then construct test cases for the minimum and the maximum value of the number. In addition, select a value smaller than the minimum and a value larger than the maximum value. For example, let the EC specification of a student dormitory specify that a housing unit can be shared by one to four students; test cases that include 1, 4, 0, and 5 students would be developed.
3. *The EC specifies an ordered set*: If the EC specifies an ordered set, such as a linear list, table, or sequential file, then focus attention on the first and last elements of the set.

Example: Let us consider the five ECs identified in our previous example to compute income tax based on AGI. The BVA technique results in test as follows for each EC. The redundant data points may be eliminated.

EC1: $\$1 \leq \text{AGI} \leq \$29{,}500$; This would result in values of \$1, \$0, \$−1, \$1.50 and \$29,499.50, \$29,500, \$29,500.50.

EC2: $\text{AGI} < 1$; This would result in values of \$1, \$0, \$−1, \$−100 billion.

EC3: $\$29{,}501 \leq \text{AGI} \leq \$58{,}500$; This would result in values of \$29,500, \$29,500.50, \$29,501, \$58,499, \$58,500, \$58,500.50, \$58,501.

EC4: $\$58{,}501 \leq \text{AGI} \leq \$100$ billion; This would result in values of \$58,500, \$58,500.50, \$58,501, \$100 billion, \$101 billion.

EC5: AGI > \$100 billion; This would result in \$100 billion, \$101 billion, \$10000 billion.

Remark. Should we test for an AGI value of \$29,500.50 (i.e., between the partitions), and if so, what should be the result? Since we have not been told whether the decimal values are actually possible, the best decision to make is to test for this value and report the result.

9.6 DECISION TABLES

A major limitation of the EC-based testing is that it only considers each input separately. The technique does not consider combining conditions. Different combinations of equivalent classes can be tried by using a new technique based on the *decision table* to handle multiple inputs. Decision tables have been used for many years as a useful tool to model software requirements and design decisions. It is a simple, yet powerful notation to describe complex systems from library information management systems to embedded real-time systems [18].

The general structure of a decision table is shown in Table 9.11. It comprises a set of *conditions* (or *causes*) and a set of *effects* (or *results*) arranged in the form of a column on the left of the table. In the second column, next to each condition, we have its possible values: yes (Y), no (N), and don't care (dash). To the right of the values column, we have a set of *rules*. For each combination of the three conditions $\{C_1, C_2, C_3\}$, there exists a rule from the set $\{R_1, R_2, \cdots, R_8\}$. Each rule comprises a yes, no, or don't care response and contains an associated list of effects $\{E_1, E_2, E_3\}$. Then, for each relevant effect, an *effect sequence number* specifies the order in which the effect should be carried out if the associated set of conditions are satisfied. For example, if C_1 and C_2 are true but C_3 is not true, then E_3 should be followed by E_1. The checksum is used for verification of the combinations the decision table represents.

TABLE 9.11 Decision Table Comprising Set of Conditions and Effects

		Rules or Combinations							
Conditions	Values	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
C_1	Y, N,—	Y	Y	Y	Y	N	N	N	N
C_2	Y, N,—	Y	Y	N	N	Y	Y	N	N
C_3	Y, N,—	Y	N	Y	N	Y	N	Y	N
Effects									
E_1		1		2	1				
E_2			2	1			2	1	
E_3		2	1	3		1	1		
Checksum	8	1	1	1	1	1	1	1	1

Test data are selected so that each rule in a table is exercised and the actual results are verified with the expected results. In other words, each rule of a decision table represents a test case. The steps in developing test cases using the decision table technique are as follows:

Step 1: The test designer needs to identify the conditions and the effects for each specification unit. A condition is a distinct input condition or an EC of input conditions. An effect is an output condition. Determine the logical relationship between the conditions and the effects.

Step 2: List all the conditions and effects in the form of a decision table. Write down the values the condition can take.

Step 3: Calculate the number of possible combinations. It is equal to the number of different values raised to the power of the number of conditions.

Step 4: Fill the columns with all possible combinations—each column corresponds to one combination of values. For each row (condition) do the following:

1. Determine the repeating factor (RF): divide the remaining number of combinations by the number of possible values for that condition.
2. Write RF times the first value, then RF times the next, and so forth until the row is full.

Step 5: Reduce combinations (rules). Find indifferent combinations—place a dash and join column where columns are identical. While doing this, ensure that effects are the same.

Step 6: Check covered combinations (rules). For each column calculate the combinations it represents. A dash represents as many combinations as the condition has. Multiply for each dash down the column. Add up total and compare with step 3. It should be the same.

Step 7: Add effects to the column of the decision table. Read column by column and determine the effects. If more than one effect can occur in a single combination, then assign a sequence number to the effects, thereby specifying the order in which the effects should be performed. Check the consistency of the decision table.

Step 8: The columns in the decision table are transformed into test cases.

Decision table–based testing is effective under certain conditions as follows:

- The requirements are easily mapped to a decision table.
- The resulting decision table should not be too large. One can break down a large decision table into multiple smaller tables.
- Each column in a decision table is independent of the other columns.

Example: Let us consider the following description of a payment procedure. Consultants working for more than 40 hours per week are paid at their hourly rate for the first 40 hours and at two times their hourly rate for subsequent hours. Consultants working for less than 40 hours per week are paid for the hours worked

at their hourly rates and an absence report is produced. Permanent workers working for less than 40 hours a week are paid their salary and an absence report is produced. Permanent workers working for more than 40 hours a week are paid their salary. We need to describe the above payment procedure using a decision table and generate test cases from the table.

Step 1: From the above description, the conditions and effects are identified as follows:

C_1: Permanent workers

C_2: Worked < 40 hours

C_3: Worked exactly 40 hours

C_4: Worked > 40 hours

E_1: Pay salary

E_2: Produce an absence report

E_3: Pay hourly rate

E_4: Pay $2 \times$ hourly rate

Step 2: The decision table with all the conditions and the effects are shown in Table 9.12.

Step 3: The total number of combinations is $2^4 = 16$.

Step 4: The RFs for row 1, row 2, row 3, and row 4 are $\frac{16}{2} = 8$, $\frac{8}{2} = 4$, $\frac{4}{2} = 2$, and $\frac{2}{2} = 1$, respectively. Therefore, the first row is filled with eight Y followed by eight N. The second row is filled with four Y, followed by four N, and so on.

Step 5: If condition C_1: Permanent workers is yes and condition C_2: Worked < 40 hours is yes, then conditions C_3: Worked exactly 40 hours and C_4: worked > 40 hours do not matter. Therefore, rules 1, 2, 3, and 4 can be reduced to a single rule without impacting the effects.

TABLE 9.12 Pay Calculation Decision Table with Values for Each Rule

		Rules or Combinations															
Conditions	Values	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C_1	Y, N	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
C_2	Y, N	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
C_3	Y, N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
C_4	Y, N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Effects																	
E_1																	
E_2																	
E_4																	
E_4																	

If condition C_1: Permanent workers is yes and condition C_2: Worked < 40 hours is no, then conditions C_3: Worked exactly 40 hours and C_4: worked > 40 hours do not matter. Therefore, rules 5, 6, 7, and 8 can be reduced to a single rule—permanent workers get paid regardless.

If condition C_1: Permanent workers is no and condition C_2: Worked < 40 hours is yes, then conditions C_3: Worked exactly 40 hours and C_4: worked > 40 hours are immaterial. Therefore, rules 9, 10, 11, and 12 can be reduced to a single rule without impacting the effects.

If conditions C_1: Permanent workers and C_2: Worked < 40 hours are no but condition C_3: Worked exactly 40 hours is yes, then rules 13 and 14 can be reduced to a single rule.

Rules 15 and 16 stand as they are. In summary, 16 rules can be reduced to a total of 6 rules, which are shown in Table 9.13.

Step 6: The checksum for columns 1, 2, 3, 4, 5, and 6 are 4, 4, 4, 2, 1, and 1, respectively, as shown in Table 9.14. The total checksum is 16, which is the same as calculated in step 3.

Step 7: In this step, the effects are included for each column (rule). For the first column, if the conditions C_1: Permanent workers and C_2: Worked < 40 hours are satisfied, then the employee must be paid and an absence report must be generated; therefore, E_1: Pay salary and E_2: Produce an absence report are marked as 1 and 2 in the decision table, respectively—the 1 and 2 indicating the order in which the effects are expected. The final decision table with effects are shown in Table 9.14. Note that for column 6 no effects are marked.

Step 8: A test case purpose can be generated from column 1 which can be described as follows: If an employee is a permanent worker and worked less than 40 hours per week, then the system should pay his or her salary and generate an absence report. Similarly, other test cases can be generated from the rest of the columns.

TABLE 9.13 Pay Calculation Decision Table after Column Reduction

		Rules or Combinations					
Conditions	Values	1	2	3	4	5	6
C_1	Y, N	Y	Y	N	N	N	N
C_2	Y, N	Y	N	Y	N	N	N
C_3	Y, N	—	—	—	Y	N	N
C_4	Y, N	—	—	—	—	Y	N
Effects							
E_1							
E_2							
E_4							
E_4							

TABLE 9.14 Decision Table for Payment Calculation

		Rules or Combinations					
Conditions	Values	1	2	3	4	5	6
C_1	Y, N	Y	Y	N	N	N	N
C_2	Y, N	Y	N	Y	N	N	N
C_3	Y, N	—	—	—	Y	N	N
C_4	Y, N	—	—	—	—	Y	N
Effects							
E_1							
E_1		1	1				
E_2		2		2			
E_4				1	1	1	
E_4						2	
Checksum	16	4	4	4	2	1	1

9.7 RANDOM TESTING

In the random testing approach, test inputs are selected randomly from the input domain of the system. We explain the idea of random testing with a simple example of computing $\sqrt{X}$, where X is an integer. Suppose that the system will be used in an environment where the input X takes on all values from the interval $[1, 10^8]$ with equal likelihood and that the result must be accurate to within 2×10^{-4}. In order to test this program, one can generate uniformly distributed pseudorandom integers within the interval $[1, 10^8]$. Then we execute the program on each of these inputs t and obtain the output z_t. For each t, we compute z_t and z_t^2 and compare z_t^2 with t. If any of the outputs fails to be within 2×10^{-4} of the desired results, the program must be fixed and the test repeated. Based on the above example, random testing can be summarized as a four-step procedure [19]:

Step 1: The input domain is identified.

Step 2: Test inputs are selected independently from the domain.

Step 3: The system under test is executed on these inputs. The inputs constitute a random test set.

Step 4: The results are compared to the system specification. The test is a failure if any input leads to incorrect results; otherwise it is a success.

Random testing corresponds to simple random sampling from the input domain [20]. If the distribution of the selected inputs (step 2) is the same as the distribution of inputs in the expected-use scenario (the operational profile), then statistical estimates for the reliability of the program can be obtained from test outcomes. Random testing gives us an advantage of easily estimating software reliability from test outcomes. Test inputs are randomly generated according to an

operational profile, and failure times are recorded. The data obtained from random testing can then be used to estimate reliability. Other testing methods cannot be used in this way to estimate software reliability.

A large number of test inputs are typically required to get meaningful statistical results. Consequently, some kind of automation is required to generate a large number of inputs for random testing. For effective generation of a large set of inputs for statistical estimation, one needs to know the operational profile of the system. On the other hand, the expected results (step 4) are usually not obvious. Computing expected outcomes becomes difficult if the inputs are randomly chosen. Therefore, the technique requires good test oracles to ensure the adequate evaluation of test results. A test oracle is a mechanism that verifies the correctness of program outputs. The term *test oracle* was coined by William E. Howden [21]. An oracle provides a method to (i) generate expected results for the test inputs and (ii) compare the expected results with the actual results of execution of the implementation under test (IUT). In other words, it consists of two parts: a result generator to obtain expected results and a comparator. Four common types of oracles are as follows [22]:

- **Perfect Oracle:** In this scheme, the system (IUT) is tested in parallel with a trusted system that accepts every input specified for the IUT and always produces the correct result. A trusted system is a defect-free version of the IUT.
- **Gold Standard Oracle:** A previous versions of an existing application system is used to generate expected results, as shown in Figure 9.9.
- **Parametric Oracle:** An algorithm is used to extract some parameters from the actual outputs and compare them with the expected parameter values, as shown in Figure 9.10.
- **Statistical Oracle:** It is a special case of a parametric oracle. In a statistical oracle, statistical characteristics of the actual test results are verified.

Figure 9.9 Gold standard oracle.

Figure 9.10 Parametric oracle.

Moreover, the actual test results are random in the case of randomized software and random testing. Therefore, it is not possible to give an exact expected value. In this scheme, the expected statistical characteristic is compared with the actual test results. A statistical oracle does not check the actual output but only some characteristics of it. Therefore, a statistical oracle cannot decide whether or not a single test case passes. If a failure occurs, identification of the failure cannot be attributed to the success of a single test case; rather, the entire group of test cases is credited with the success. The decision of a statistical oracle is not always correct. In other words, at best the probability for a correct decision can be given. Figure 9.11 shows the structure of a statistical oracle [23]. It consists of a statistical analyzer and a comparator. The statistical analyzer computes various characteristics that may be modeled as random variables and delivers it to the comparator. The comparator computes the empirical sample mean and the empirical sample variance of its inputs. Furthermore, expected values and properties of the characteristics are computed by the comparator based on the distributional parameters of the random test input.

Adaptive Random Testing In adaptive random testing the test inputs are selected from the randomly generated set in such a way that these are evenly spread over the entire input domain. The goal is to select a small number of test inputs to detect the first failure. A number of random test inputs are generated, then the "best" one among them is selected. We need to make sure the selected new test input is not too close to any of the previously selected ones. That is, the selected test inputs should be distributed as spaced out as possible.

An adaptive random testing technique proposed by Chen et al. [24] keeps two sets, namely, T and C, as follows:

- *The executed set* T is the set of distinct test inputs that have been selected and executed without revealing any failure.
- *The candidate set* C is a set of test inputs that are randomly selected.

Initially the set T is empty, and the first test input is randomly selected from the input domain. The set T is then incrementally updated with the selected element from the set C and executed until a failure is revealed. From the set C, an element that is farthest away from all the elements in the set T is selected as the next test input. The criterion "farthest away" can be defined as follows. Let

Figure 9.11 Statistical oracle.

$T = \{t_1, t_2, \ldots, t_n\}$ be the executed set and $C = \{c_1, c_2, \ldots, c_k\}$ be the candidate set such that $C \cap T = \phi$. The criterion is to select the element c_h such that, for all $j \in \{1, 2, \ldots, k\}$ and $j \neq h$,

$$\min_{i=1}^{n} \text{dist}(c_h, t_i) \geq \min_{i=1}^{n} \text{dist}(c_j, t_i)$$

where dist is defined as the Euclidean distance. In an m-dimensional input domain, for inputs $a = (a_1, a_2, \ldots, a_m)$ and $b = (b_1, b_2, \ldots, b_m)$, $\text{dist}(a, b) = \sqrt{\sum_{i=1}^{m}(a_i - b_i)^2}$. The rationale of this criterion is to evenly spread the test input by maximizing the minimum distance between the next test input and the already executed test cases.

It should be noted that there are various ways to construct the candidate set C giving rise to various versions of adaptive random testing. For example, a new candidate set can be constructed of size 10 each time a test input is selected. Empirical study shows that adaptive random testing does outperform ordinary random testing by 50% [24]. In the above comparison the performance metric is the size of the test suite used to detect the first failure.

9.8 ERROR GUESSING

Error guessing is a test case design technique where a test engineer uses experience to (i) guess the types and probable locations of defects and (ii) design tests specifically to reveal the defects. For example, if memory is allocated dynamically, then a good place to look for error is the portion of the code after the allocated memory is used—there is a possibility that unused memory is not deallocated. An experienced test engineer can ask the question: Are all the allocated memory blocks correctly deallocated?

Though experience is of much use in guessing errors, it is useful to add some structure to the technique. It is good to prepare a list of types of errors that can be uncovered. The error list can aid us in guessing where errors may occur. Such a list should be maintained from experience gained from earlier test projects. The following are the critical areas of the code where defects are most likely to be found:

- Different portions of the code have different complexity. One can measure code complexity by means of cyclomatic complexity. Portions of the code with a high cyclomatic complexity are likely to have defects. Therefore, it is productive to concentrate more efforts on those portions of code.
- The code that has been recently added or modified can potentially contain defects. The probability of inadvertently introducing defects with addition and modification of code is high.
- Portions of code with prior defect history are likely to be prone to errors. Such code blocks are likely to be defective, because of clustering tendency of the defects, despite the efforts to remove the defects by rewriting the code.

- Parts of a system where new, unproven technology has been used is likely to contain defects. For example, if the code has been automatically generated from a formal specification of the system, then there is higher possibility of defects imbedded into the code.
- Portions of the code for which the functional specification has been loosely defined can be more defective.
- Code blocks that have been produced by novice developers can be defective. If some developers have not been careful during coding in the past, then any code written by these developers should be examined in greater detail.
- Code segments for which a developer may have a low confidence level should receive more attention. The developers know the internal details of the system better than anyone else. Therefore, they should be quizzed on their comfort levels and more test effort be put on those areas where a developer feels less confident about their work.
- Areas where the quality practices have been poor should receive additional attention. An example of poor quality practice is not adequately testing a module at the unit level. Another example of poor quality practice is not performing code review for a critical part of a module.
- A module that involved many developers should receive more test effort. If several developers worked on a particular part of the code, there is a possibility of misunderstanding among different developers and, therefore, there is a good possibility of errors in these parts of the code.

9.9 CATEGORY PARTITION

The category partition method (CPM) is a generalization and formalization of a classical functional testing approach. The reader is reminded of the two steps of the classical functional testing approach: (i) partition the input domain of the functional unit to be tested into equivalence classes and (ii) select test data from each EC of the partition. The CPM [25] is a systematic, specification-based methodology that uses an informal functional specification to produce formal test specification. The test designer's key job is to develop *categories*, which are defined to be the major characteristics of the input domain of the function under test. Each category is partitioned into ECs of inputs called *choices*. The choices in each category must be disjoint, and together the choices in each category must cover the input domain. In a later paper [26], Grochtmann and Grimm extend this approach by capturing the constraints using a tree structure to reduce the number of impossible test cases.

The main advantage of this approach is the creation of a formal test specification written in languages such as test specification language (TSL) [27], Z [28], [29], or testing and test control notation (TTCN) [30] to represent an informal or natural language formal specification. The formal test specification gives

the test engineer a logical way to control test generation and is easily modified to accommodate changes or mistakes in the functional specification. The use of Z allows more flexibility in the specification of constraints and more formalities in the representation.

The category partition testing method is comprised of the following steps:

Step 1: Analyze the Specification The method begins with the decomposition of a functional specification into functional units that can be separately tested. For each functional unit, identify the following:

- Parameters of the functional unit
- Characteristics of each parameter, that is, the elementary characteristics of the parameters that affect the execution of the unit
- Objects in the environment whose state might affect the operation of the functional unit
- Characteristics of each environment object

Parameters are the explicit input to a functional unit, and environment conditions are the state characteristics of the system at the time of execution of a functional specification.

Step 2: Identify Categories A category is a classification of the major properties (or characteristics) of a parameter or an environmental condition. Consider, for instance, a program P_{SORT} that reads an input file F containing a variable-length array of values of arbitrary type. The expected output is a permutation of input with values sorted according to some total ordering criterion. The environmental condition for P_{SORT} is status of F, which can be classified into three categories, namely Status of $F =$ Does Not Exist, Status of $F =$ Exists But Empty, and Status of $F =$ Exists and Nonempty. The properties of the input parameter categories for the P_{SORT} are as follows: the array size, the type of elements, the minimum element value, the maximum element value, and the positions in the array of the maximum and minimum values. Categories can be derived directly from the functional specification, implicit design information, or intuition of the test engineer. Often categories are derived from preconditions and type information about the input parameters and system state components.

Step 3: Partition Categories into Choices Partition each category into distinct choices that include all the different kinds of values that are possible for the category. Each choice is an equivalent class of values assumed to have identical properties as far as testing and error detection capability are concerned. The choices must be disjoint and cover all the categories. While the categories are derived from a functional specification, the choices can be based on specification and the test engineer's past experience of designing effective test cases, such as error guessing.

In the sorting program P_{SORT} example, one possible way to partition the category *array size* is size $= 0$, size $= 1$, $2 \Leftarrow$ size $\Leftarrow 100$, and size > 100. These choices are based primarily on experience with likely

errors. The selection of a single choice from each category determines a *test frame*, which is the basis for constructing the actual test cases. A test frame consists of a set of choices from the specification, with each category contributing either zero or one choice. Since the choices in different categories frequently interact with each other in ways that affect the resulting test cases, the choices can be annotated with *constraints* (see next step) to indicate these relations. In the absence of constraints the number of potential test frames is the product of the number of choices in each category—and this is likely to be very large.

Step 4: Determine Constraints among Choices Constraints are restrictions among the choices within different categories that can interact with one another. A typical constraint specifies that a choice from one category cannot occur together in a test frame with certain choices from another category. Choices and constraints are derived from the natural language functional specifications but can often be specified by formal methods, thus making their analysis easier to automate. With a careful specification of constraints, the number of potential test frames can be reduced to a manageable number.

Step 5: Formalize and Evaluate Test Specification Specify the categories, choices, and constraints using a specification technique that is compatible with the test generation tool, such as TSL, Z, or TTCN, that produces test frames. Most test generation tools also provide automated techniques for evaluating the internal consistency of the formal specification. This evaluation often discovers errors or inconsistencies in the specification of constraints and sometimes leads to discovery of errors in the source functional specification.

Step 6: Generate and Validate Test Cases The final step in the test production process is to transform the generated test frames into executable test cases. If the test specification includes postconditions that must be satisfied, then the tool verifies the postconditions. In case a reference implementation is available, the test cases can be validated by executing them and checking their results against the reference implementation. The validation of test cases is a labor-intensive process in the absence of a reference implementation [31].

9.10 SUMMARY

This chapter began with an introduction to the concept of functional testing, which consists of (i) precise identification of the domain of each input and each output variable, (ii) selection of values from a data domain having important properties, and (iii) combination of values of different variables. Next, we examined four different types of variables, namely, numeric, arrays, substructures, and strings. Only functionally related subsets of input variables are considered to be combined

to reduce the total number of input combinations [3, 4]. In this way, the total number of combinations of special values of the variables is reduced. Then, we discussed the scope and complexity of functional testing. Next, we introduced seven testing techniques as summarized in the following:

- **Pairwise Testing:** Pairwise testing requires that for a given numbers of input parameters to the system each possible combination of values for any pair of parameters be covered by at least one test case. It is a special case of combinatorial testing that requires n-way combinations be tested, $n = 1, 2, \ldots, N$, where N is the total number of parameters in the system. We presented two popular pairwise test selection techniques, namely, OA and IPO.
- **Equivalence Class Partitioning:** The aim of EC partitioning is to divide the input domain of the system under test into classes (or groups) of test cases that have a similar effect on the system. Equivalence partitioning is a systematic method for identifying sets of interesting classes of input conditions to be tested. We provided guidelines to identify (i) ECs and (ii) test data from the ECs that need to be executed.
- **Boundary Value Analysis:** The boundary conditions for each EC are analyzed in order to generate test cases. Boundary conditions are predicates that apply directly on, above, and beneath the boundaries of input ECs and output ECs. We explained useful guidelines which are applicable to the input and the output conditions of the identified EC. These are useful in identifying quality test cases.
- **Decision Tables:** This is a simple but powerful technique to describe a complex system. A decision table comprises a set of conditions placed above a set of effects (or results) to perform in a matrix form. There exists a rule for each combination of conditions. Each rule comprises a Y (yes), N (no), or—(don't care) response and contains an associated list of effects. Thus, each rule of the decision table represents a test case.
- **Random Testing:** The random testing technique can be summarized as a four-step procedure:

  1. The input domain is identified.
  2. The test inputs are independently selected from this domain.
  3. The system under test is executed on a random set of inputs.
  4. The results are compared with the system specification.

  The test is a failure if any input leads to incorrect results; otherwise it is a success.

  The procedure requires a test oracle to ensure adequate evaluation of the test results. A test oracle tells us whether a test case passes or not. We discussed four standard types of oracles, namely, perfect, gold, parametric, and statistical. We also examined the concept of adaptive random testing in which test inputs are selected from a randomly generated set. These test

inputs are evenly spread over the entire input domain in order to achieve a small number of test inputs to detect the first failure.

- **Error Guessing:** This is a test case design technique where the experience of the testers is used to (i) guess the type and location of errors and (ii) design tests specifically to expose those errors. Error guessing is an ad hoc approach in designing test cases.
- **Category Partition:** The category partition methodology requires a test engineer to divide the functional specification into independent functional units that can be tested separately. This method identifies the input parameters and the environmental conditions, known as categories, by relying upon the guidance of the test engineer. Next, the test engineer decomposes each identified category into mutually exclusive choices that are used to describe the partition of the input within the category. Then the categories, choices, and constraints are specified using a formal specification language such as TSL. The specification is then processed to produce a set of *test frames* for the functional unit. The test engineer examines the test frames and determines if any changes to the test specification are necessary. Finally, the test frames are converted to executable test cases.

LITERATURE REVIEW

The IPO algorithm discussed in this chapter has been implemented in a tool called PairTest [8] which provides a graphical interface to make the tool easy to use. A test tool called automatic efficient test generator (AETG) was created at Telcordia and published in the following two articles:

D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design," *IEEE Transactions on Software Engineering*, Vol. 23, No. 7, July 1997, pp.437–444.

D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The Combinatorial Design Approach to Automatic Test Generation," *IEEE Software*, Vol. 13, No. 5, September 1996, pp.83–89.

The strategy used in this tool starts with an empty test suite and adds one test case at a time. The tool produces a number of candidate test cases according to a greedy algorithm and then selects one that covers the most uncovered pairs.

Few topics related to software testing techniques seem to be more controversial than the question of whether it is efficient to use randomly generated test input data. The relative effectiveness of random testing (i.e., random selection of test inputs from the entire input domain) versus partition testing (i.e., dividing the input domain into nonoverlapping subdomains and selecting one test input from each subdomain) has been the subject of many research papers. The interested reader is referred to the following papers for some of these discussions:

T. Chen and Y. Yu, "On the Expected Number of Failures Detected by Subdomain Testing and Random Testing," *IEEE Transactions on Software Engineering*, Vol. 22, 1996, pp.109–119.

J .W. Duran and S. C. Ntafos, "An Evaluation of Random Testing," *IEEE Transactions on Software Engineering*, Vol. 10, July 1984, pp.438–444.

R. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence," *IEEE Transactions on Software Engineering*, Vol. 16, 1990, pp.1402–1411.

W. Gutjahr, "Partition Testing versus Random Testing: The Influence of Uncertainty," *IEEE Transactions on Software Engineering*, Vol. 25, 1999, pp.661–674.

E. J. Weyuker and B. Jeng, "Analyzing Partition Testing Strategies," *IEEE Transactions on Software Engineering*, Vol. 17, 1991, pp.703–711.

The boundary value analysis technique produces test inputs near the boundaries to find failures caused by incorrect implementation of the boundaries. However, boundary value analysis can be adversely affected by coincidental correctness, that is, the system produces the the expected output, but for the wrong reason. The article by Rob M. Hierons ("Avoiding Coincidental Correctness in Boundary Value Analysis," *ACM Transactions on Software Engineering and Methodology*, Vol. 15, No. 3, July 2006, pp. 227–241) describes boundary value analysis that can be adapted in order to reduce the likelihood of coincidental correctness. The work described by the author can be seen as a formalization and generalization of work suggested by Lori A. Clarke, Johnette Hassell, and Debra J. Richardson ("A Close Look at Domain Testing," *IEEE Transactions on Software Engineering*, Vol. 8, No. 4, July 1982, pp. 380–390).

REFERENCES

1. W. E. Howden. Functional Program Testing. *IEEE Transactions on Software Engineering*, March 1980, pp. 162–169.
2. W. E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill, New York, 1987.
3. W. E. Howden. Applicability of Software Validation Techniques to Scientific Programs. *ACM Transactions on Software Engineering and Methodology*, July 1980, pp. 307–320.
4. W. E. Howden. A Functional Approach to Program Testing and Analysis. *IEEE Transactions on Software Engineering*, October 1986, pp. 997–1005.
5. D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering*, June 2004, pp. 418–421.
6. M. Grindal, J. Offutt, and S. F. Andler. Combination Testing Strategies: A Survey. *Journal of Software Testing, Verification, and Reliability*, September 2005, pp. 97–133.
7. M. S. Phadke. *Quality Engineering Using Robust Design*. Prentice-Hall, Englewood Cliffs, NJ, 1989.
8. K. C. Tai and Y. Lei. A Test Generation Strategy for Pairwise Testing. *IEEE Transactions on Software Engineering*, January 1992, pp. 109–111.
9. L. Copeland. Object-Oriented Testing. *Software Quality Engineering*, STAR East, Orlando, FL, May 2001.
10. A. S. Hedayat, N. J. A. Sloane, and J. Stufken. *Orthogonal Arrays: Theory and Applications*, Springer Series in Statistics, Springer-Verlag, New York, 1999.

11. R. K. Roy. *Design of Experiments Using the Taguchi Approach: 16 Steps to Product and Process Improvement*. Wiley, New York, 2001.
12. B. Mandl. Orthogonal Latin Square: An Application of Experiment Design to Compiler Testing. *Communications of the ACM*, October 1985, pp. 1054–1058.
13. N. J. A. Sloane. *A Library of Orthogonnal Arrays*. Information Sciences Research Center, AT&T Shannon Labs, 2001. Available at http://www.research.att.com/~njas/oadir/.
14. B. Beizer. *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, New York, 1990.
15. D. Richardson and L. Clarke. A Partition Analysis Method to Increase Program Reliablity. In *Proceedings of the 5th International Conference Software Engineering*, San Diego, CA, IEEE Computer Society Press, Piscataway, March 1981, pp. 244–253.
16. G. Myers. *The Art of Software Testing*, 2nd ed. Wiley, New York, 2004.
17. B. Beizer. *Black Box Testing*. Wiley, New York, 1995.
18. D. Thomas. Agile Programming: Design to Accommodate Change. *IEEE Software*, May/June 2005, pp. 14–16.
19. R. Hamlet. Random Testing. *In Encyclopedia of Software Engineering*, J. Marciniak, Ed. Wiley, New York, 1994, pp. 970–978.
20. W. Cochran. *Sampling Techniques*. Wiley, New York, 1977.
21. W. E. Howden. A Survey of Dynamic Analysis Methods. In *Software Testing and Validation Techniques*, 2nd ed., E. Miller and W. E. Howden, Eds. IEEE Computer Society Press, Los Alamito, CA, 1981.
22. R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, Reading, MA, 2000.
23. J. Mayer and R. Guderlei. *Test Oracles Using Statistical Methods*. SOQUA/TECOS, Erfurt, Germany, 2004, pp. 179–189.
24. T. Y. Chen, H. Leung, and I. K. Mak. Adaptive Random Testing. *Advances in Computer Science—ASIAN 2004, Higher-Level Decision Making*, 9th Asian Computing Science Conference, Dedicated to Jean-Louis Lassez on the Occasion of His 5th Cycle Birthday, Chiang Mai, Thailand, December 8–10, 2004, M. J. Maher, Ed., *Lecture Notes in Computer Science*, Vol. 3321, Springer, Berlin/Heidelberg, 2004, pp. 320–329.
25. T. J. Ostrand and M. J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, June 1988, pp. 676–686.
26. M. Grochtmann and K. Grimm. Classification Trees for Partition Testing. *Journal of Testing, Verification, and Reliability*, June 1993, pp. 63–82.
27. M. Balcer, W. Halsing, and T. J. Ostrand. Automatic Generation of Test Scripts from Formal Test Specification. In *Proceeding of the Third Symposium on Software Testing, Analysis, and Verification*, Key West, FL, ACM Press, New York, December 1989, pp. 210–218.
28. G. Laycock. Formal Specification and Testing: A Case Study. *Journal of Testing, Verification, and Reliability*, March 1992, pp. 7–23.
29. P. Ammann and J. Offutt. Using Formal Methods to Derive Test Frames in Category-Partition Testing. Paper presented at the Ninth Annual Conference on Computer Assurance, Gaithersburg, MD, June 1994, pp. 69–80.
30. C. Willcock, T. Deiss, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN-3*. Wiley, New York, 2005.
31. K. Naik and B. Sarikaya. Test Case Verification by Model Checking. *Formal Methods in Systems Design*, June 1993, pp. 277–321.

Exercises

1. (a) What is the central idea in Howden's theory of functional testing?
   (b) What is a *functionally identifiable* substructure?
   (c) All combinations of special values of input variables can lead to a large number of test cases being selected. What technique can be used to reduce the number of combinations of test cases?

2. Consider the system S in Figure 9.7, which has three input parameters X, Y, and Z. Assume that set D, a set of input test data values, has been selected for each of the input variables such that $D(X) = \{\text{True, False}\}$, $D(Y) = \{0, 5\}$, and $D(Z) = \{P, Q, R\}$. Using the orthogonal array method discussed in this chapter, generate pairwise test cases for this system. Compare the results with the test suite generated using the IPO algorithm.
3. Discuss the drawback of the orthogonal array methodology compared to the IPO algorithm.
4. Consider the system S which can take n input parameters and each parameter can take on m values. For this system answer the following questions:
   (a) What is the maximum number of pairs a single test case for this system can cover?
   (b) In the best case, how many test cases can provide full pairwise coverage?
   (c) Calculate the total number of pairs the test suite must cover.
   (d) Suppose that $n = 13$ and $m = 3$. What is the minimum number of test cases to be selected to achieve pairwise coverage?
5. Consider the following triangle classification system, originally used by Myers [16]: The system reads in three positive values from the standard input. The three values A, B, and C are interpreted as representing the lengths of the sides of a triangle. The system then prints a message on the standard output saying whether the triangle is scalene, isosceles, equilateral, or right angled if a triangle can be formed. Answer the following questions for the above program:
   (a) What is the input domain of the system?
   (b) What are the input conditions?
   (c) Identify the equivalence classes for the system?
   (d) Identify test cases to cover the identified ECs?
6. Consider again the triangle classification program with a slightly different specification: The program reads floating values from the standard input. The three values A, B, and C are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right angled. Determine the following for the above program:
   (a) For the boundary condition $A + B > C$ case (scalene triangle), identify test cases to verify the boundary.
   (b) For the boundary condition $A = C$ case (isosceles triangle), identify test cases to verify the boundary.
   (c) For the boundary condition $A = B = C$ case (equilateral triangle), identify test cases to verify the boundary.
   (d) For the boundary condition $A^2 + B^2 = C^2$ case (right-angle triangle), identify test cases to verify the boundary.

(e) For the nontriangle case, identify test cases to explore the boundary.

(f) For nonpositive input, identify test points.

7. Consider the triangle classification specification. The system reads in three positive values from the standard input. The three values A, B, and C are interpreted as representing the lengths of the sides of a triangle. The system then prints a message to the standard output saying whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or not a triangle. Develop a decision table to generate test cases for this specification.
8. What are the advantages and disadvantages of random testing?
9. What is a test oracle? What are the differences between a parametric oracle and a statistical oracle?
10. Discuss the similarity between the decision table–based and category partition–based testing methodology.

CHAPTER 10

Test Generation from FSM Models

> The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work.
>
> — *John von Neumann*

10.1 STATE-ORIENTED MODEL

Software systems can be broadly classified into two groups, namely, *stateless* and *state-oriented* systems. The actions of a stateless system do not depend on the previous inputs to the system. A compiler is an example of a stateless system because the result of compiling a program does not depend on the programs that had been previously compiled. The response of the system to the present input depends on the past inputs to the system in a state-oriented system. A state-oriented system memorizes the sequence of inputs it has received so far in the form of a *state*. A telephone switching system is an example of a state-oriented system. The interpretation of digits by a telephone switch depends on the previous inputs, such as a phone going off the hook, the sequence of digits dialed, and other the keys pressed.

A state-oriented system can be viewed as having a *control portion* and a *data portion*. The control portion specifies the sequences of interactions with its environment, and the data portion specifies the data to be processed and saved. Depending on the characteristics of systems, a system can be predominantly data oriented, be predominantly control oriented, or have a balanced mix of both data and control, as illustrated in Figure 10.1.

In a data dominating system, the system spends most of its time processing user requests, and the interaction sequences with the user are very simple. Therefore, the control portion is simpler compared to the data processing, which is more complex. This situation is depicted in Figure 10.2.

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

Figure 10.1 Spectrum of software systems.

Figure 10.2 Data-dominated systems.

Example. A *web browsing* application is an example of data dominating systems. The system spends a significant amount of time in accessing remote data by making http requests and formatting it for display. The system responds to each command input from the user, and there is not much state information that the system must remember. A need for having state information is to perform the **Back** operation. Moreover, web browsing is not a time-dependent application, except for its dependence on the underlying Transmission Control Protocol/Internet Protocol (TCP/IP) operations.

In a control dominating system, the system performs complex (i.e., many time-dependent and long-sequence) interactions with its user, while the amount of data being processed is relatively small. Therefore, the control portion is a large one, whereas the data processing functionality is very small. This situation is depicted in Figure 10.3.

Example. A telephone switching system is an example of a control dominating system. The amount of user data processed is rather minimal. The data involved are

Figure 10.3 Control-dominated systems.

a mapping of phone numbers to equipment details, off- and on-hook events generated by a user, phone number dialed, and possibly some other events represented by the push of other keys on a telephone.

The control portion of a software system can often be modeled as a finite-state machine, (FSM), that is, the interactions between the system and its user or environment (Figures 10.2 and 10.3).

We have modeled the interactions of a user with a dual-boot laptop computer (Figure 10.4). Initially, the laptop is in the **OFF** state. When a user presses the

Figure 10.4 FSM model of a dual-boot laptop computer.

power **ON** button, the system moves to the **BOOT** state, where it receives one of two inputs **LINUX** and **WINDOWS**. If the user input is **LINUX**, then the system boots with the Linux operating system and moves to the **LINUX** state, whereas the **WINDOWS** input causes the system to boot with the Windows operating system and moves to the **WIN** state. Whether the laptop is running Linux or Windows, the user can put the machine in standby states. The standby state for the Linux mode is **LSTND** and for the Windows mode it is **WSTND**. The computer can be brought back to its operating state **LINUX** or **WIN** from a standby state **LSTND** or **WSTND**, respectively, with a **WAKEUP** input. The laptop can be moved between **LINUX** and **WIN** states using **RESTART** inputs. The laptop can be shut down using the **SHUTDOWN** input while it is in the **LINUX** or **WIN** state. The laptop can also be brought to the **OFF** state by using the power button, but we have not shown these transitions in Figure 10.4.

The reader may note that for the purpose of generating test cases we do not consider the *internal* behavior of a system; instead we assume that the external behavior of the system has been modeled as an FSM. To be more precise, the interactions of a system with its environment is modeled as an FSM, as illustrated in Figure 10.5.

Now we can make a correspondence between Figures 10.5 and 10.4. The **software system** block in Figure 10.5 can be viewed as the boot software running on a laptop, and the **environment** block in Figure 10.5 can be viewed as a user. The FSM shown in Figure 10.4 models the interactions shown by the bidirectional arrows in Figure 10.5.

An FSM model of the external behavior of a system describes the sequences of input to and expected output from the system. Such a model is a prime source of test cases. In this chapter, we explain how to derive test cases from an FSM model.

Figure 10.5 Interactions between system and its environment modeled as FSM.

10.2 POINTS OF CONTROL AND OBSERVATION

A *point of control and observation* (PCO) is a well-designated point of interaction between a system and its *users*. We use the term *users* in a broad sense to include all entities, including human users and other software and hardware systems, lying outside but interacting with the system under consideration. PCOs have the following characteristics:

- A PCO represents a point where a system receives input from its users and/or produces output for the users.
- There may be multiple PCOs between a system and its users.
- Even if a system under test (SUT) is a software system, for a human user a PCO may be "nearer" to the user than to the software under test. For example, a user may interact with a system via a push button, a touch screen, and so on. We want to emphasize that even if we have a software SUT, we may not have a keyboard and a monitor for interacting with the system.
- In case a PCO is a *physical entity*, such as a push button, a keyboard, or a speaker, there is a need to find their computer representations so that test cases can be automatically executed.

Example. Assume that we have a software system controlling a telephone switch PBX to provide connections between users. The SUT and the users interact via the different subsystems of a telephone. We show the user–interface details of a basic telephone to explain the concept of a PCO (Figure 10.6) and summarize those details in Table 10.1.

Figure 10.6 PCOs on a telephone.

TABLE 10.1 PCOs for Testing Telephone PBX

PCO	In/Out View of System	Description
Hook	In	The system receives off-hook and on-hook events.
Keypad	In	The caller dials a number and provides other control input.
Ring indicator	Out	The callee receives ring indication.
Speaker	Out	The caller receives tones (dial, fast busy, slow busy, etc.) and voice.
Mouthpiece	In	The caller produces voice input.

Figure 10.7 FSM model of a PBX.

The reader may notice that even for a simple device such as a telephone we have *five* distinct PCOs via which a user interacts with the switching software. In real life, users interact with the switching software via these distinct PCOs, and automated test execution systems must recognize those distinct PCOs. However, to make our discussion of test case generation from FSM models simple, clear, and concise, we use fewer PCOs. We designate all the PCOs on a *local phone* by LP and all the PCOs on a *remote phone* by RP (Figure 10.7).

10.3 FINITE-STATE MACHINE

A FSM M is defined as a tuple as follows: $M = <S, I, O, s_0, \delta, \lambda>$, where

S is a set of states,

I is a set of inputs,

O is a set of outputs,

s_0 is the initial state,

$\delta : S \times I \rightarrow S$ is a next-state function, and

$\lambda : S \times I \rightarrow O$ is an output function.

Note the following points related to inputs and outputs because of the importance of the concept of *observation* in testing a system:

- Identify the inputs and the outputs which are observed by explicitly specifying a set of PCOs. For each state transition, specify the PCO at which the input occurs and the output is observed.
- There may be many outputs occurring at different PCOs for a single input in a state.

An FSM specification of the interactions between a user and a PBX system is shown in Figure 10.8. The FSM has nine distinct states as explained in Table 10.2.

Figure 10.8 FSM model of PBX.

TABLE 10.2 Set of States in FSM of Figure 10.8

Abbreviation	Expanded Form	Meaning
OH	On hook	A phone is on hook.
AD	Add digit	The user is dialing a number.
SB	Slow busy	The system has produced a slow busy tone.
FB	Slow busy	The system has produced a fast busy tone.
RNG	Ring	The remote phone is ringing.
TK	Talk	A connection is established.
LON	Local on hook	The local phone is on hook.
RON	Remote on hook	The remote phone is on hook.
IAF	Idle after Fast busy	The local phone is idle after a fast busy.

TABLE 10.3 Input and Output Sets in FSM of Figure 10.8

Input	Output
OFH: Off hook	DT: Dial tone
ONH: On hook	RING: Phone ringing
#1: Valid phone number	RT: Ring tone
#2: Invalid phone number	SBT: Slow busy tone
NOI: No input	FBT: Fast busy tone
	IT: Idle tone
	—: Don't care

The initial state of the FSM is OH, which appears twice in Figure 10.8, because we wanted to avoid drawing transition lines from states LON, RON, and IAF back to the first occurrence of OH at the top.

There are five distinct input symbols (Table 10.3) accepted by the FSM. NOI represents user inaction; that is, the user never provides an input in a certain state. We have introduced the concept of an explicit NOI because we want to describe the behavior of a system without introducing internal events, such as timeouts. There are seven output symbols, one of which denotes a *don't care* output. A don't care output is an absence of output or an arbitrary output which is ignored by the user.

There are two abstract PCOs used in the FSM of Figure 10.8. These are called LP and RP to represent a local phone used by a caller and a remote phone represented by the callee, respectively. We call LP and RP abstract PCOs because each LP and RP represents *five* real, distinct PCOs, as explained in Section 10.2.

The input and output parts of a state transition are represented as follows:

$$\frac{\mathrm{PCO}_i : a}{\mathrm{PCO}_j : b}$$

where input a occurs at PCO_i and output b occurs at PCO_j. If a state transition produces multiple outputs, we use the notation

$$\frac{\text{PCO}_i : a}{\{\text{PCO}_j : b, \text{PCO}_k : c\}}$$

where input a occurs at PCO_i, output b occurs at PCO_j, and output c occurs at PCO_k. We will represent a complete transition using the following syntax:

<present state, input, output, next state> or <present state, input/output, next state>

The state transition <OH, LP: OFH, LP: DT, AD> means that if the FSM is in state OH and receives input OFH (off hook) at port (PCO) LP, it produces output DT (dial tone) at the same port LP and moves to state AD (Figure 10.8).

10.4 TEST GENERATION FROM AN FSM

Given an FSM model M of the requirements of a system and an implementation I_M of M, the immediate testing task is to confirm that the implementation I_M behaves as prescribed by M. The testing process that verifies that an implementation conforms to its specification is called *conformance testing*. The basic idea in conformance testing is summarized as follows:

- Obtain sequences of state transitions from M.
- Turn each sequence of a state transition into a test sequence.
- Test I_M with a set of test sequences and observe whether or not I_M possesses the corresponding sequences of state transitions.
- The conformance of I_M with M can be verified by carefully choosing enough state transition sequences from M.

In the following sections, first, we explain the ways to turn a state transition sequence into a test sequence. Next, we explain the process of selecting different state transition sequences.

10.5 TRANSITION TOUR METHOD

In this section, we discuss a process to generate a *test sequence* or *test case* T_c from a *state transition sequence* S_t of a given FSM M. Specifically, we consider *transition tours*, where a transition tour is a sequence of state transitions beginning and ending at the initial state. Naito and Tsunoyama [1] introduced the transition tour method for generating test cases from FSM specifications of sequential circuits. Sarikaya and Bochmann [2] were the first to observe that the transition tour method can be applied to protocol testing. An example of a transition tour obtained from

Figure 10.9 Interaction of test sequence with SUT.

Figure 10.8 is as follows: <OH, LP: OFH, LP: DT, AD>, <AD, LP: ONH, LP: —, OH>. One can easily identify the *state*, *input*, and *expected output* components in the sequence of Figure 10.8. However, it may be noted that a test case is not merely a sequence of pairs of <input, expected output>, but rather a complete test case must contain additional behavior such that it can be executed autonomously even if the SUT contains faults. A *test system* interacting with a SUT is shown in Figure 10.9. A test system consists of a set of *test cases* and a test case *scheduler*. The scheduler decides the test case to be executed next depending on the *test case dependency* constraints specified by a test designer. A test case in execution produces inputs for the SUT and receives outputs from the SUT. It is obvious that a faulty SUT may produce an output which is different from the expected output, and sometimes it may not produce any output at all. Therefore, the test system must be able to handle these exceptional cases in addition to the normal cases. This idea leads us to the following formalization of a process for designing a complete test case:

- A test case contains a sequence of input and expected output data. This information is derived from the FSM specification of the SUT.
- A test case must be prepared to receive *unexpected* outputs from the SUT.
- A test case must not wait indefinitely to receive an output—expected or unexpected.

Example: Transition Tour. Let us derive a test case from the state transition sequence <OH, LP: OFH, LP: DT, AD>, <AD, LP: ONH, LP: —, OH>. It is useful to refer to a PCO of Figure 10.9, which explains an input–output relationship between the test system and a SUT. A sequence of inputs to the SUT can be obtained from its FSM model. For instance, the state transition sequence contains the input sequence {OFH, ONH}. Therefore, the test system must produce an output sequence {OFH, ONH} for the SUT. Therefore, an input in a state transition sequence of an FSM is an output of the test system at the same PCO. An output is represented by prefixing an *exclamation* mark ('!') to an event (or message)

```
1     LP !OFH
2       START(TIMER1, d1)
3         LP ?DT                    PASS
4            CANCEL(TIMER1)
5              LP !ONH
6         LP ?OTHERWISE             FAIL
7            CANCEL(TIMER1)
8              LP !ONH
9         ?TIMER1                   FAIL
10           CANCEL(TIMER1)
11             LP !ONH
```

Figure 10.10 Derived test case from transition tour.

in Figure 10.10. In line 1 of Figure 10.10, LP !OFH means that the test system outputs an event OFH at PCO LP.

An output produced in a state transition of an FMS M is interpreted as an *expected* output of an implementation I_M. Sometimes a faulty implementation may produce unexpected outputs. An output of an SUT becomes an input to the test system. Therefore, an output in a state transition sequence of an FSM is an input to the test system at the same PCO. In Figure 10.10, an input is represented by prefixing a *question* mark ('?') to an event (or message). Therefore, in line 3 of Figure 10.10, LP ?DT means that the test system is ready to receive the input DT at PCO LP.

Here the test system expects to receive an input DT at PCO LP, which has been specified as LP ?DT. However, a faulty SUT may produce an unexpected output instead of the expected output DT at PCO LP. In line 6, the test system is ready to receive any event *other* than a DT at PCO LP. The reader may notice that LP ?DT in line 3 and LP ?OTHERWISE in line 6 appear at the same level of indentation and both lines have the same immediate predecessor action START(TIMER1, d1) in line 2.

If an SUT fails to produce any output—expected or unexpected—then the test system and the SUT will be deadlocked, which is prevented by including a timeout mechanism in the test system. Before a test system starts to wait for an input, it starts a timer of certain duration as shown in line 2. The name of the timer is TIMER1 in line 2, and its timeout duration is d1. If the SUT fails to produce the expected output DT or any other output within an interval of d1 after receiving input OFH at PCO LP, the test system will produce an internal timeout event called TIMER1, which will be received in line 9. One of the events specified in lines 3, 6, and 9 eventually occurs. This means that the test system is not deadlocked in the presence of a faulty SUT.

Coverage Metrics for Selecting Transition Tours. One can design one test case from one transition tour. One transition tour may not be sufficient to cover an entire FSM, unless it is a long one. Considering the imperative to simplify test design and to test just a small portion of an implementation with one test case,

there is a need to design many test cases. A perpetual question is: How many test cases should one design? Therefore, there is a need to identify several transition tours from an FSM. The concept of *coverage metrics* is used in selecting a set of transition tours. In order to test FSM-based implementations, two commonly used coverage metrics are:

- State coverage
- Transition coverage

Transition Tours for State Coverage. We select a set of transition tours so that every state of an FSM is visited at least once to achieve this coverage criterion. We have identified three transition tours, as shown in Table 10.4, to cover all the states of the FSM shown in Figure 10.8. One can easily obtain test sequences from the three transition tours in Table 10.4 following the design principles explained in Section 10.5 to transform a transition tour into a test sequence. State coverage is the weakest among all the selection criteria used to generate test sequences from FSMs. The three transition tours shown in Table 10.4 cover every state of the FSM shown in Figure 10.8 at least once. However, of the 21 state transitions, only 11 are covered by the three transition tours. The 10 state transitions which have not been covered by those three transition tours are listed in Table 10.5. We next consider a stronger form of coverage criterion, namely, *transition coverage*.

Transition Tours for Transition Coverage. We select a set of transition tours so that every state transition of an FSM is visited at least once to achieve this coverage criterion. We have identified nine transition tours, as shown in Table 10.6, to cover all the state transitions of the FSM shown in Figure 10.8. One can easily obtain test sequences from the nine transition tours in Table 10.6 following the design principles explained in Section 10.5 to transform a transition tour into a test sequence.

TABLE 10.4 Transition Tours Covering All States in Figure 10.8

Serial Number	Transition Tours	States Visited
1	<OH, LP:OFH/LP:DT, AD>; <AD, LP:#2/LP:SBT, SB>; <SB, LP:NOI/LP:FBT, FB>; <FB, LP:NOI/LP:IT, IAF>; <IAF, LP:ONH/—, OH>	OH, AD, SB, FB, IAF
2	<OH, LP:OFH/LP:DT, AD>; <AD, LP:#1/{LP:RT, RP:RING}, RNG>; <RNG, RP:OFH/{LP:IT, RP:IT}, TK>; <TK, LP:ONH/RP:IT, LON>; <LON, LP:NOI/—, OH>	OH, AD, RNG, TK, LON
3	<OH, LP:OFH/LP:DT, AD>; <AD, LP:#1/{LP:RT, RP:RING}, RNG>; <RNG, RP:OFH/{LP:IT, RP:IT}, TK>; <TK, RP:ONH/LP:IT, RON>; <RON, LP:ONH/—, OH>	OH, AD, RNG, TK, RON

TABLE 10.5 State Transitions Not Covered by Transition Tours of Table 10.4

Serial Number	State Transitions
1	<AD, LP:ONH/—, OH>
2	<AD, LP:NOI/LP:FBT, FB>
3	<SB, LP:ONH/—, OH>
4	<FB, LP:ONH/—, OH>
5	<LON, LP:OFH/LP:IT, TK>
6	<LON, RP:ONH/—, OH>
7	<RON, RP:OFH/—, TK>
8	<RON, LP:NOI/LP:DT, AD>
9	<RNG, LP:ONH/—, OH>
10	<RNG, RP:NOI/FBT, FB>

TABLE 10.6 Transition Tours Covering All State Transitions in Figure 10.8

Serial Number	Transition Tours
1	<OH, LP:OFH/LP:DT, AD>; <AD, LP:#2/LP:SBT, SB>; <SB, LP:NOI/LP:FBT, FB>; <FB, LP:NOI/LP:IT, IAF>; <IAF, LP:ONH/—, OH>
2	<OH, LP:OFH/LP:DT, AD>; <AD, LP:#1/\{LP:RT, RP:RING\}, RNG>; <RNG, RP:OFH/{LP:IT, RP:IT}, TK>; <TK, LP:ONH/RP:IT, LON>; <LON, LP:NOI/—, OH>
3	<OH, LP:OFH/LP:DT, AD>; <AD, LP:#1/{LP:RT, RP:RING}, RNG>; <RNG, RP:OFH/{LP:IT, RP:IT}, TK>; <TK, RP:ONH/LP:IT, RON>; <RON, LP:ONH/—, OH>
4	<OH, LP:OFH/LP:DT, AD>; <AD, LP:#2/LP:SBT, SB>; <SB, LP:ONH/—, OH>;
5	<OH, LP:OFH/LP:DT, AD>; <AD, LP:NOI/LP:FBT, FB>; <FB, LP:ONH/—, OH>
6	<OH, LP:OFH/LP:DT, AD>; <AD, LP:#1/{LP:RT, RP:RING}, RNG>; <RNG, RP:OFH/{LP:IT, RP:IT}, TK>; <TK, LP:ONH/RP:IT, LON>; <LON, LP:OFH/LP:IT, TK>; <TK, RP:ONH/LP:IT, RON>; <RON, RP:OFH/—, TK>; <TK, LP:ONH/RP:IT, LON>; <LON, LP:NOI/—, OH>
7	<OH, LP:OFH/LP:DT, AD>; <AD, LP:#1/{LP:RT, RP:RING}, RNG>; <RNG, RP:OFH/{LP:IT, RP:IT}, TK>; <TK, RP:ONH/LP:IT, RON>; <RON, LP:NOI/LP:DT, AD>; <AD, LP:ONH/LP:DT, OH>
8	<OH, LP:OFH/LP:DT, AD>; <AD, LP:#1/{LP:RT, RP:RING}, RNG>; <RNG, LP:ONH/—, OH>
9	<OH, LP:OFH/LP:DT, AD>; <AD, LP:#1/{LP:RT, RP:RING}, RNG>; <RNG, RP:NOI/FBT, FB>; <FB, LP:ONH/—, OH>

10.6 TESTING WITH STATE VERIFICATION

There are two *functions* associated with a state transition, namely, an *output* function (λ) and a *next-state* function (δ). Test cases generated using the transition tour method discussed in Section 10.5 focused on the outputs. Now we discuss a method

Figure 10.11 Conceptual model of test case with state verification.

for generating test cases by putting emphasis on both the output and the next state of every state transition of an FSM. It is easy to verify outputs since they appear at PCOs, which are external to a system under test. However, verification of the next state of a state transition is not an easy task because the concept of *state* is purely internal to a SUT. The next state of a state transition is verified by applying further inputs to an SUT and observing its response at the PCOs. A conceptual model of a method to generate test cases from an FSM with both output and state verifications is illustrated in Figure 10.11. The five steps of the method are explained in the following. The method is explained from the standpoint of testing a state transition from state s_i to s_j with input a.

Methodology for Testing with State Verification

Step 1: Assuming that the FSM is in its initial state, move the FSM from the initial state s_0 to state s_i by applying a sequence of inputs called a *transfer sequence* denoted by $T(s_i)$. It may be noted that different states will have different transfer sequences, that is, $T(s_i) \neq T(s_j)$ for $i \neq j$. For state s_i, $T(s_i)$ can be obtained from the FSM. At the end of this step, the FSM is in state s_i.

Step 2: In this step we apply input a to the SUT and observe its actual output, which is compared with the expected output b of the FSM. At the end of this step, a correctly implemented state transition takes the SUT to its new state s_j. However, a faulty implementation can potentially take it to a state different from s_j. The new state of the SUT is verified in the following step.

Step 3: Apply a verification sequence VER_j to the SUT and observe the corresponding output sequence. An important property of VER_j is that $\lambda(s_j, \text{VER}_j) \neq \lambda(s_\ell, \text{VER}_j)\ \forall s_\ell$ and $s_\ell \neq s_j$. At the end of this step, the SUT is in state s_k.

Step 4: Move the SUT back to the initial state s_0 by applying a *reset* sequence RI. It is assumed that an SUT has correctly implemented a reset mechanism.

Step 5: Repeat steps 1–4 for all state transitions in the given FSM.

For a selected transition from state s_i to state s_j with input a, the above four steps induce a transition tour defined by the *input sequence* $T(s_i)@a@\text{VER}_j@\text{RI}$ applied to the system in its initial state s_0. The symbol '@' represents *concatenation* of two sequences. Applying the test design principles discussed in Section 10.5, one can derive a test case from such transition tours. Identifying a transfer sequence $T(s_i)$ out of the input sequence $T(s_i)@a@\text{VER}_j@\text{RI}$ for state s_i is a straightforward task. However, it is not trivial to verify the next state of an implementation. There are three kinds of commonly used input sequences to verify the next state of a SUT. These input sequences are as follows:

- Unique input–output sequence
- Distinguishing sequence
- Characterizing sequence

In the following sections, we explain the meanings of the three kinds of input sequences and the ways to generate those kinds of sequences.

10.7 UNIQUE INPUT–OUTPUT SEQUENCE

We define a *unique input–output* (UIO) sequence and explain an algorithm to compute UIO sequences from an FSM $M = <S, I, O, s_0, \delta, \lambda>$, where S, I, O, s_0, δ, and λ have been explained in Section 10.3. First, we extend the semantics of δ and λ as follows:

- We extend the domain of λ and δ to include strings of input symbols. For instance, for state s_0 and input sequence $x = a_1, \ldots, a_k$, the corresponding output sequence is denoted by $\lambda(s_0, x) = b_1, \ldots, b_k$, where $b_i = \lambda(s_{i-1}, a_i)$ and $s_i = \delta(s_{i-1}, a_i)$ for $i = 1, \ldots, k$ and the final state is $\delta(s_0, x) = s_k$.
- Similarly, we extend the domain and range of the transition and output functions to include sets of states. For example, if Q is a set of states and x is an input *sequence*, then $\delta(Q, x) = \{\delta(s, x) | s \in Q\}$.
- If x and y are two input strings, the notation $x@y$ denotes *concatenation* of the two strings in that order.

Next, we define four properties of FSMs in the following. These properties are essential to the computation of UIO sequences.

Completely Specified: An FSM M is said to be *completely specified* if for each input $a \in I$ there is a state transition defined at each state of M.

Deterministic: An FSM M is said to be *deterministic* if (i) for each input $a \in I$ there is at most one state transition defined at each state of M and (ii) there is no *internal* event causing state transitions in the FSM. A timeout is an example of internal events that can cause a state transition.

Reduced: An FSM M is said to be *reduced* if for any pair of states s_i and s_j, $i \neq j$, there is an input sequence y such that $\lambda(s_i, y) \neq \lambda(s_j, y)$.

Intuitively, an FSM is reduced if no two states are equivalent, that is, there exists an input sequence that distinguishes one state from the other.

Strongly Connected: An FSM M is said to be *strongly connected* if any state is reachable from any other state.

An input–output sequence $y/\lambda(s_i, y)$ is said to be a UIO sequence for state s_i if and only if $y/\lambda(s_i, y) \neq y/\lambda(s_j, y)$, $i \neq j$, $\forall s_j$ in M. Hsieh [3] introduced the concept of simple input–output sequence in 1971 to identify FSMs. Sabnani and Dahbura [4] were the first to coin the term UIO sequences and applied the concept to testing communication protocols. In the following, an efficient algorithm to compute UIO sequences is explained [5].

Given an FSM M, a *path vector* is a collection of state pairs $(s_1/s_1', \ldots, s_i/s_i', \ldots, s_k/s_k')$ with the following two properties: (i) s_i and s_i' denote the head and tail states, respectively, of a path, where a path is a sequence of state transitions; (ii) an identical input–output sequence is associated with all the paths in the path vector. Given a path vector $\mathrm{PV} = (s_1/s_1', \ldots, s_i/s_i', \ldots, s_k/s_k')$, the *initial vector* (IV) is an ordered collection of head states of PV, that is, $\mathrm{IV(PV)} = (s_1, \ldots, s_i, \ldots, s_k)$. Similarly, the *current vector* (CV) is the ordered collection of tail states of PV, that is, $\mathrm{CV(PV)} = (s_1', \ldots, s_i', \ldots, s_k')$. A path vector is said to be a *singleton* vector if it contains exactly one state pair. A path vector is said to be a *homogeneous* vector if all members of CV(PV) are identical. It may be noted that a singleton vector is also a homogeneous vector. For an n-state FSM, we define a unique *initial* path vector $(s_1/s_1, \ldots, s_i/s_i, \ldots, s_n/s_n)$ such that a *null* path is associated with all state pairs.

Given a path vector and the input–output label a/b of a transition, *vector perturbation* means computing a new path vector PV′ from PV and a/b. Given $\mathrm{PV} = (s_1/s_1', \ldots, s_i/s_i', \ldots, s_k/s_k')$ and a transition label a/b, *perturbation* of PV with respect to edge label a/b, denoted by $\mathrm{PV}' = \mathrm{pert}(\mathrm{PV}, a/b)$, is defined as $\mathrm{PV}' = \{s_i/s_i'' | s_i'' = \delta(s_i', a) \wedge \lambda(s_i', a) = b \wedge s_i/s_i' \in \mathrm{PV}\}$. We can infinitely perturb all the path vectors for all transition labels given a reduced FSM and its initial path vector. One can imagine the perturbation function $\mathrm{PV}' = \mathrm{pert}(\mathrm{PV}, a/b)$ as an edge from a node PV to a new node PV′ with edge label a/b. In addition, given PV and a set of transition labels L, we can arrange the new $|L|$ nodes $\{\mathrm{pert}(\mathrm{PV}, a/b)\ \forall a/b \in L\}$ on one level. That is, all the path vectors of a given FSM can be arranged in the form of a tree with successive levels $1, 2, \ldots, \infty$. Such a tree is called a *UIO tree*. It may be noted that graphically a path vector is represented by two rows of states—the top row denotes IV(PV) and the bottom row denotes CV(PV). Theoretically, a UIO tree is a tree with infinite levels. However, we need to prune the tree based on some conditions, called *pruning* conditions. After each perturbation $\mathrm{PV}' = \mathrm{pert}(\mathrm{PV}, a/b)$, we check the following conditions:

C1: CV(PV′) is a homogeneous vector or a singleton vector.

C2: On the path from the initial vector to PV, there exists PV″ such that $\mathrm{PV}' \subseteq \mathrm{PV}''$.

While constructing a UIO tree, if one of the pruning conditions is satisfied, we declare PV′ to be a *terminal* path vector. Given a UIO tree and a state s_i of an FSM, s_i has a UIO sequence if and only if the UIO tree obtained from M has a singleton path vector ψ such that $v_i = IV(\psi)$, that is, v_i is the initial vector of the singleton ψ. The input–output sequence associated with the edges of the path from the initial path vector to a singleton path vector is a UIO sequence of the state found in the initial vector of the singleton. In the following, we present an algorithm to compute a finite UIO tree from an FSM.

Algorithm. Generation of UIO Tree

Input: $M = <S, I, O, \delta, \lambda>$ and L.

Output: UIO tree.

Method: Execute the following steps:

Step 1: Let Ψ be the set of path vectors in the UIO tree. Initially, Ψ contains the initial vector marked as *nonterminal*.

Step 2: Find a nonterminal member $\psi \in \Psi$ which has not been perturbed. If no such member exists, then the algorithm terminates.

Step 3: Compute $\psi' = \text{pert}(\psi, a_i/b_i)$ and add ψ' to Ψ $\forall a_i/b_i \in L$. Mark ψ to be perturbed and update the UIO tree.

Step 4: If $\text{pert}(\psi, a_i/b_i)$, computed in **step 3**, satisfies termination condition **C1** or **C2**, then mark ψ' as a terminal node.

Step 5: Go to **step 2**.

Example. Consider the FSM $G_1 = <S, I, O, A, \delta, \lambda>$ of Figure 10.12, where $S = \{A, B, C, D\}$ is the set of states, $I = \{0, 1\}$ is the set of inputs, $O = \{0, 1\}$ is the set of outputs, A is the initial state, $\delta : S \times I \rightarrow S$ is the next-state function, and $\lambda : S \times I \rightarrow O$ is the output function. The set of distinct transition labels is given by $L = \{0/0, 0/1, 1/0\}$. The initial path vector is given by

Figure 10.12 Finite-state machine G_1 (From ref. 5. © 1997 IEEE.)

$\psi_1 = (A/A, B/B, C/C, D/D)$ and is perturbed by using all members of L as follows:

$$(A/B, B/A) = \text{pert}(\psi_1, 0/0)$$
$$(C/D, D/D) = \text{pert}(\psi_1, 0/1)$$
$$(A/D, B/B, C/A, D/C) = \text{pert}(\psi_1, 1/0)$$

Now, we represent the UIO tree in graphical form. We put three edges from the initial path vector $\psi_1 = (A/A, B/B, C/C, D/D)$ to path vectors $(A/B, B/A)$, $(C/D, D/D)$, and $(A/D, B/B, C/A, D/C)$ with transition labels 0/0, 0/1, and 1/0, respectively, as shown in Figure 10.13. The reader may recall that the new

Figure 10.13 UIO tree for G_1 in Figure 10.12. (From ref. 5. © 1997 IEEE.)

path vector $(C/D, D/D)$ is a terminal node because its current vector (D, D) is a homogeneous vector. Therefore, path vector $(C/D, D/D)$ is no more perturbed, whereas the other two path vectors $(A/B, B/A)$ and $(A/D, B/B, C/A, D/C)$ are further perturbed, as shown in Figure 10.13. A complete UIO tree is shown in Figure 10.13.

Figure 10.13 is redrawn in the form of Figure 10.14 by highlighting a UIO sequence of minimal length for each state. We have identified four singleton path vectors, namely, (A/A), (B/D), (C/A), and (D/A), and highlighted the corresponding paths leading up to them from the initial path vector. We show the UIO sequences of the four states in Table 10.7.

Figure 10.14 Identification of UIO sequences on UIO tree of Figure 10.13.

TABLE 10.7 UIO Sequences of Minimal Lengths Obtained from Figure 10.14

State	Input Sequence	Output Sequence
A	010	000
B	010	001
C	1010	0000
D	11010	00000

10.8 DISTINGUISHING SEQUENCE

Given an FSM $M = <S, I, O, s_0, \delta, \lambda>$, an input sequence x is said to be a distinguishing sequence if and only if $\lambda(s_i, x) \neq \lambda(s_j, x)$ $\forall s_i, s_j \in S$, and $s_i \neq s_j$. Therefore, every state produces a unique output sequence in response to a distinguishing input sequence. The concept of distinguishing sequences was studied to address the *machine identification* problem in the early days of computing [6–9]. The machine identification problem is as follows: Given an implementation of an FSM in the form of a black box and the number of states of the FSM, derive the state table (equivalently, a state machine) which accurately represents the behavior of the black box. In contrast, the objective of a checking experiment is to decide whether a given black-box implementation of an FSM behaves according to a given state table representation of the FSM.

Given an FSM $M = <S, I, O, s_0, \delta, \lambda>$ with $|S| = n$, a *state block* is defined as a set $\mathcal{B}$ of multisets of S such that the sum of the cardinalities of the multisets of $\mathcal{B}$ is equal to n. Thus, a state block may contain just one multiset with all the states of M. Also, a state block may contain n multisets with exactly one element in each of them. Assuming that $S = \{A, B, C, D\}$ some examples of state blocks are given in Table 10.8.

Given a state block $\mathcal{B} = \{W_1, \ldots, W_i, \ldots, W_m\}$ and an input symbol $a \in I$, we define a function $\mathcal{B}' = \text{dpert}(\mathcal{B}, a)$ such that for each multiset member $W_i = (w_{i1}, w_{i2}, \ldots, w_{ik})$ of $\mathcal{B}$ we obtain one or more members of $\mathcal{B}'$ as follows:

- If two states of W_i produce *different* outputs in response to input a, then their next states are put in different multisets of $\mathcal{B}'$.
- The next states of all those states in W_i which produce the same output in response to input a are put in the same multiset of $\mathcal{B}'$.

TABLE 10.8 Examples of State Blocks

State block 1	$\{(ABCD)\}$
State block 2	$\{(AB), (CC)\}$
State block 3	$\{(AB), (CD)\}$
State block 4	$\{(A), (B), (C), (D)\}$
State block 5	$\{(A), (A), (B), (C)\}$

Given a reduced FSM, the initial state block consists of just one element containing the set of states of the FSM. Next, we can infinitely perturb all the state blocks for all the input symbols using the dpert*()* function. One can view the perturbation function $\mathcal{B}' = \text{dpert}(\mathcal{B}, a)$ as an edge from a node $\mathcal{B}$ to a new node $\mathcal{B}'$ with edge label a. Given a state block and the set of inputs I, we can arrange the new $|I|$ nodes $\{\text{dpert}(\mathcal{B}, a), \forall a \in I\}$ at the same level. All the state blocks of a given FSM can be arranged in the form of a tree with successive levels $1, 2, \ldots, \infty$. Such a tree is called a *DS tree*. Theoretically, a distinguishing sequence (DS) tree is a tree with infinite levels. However, a finite-level tree would serve our purpose. A finite-level tree is obtained by pruning the tree based on some conditions, called *pruning conditions*, defined using the following terms:

D1: A state block $\mathcal{B}$ is a homogeneous state block if at least one multiset member W_i of $\mathcal{B}$ has repeated states.

D2: A state block $\mathcal{B}$ is a singleton state block if all elements of $\mathcal{B}$ have exactly one state member.

D3: On the path from the initial state block to the current state block $\mathcal{B}$, there exists a state block $\mathcal{B}$.

While constructing a DS tree, if one of the above three pruning conditions is satisfied, we declare state block $\mathcal{B}$ to be a terminal state block. Given a DS tree of an FSM M, M has a distinguishing sequence if and only if there exists a singleton state block in the DS tree. The sequence of inputs from the initial state block to a singleton state block is a DS of machine M. In the following, we present an algorithm to compute a finite DS tree from an FSM.

Algorithm. Generation of DS Tree

Input: $M = <S, I, O, s_0, \delta, \lambda>$.

Output: DS tree.

Method: Execute the following steps:

Step 1: Let Ψ be the set of state blocks in the DS tree. Initially, Ψ contains the initial state block marked as nonterminal.

Step 2: Find a nonterminal member $\psi \in \Psi$ which has not been perturbed. If no such member exists, then the algorithm terminates.

Step 3: Compute $\psi' = \text{dpert}(\psi, a)$ and add ψ' to Ψ $\forall a \in I$. Mark ψ to be perturbed and update the DS tree.

Step 4: If $\text{dpert}(\psi, a)$, computed in **step 3**, satisfies the termination condition **D1**, **D2**, or **D3**, then mark ψ' as a terminal node.

Step 5: Go to **step 2**.

Example. Let us consider an FSM $G_2 = <S, I, O, A, \delta, \lambda>$, shown in Figure 10.15, where $S = \{A, B, C, D\}$ is the set of states, $I = \{0, 1\}$ is the set of inputs, $O = \{0, 1\}$ is the set of outputs, A is the initial state, $\delta : S \times I \rightarrow S$

Figure 10.15 Finite-state machine G_2.

Figure 10.16 Distinguishing sequence tree for G_2 in Figure 10.15.

is the next-state function, and $\lambda : S \times I \rightarrow O$ is the output function. The DS tree for G_2 is shown in Figure 10.16. The initial state block contains just one multiset $\{A, B, C, D\}$, which has been represented as the initial node $(ABCD)$. We have perturbed the initial node $(ABCD)$ with the members of the input set $I = \{0, 1\}$. Perturbation of the initial state block with input 0 produces the state block $(AB)(DD)$ because of the following reasons:

- States A and B produce the same output 0 with input 0 and move the machine to states B and A, respectively.
- States C and D produce the same output 1 with input 0 and move the FSM to the same state D.

Since $(AB)(DD)$ is a homogeneous state block, it is a terminal node in the DS tree, and, thus, it is no more perturbed. Perturbation of the initial state block with input 1 produces the state block $(DC)(BA)$ because of the following reasons:

- States A and D produce the same output 0 with input 1 and move the machine to states D and C, respectively.
- States B and C produce the same output 1 with input 1 and move the machine to states B and A, respectively.

We obtain a homogeneous state block $(DD)(BA)$ by perturbing the state block $(DC)(BA)$ with input 0 and we obtain a singleton state block $(A)(C)(D)(B)$ by perturbing state block $(DC)(BA)$ with input 1. The complete DS tree is shown

TABLE 10.9 Outputs of FSM G_2 in Response to Input Sequence 11 in Different States

Present State	Output Sequence
A	00
B	11
C	10
D	01

in Figure 10.16. Therefore, the input sequence 11 that takes the DS tree from its initial state block to its only singleton state block is a DS for FSM G_2. The output sequences of FSM G_2 in Figure 10.15 in response to the distinguishing input sequence 11 in all the four different states are shown in Table 10.9.

10.9 CHARACTERIZING SEQUENCE

It is still possible to determine uniquely the state of an FSM for FSMs which do not possess a DS. The FSM shown in Figure 10.17 does not have a DS because there is no singleton state block in the DS tree, as shown in Figure 10.18. The *W-method* was introduced for FSMs that do not possess a DS [9, 10]. A *characterizing set* of a state s_i is a set of input sequences such that, when each sequence is applied to the implementation at state s_i, the set of output sequences generated by the implementation uniquely identifies state s_i. Each sequence of the characterizing set of state s_i distinguishes state s_i from a group of states. Therefore, applying all of the sequences in the characterizing set distinguishes state s_i from all other state. For an FSM-based specification, a set that consists of characterizing sets of every state is called the W-set $= \{W_1, W_2, \ldots, W_p\}$ of the FSM. The members of the W-set are called *characterizing sequences* of the given FSM.

The basic test procedure for testing a state transition $(s_i, s_j, a/b)$ using the W-method follows.

Figure 10.17 FSM that does not possess distinguishing sequence. (From ref. 11. © 1994 IEEE.)

Figure 10.18 DS tree for FSM (Figure 10.17).

Testing Transition (s_i, s_j, a/b) Using W-Method Repeat the following steps for each input sequence of the W-set:

Step 1: Assuming that the SUT is in its initial state, bring the SUT from its initial state s_0 to state s_i by applying a transfer sequence $T(s_i)$ as shown in Figure 10.11.

Step 2: Apply the input a and verify that the SUT generates the output b.

Step 3: Apply a verification sequence from the W-set to the SUT and verify that the corresponding output sequence is as expected. Assume that the SUT is in state s_k at the end of this step.

Step 4: Move the SUT back to the initial state s_0 by applying a *reset* sequence RI in state s_k.

Example: Characterizing Sequences. Consider the FSM specification $M = <S, I, O, A, \delta, \lambda>$, shown in Figure 10.17, where $S = \{A, B, C, D\}$ is the set of states, $I = \{a, b\}$ is the set of inputs, $O = \{x, y\}$ is the set of outputs, A is the initial state, $\delta : \times I \rightarrow S$ is the next-state function, and $\lambda : S \times I \rightarrow O$ is the output function. Kohavi [9] used *multiple experiments* to construct the W-set for this FSM. Consider the input sequence $W_1 = aba$. The output sequences generated by W_1 for each state of the FSM are shown in Table 10.10. The output sequence generated by the input sequence W_1 can identify whether the state of an SUT was either B or C before W_1 is applied. This is because state B leads to the output sequence yyy, whereas state C leads to the output sequence yyx. However, W_1 cannot identify the state of an SUT if the FSM is in A or D because the output sequences are xyx for both states, as shown in Table 10.10.

Now let us examine the response of the SUT to the input sequence $W_2 = ba$ for each state. The output sequences generated by W_2 for each state of the

TABLE 10.10 Output Sequences Generated by FSM of Figure 10.17 as Response to W_1

Starting States	Output Generated by $W_1 = aba$
A	xyx
B	yyy
C	yyx
D	xyx

TABLE 10.11 Output Sequences Generated by FSM of Figure 10.17 as Response to W_2

Starting States	Output Generated by $W_2 = ba$
A	yx
B	yx
C	yy
D	yy

FSM are shown in Table 10.11. The FSM implementation generates distinct output sequences as a response to W_2 if an SUT was at A or D, as shown in Table 10.11. This is because states A and D lead to distinct output sequences yx and yy, respectively. Therefore, the W-set for the FSM consists of two input sequences: W-set $= \{W_1, W_2\}$, where $W_1 = aba$ and $W_2 = ba$. The transfer sequences for all the states are $T(B) = bb$, $T(C) = ba$, and $T(D) = b$. The reset input sequence is RI $= bababa$. The input sequence for testing the state transition $(D, A, a/x)$ is given in Table 10.12. In Table 10.12, the columns labeled "message to SUT" and "message from SUT" represent the input message sent to the SUT and the expected output message generated by the SUT, respectively. The current state and the expected next state of the SUT are shown in the columns labeled "current state" and "next state," respectively. During testing, the inputs are applied to the SUT in the order denoted by the column "step." In the first step a transfer sequence is applied to bring the SUT to state D. In step 2, the transition is tested. Then $W_1 = aba$ is applied to verify the state (steps 3, 4, and 5). At this point, the state transition is only partially tested, since W_1 is not enough to identify the state of an implementation. The reset sequence RI $= bababa$ (steps 6–11) is applied followed by the transfer sequence of $T(D) = b$ (step 12) to bring the SUT into the initial state and into state D, respectively. The test is repeated for the same transition by using $W_2 = ba$ (steps 13–21). If all the outputs received from the SUT are defined by the FSM, the state transition test is completed successfully. If the output of the SUT is not the expected response at any step, an error is detected in the SUT.

TABLE 10.12 Test Sequences for State Transition (D, A, a/x) of FSM in Figure 10.17

Step	Current State	Next State	Message to SUT	Message from SUT
Apply $T(D)$				
1	A	D	b	y
Test Transition (D, A, a/x)				
2	D	A	a	x
Apply W_1				
3	A	A	a	x
4	A	D	b	y
5	D	A	a	x
Apply RI				
6	A	D	b	y
7	D	A	a	x
8	A	D	b	y
9	D	A	a	x
10	A	D	b	y
11	D	A	a	x
Apply $T(D)$				
12	A	D	b	y
Test Transition (D, A, a/x)				
13	D	A	a	x
Apply W_2				
14	A	D	b	y
15	D	A	a	x
Apply RI				
16	A	D	b	y
17	D	A	a	x
18	A	D	b	y
19	D	A	a	x
20	A	D	b	y
21	D	A	a	x

Source: From ref. 11.

Four major methods—*transition tours, distinguishing sequences, characterizing sequences, and unique input–output sequences*—are discussed for the generation of tests from an FSM. A question that naturally comes to mind is the effectiveness of these techniques, that is, the types of discrepancies detected by each of these methods. Sidhu and Leung [12] present a fault model based on the Monte Carlo simulation technique for estimating the fault coverage of the above four test generation methods. The authors introduced 10 different *classes* of randomly faulty specification, each obtained by random altering a given specification. For example, *class I* faults consist of randomly altering an output operation in a given specification. The authors conclude that all four methods, except for the transition tour method, can detect all single faults as opposed to several faults

introduced in a given specification. In addition, it is also shown that distinguishing, characterizing, and UIO sequences have the same fault detection capability. Another study, similar to the one by Sidhu and Leung, is reported by Dahbura and Sabnani for the UIO sequence method [13].

10.10 TEST ARCHITECTURES

An overview of four abstract test architectures developed by the ISO is presented in this section. The ISO documents [14–16] and Linn [17] and Rayner [18] provide more detail about the test architectures. The ISO test architectures are based on the Open System Interconnection (OSI) reference architecture, which consists of a hierarchical layer structure of entities. The purpose of an entity at layer N is to provide certain services, called N-services, to its upper layer entity. It uses the service provided by the $N-1$ layer while isolating the implementation details of the lower layer entities from the upper layers. Peer N entities communicate with each other through an $N-1$ service provider by exchanging N-protocol data units [(N)-PDUs], as shown in Figure 10.19. Interactions of an N-entity with its upper and lower entities are defined by N and $N-1$ abstract service primitives (ASPs).

The abstract test architectures are described in terms of the inputs to the IUT that can be given in a controlled manner and the corresponding outputs from the IUT are observed. Specifically, an abstract test architecture is described by identifying the points closest to the IUT where controls and observations are specifed. The abstract test architectures are classified into two major categories: *local* and *external*. The local test architectures are characterized by observation and control being specified in terms of events occurring within the SUT at the layer boundaries immediately below and above the IUT, as shown in Figure 10.20. On the other

Figure 10.19 Abstraction of N entity in OSI reference architecture.

Figure 10.20 Abstract local test architecture.

Figure 10.21 Abstract external test architecture.

hand, external test architectures are characterized by the observation and control of the events taking place externally from the SUT, on the other side of the underlying service provider from the IUT, as shown in Figure 10.21. The system in which an IUT resides is a SUT. The external architectures assume that an underlying communications service is used in testing. The local test architectures are only applicable to in-house testing by the suppliers. Whereas the external tests architectures are applied to both in-house testing by the supplier and the buyers/third-party test centers. There are three types of external test architectures, which are discussed along with the local test architecture, in the following.

10.10.1 Local Architecture

A basic assumption in the local architecture is that exposed interfaces above and below the IUT exist. These interfaces serve as PCOs, that is, points at which a real test system can provide inputs to and observe outputs from the IUT. The test architecture includes two logically distinct elements, called upper tester and lower tester, as shown in Figure 10.22. The test events are specified in terms of N ASPs above the IUT, and $N-1$ ASPs and N PDUs below the IUT. The coordination between the upper and the lower testers are provided by test coordination procedures during the testing.

In summary, the local test architecture comprises a test harness around the IUT, which coordinates the actions of lower and upper testers. The roles of the upper and lower testers are to stimulate the IUT by exchanging test events at the

Figure 10.22 Local architecture.

top and bottom interfaces of the IUT. The local architecture implicitly provides the capability to synchronize and control the upper and lower testers because both of them are elements of the same test harness.

10.10.2 Distributed Architecture

The distributed architecture is illustrated in Figure 10.23. This architecture is one of three external architectures and makes no assumptions about the existence of a PCO below the IUT. This test architecture defines the PCOs as being at the service boundaries above the IUT and at the opposite side of the $N-1$ service provider from the IUT.

Note that the lower tester and the IUT reside in two different systems. The lower tester and the IUT are connected by an underlying service which offers an $N-1$ service using lower layer protocol and the physical media connecting the two systems. The lower tester is obviously a peer entity of the IUT in Figure 10.23. The arrows between the IUT and the $N-1$ service provider are not real interfaces, just

Figure 10.23 Distributed architecture.

the conceptual flow of N PDUs. The test events are specified in terms of N ASPs above the IUT and $N-1$ ASPs and N PDUs remotely, as shown in Figure 10.23. Three important points should be kept in mind with this architecture:

- The lower tester and IUT are physically separated with the implication that they perhaps observe the same test event at different times.
- Delivery out of sequence, data corruption, and loss of data are possible because of the unreliable quality of the lower service provider.
- Synchronization and control (test coordinate procedures) between upper and lower testers are more difficult due to the distributed nature of the test system.

In summary, the distributed test architecture is a logical equivalent of the local architecture with the lower tester and the IUT interconnected by a communication service. However, the structure of the local architecture implicitly gives the capability to synchronize and control the upper and lower testers because they are elements of the same test harness. Thus, distributed architecture is not a functional equivalent to the local architecture.

10.10.3 Coordinated Architecture

The coordinated architecture is an enhancement of the distributed architecture. Control and observation of N ASPs are performed by a Test Management Protocol (TMP). There is just one PCO at the opposite side of the $N-1$ service provider from the IUT, as shown in Figure 10.24. Note that, even though a PCO appears between the upper tester and the IUT, it is optional; as a choice made by the implementer, the upper tester is perhaps integrated as part of the IUT. Two features that distinguish the coordinated architecture are as follows:

- No interface is exposed to the upper tester of an IUT (although this is not precluded).
- A standard TMP and TMP data units (TMPDUs) are used to communicate between the upper tester and the lower tester.

Figure 10.24 Coordinated architecture.

Figure 10.25 Remote architecture.

The lower tester is considered to be the master of the upper tester. The actions of the upper tester are controlled by the lower tester through a TMP. Test events are specified in terms of $N-1$ ASPs, N PDUs, and TMPDUs, as illustrated in Figure 10.24.

10.10.4 Remote Architecture

The remote architecture is applicable to IUTs that do not have an exposed upper interface. In the absence of an upper tester, the test architecture identifies a PCO away from the IUT, on the opposite side of the $N-1$ service provider. The test events are specified in terms of the $N-1$ ASPs and N PDUs, as shown in Figure 10.25. There are two major features of this architecture:

- No interface at the top of the IUT is assumed.
- No explicit test coordination procedure is used. The coordination between upper and lower testers is manual (e.g., talking over the telephone). The coordination is implicit in the PDUs initiated by the lower tester or provided by the actions taken by an upper layer entity to stimulate the IUT.

The architecture relies on the protocol being tested for synchronization between the lower tester and the IUT. Verdicts must be formulated based on the stimulus provided by the lower tester and the responses of the IUT as observed by the lower tester.

10.11 TESTING AND TEST CONTROL NOTATION VERSION 3 (TTCN-3)

As the name suggests, TTCN-3 is a language for specifying test cases [19]. The language is increasingly being accepted in the industry as a test specification language after it was standardized by the ETSI (European Telecommunication Standards

Institute). The language has been designed by keeping in mind the needs of testing complex telecommunication systems. Consequently, the language is being used to write test cases to test complex communication protocols, such as the Session Initiation Protocol (SIP) and the Internet Protocol version 6 (IPv6).

In the early efforts to develop a test specification language, the acronym TTCN-1 stood for *Tree and Tabular Combined Notation version 1*. The TCCN was developed in the mid-1980s and it evolved from TTCN-1 to TTCN-2 (Tree and Tabular Combined Notation version 2) while still retaining its core syntax and semantics to a large extent. Though much effort went into the development of the TCCN, it was not widely accepted as a test specification language in the industry. The reason for an absence of a broad interest in the notation was largely due to a wide gap between the syntax and the execution semantics.

In 2001, TTCN-2 got a major face lift and TTCN-3 saw the light of day. Though TTCN-3 still retains the basic characteristics of TTCN-2, the syntax of TTCN-3 was designed in line with a procedural programming language. Programmers and test designers can write test cases the way they write programs, thereby reducing the gap between syntax and execution semantics. The programming language look and feel of TTCN-3 makes it more acceptable. TTCN-3 has seen much improvement since 2001, and the user and support bases are ever expanding. There is increasing tool support and an active team maintaining the language.

In this section, we give a brief introduction to TTCN-3. We focus on the core features of TTCN-3, such as *module*, *data types*, *templates*, *ports*, *components*, and *test cases*.

10.11.1 Module

A module is a fundamental unit for specifying test cases. In terms of programming, a test case comprises some data declarations and some execution behavior, which are specified in the form of one or more modules. The execution behavior of a test case is referred to as the *control* part. The structure of a module is shown in Figure 10.26.

10.11.2 Data Declarations

TTCN-3 allows the declarations of constants and variables. A constant is denoted by the const keyword, and the value of the constant is assigned at the point of declaration. The value of a constant cannot be changed afterward.

TTCN-3 has its own scoping rules for all data definitions. All declarations made at the "top" level, that is, module level, are accessible throughout the module. Here, top level means before the control part. The concept of a "code block," enclosed by a matching pair of curly braces, helps us understand scoping rules. Definitions made within a specific code block are only accessible within that code block, and identifiers are not reused in nested code blocks. In other words, there do not exist two data items with the same name but different scopes. No *variable* can be defined at the module level, and, therefore, all module-level declarations are constants. Absence of module-level variables means absence of global variables.

```
/* One can document a module by writing comments in this way. */
// Additional comments can be included here.
module ExampleTestModule1 { // A module can be empty.
      // First, define some data to be used in the control part
      const integer MaxCount := 15;
      constant integer UnitPacket = 256;
      // More data can be defined here ...

      // Second, specify the control part to execute
     control { // The control part is optional
            var integer counter := 0;
            var integer loopcount := MaxCount;
            const integer PacketSize := UnitPacket * 4;

            // Specify more execution behavior here ...

     } // End of the control part
} // end of module TestCase1
```

Figure 10.26 Structure of module in TTCN-3.

Since a test case can have distributed components, it is difficult to guarantee the semantics of global variable across distributed test components.

TTCN-3 supports a powerful set of built-in types, such as integer, float, Boolean (universal), charstring, verdicttype, bitstring, hexstring, octetstring, objid, and default. Such a rich set of built-in types is essential to protocol testing. TTCN-3 also allows the definitions of structured types and list types from existing types. Constructs to define structured types are enumerated, record, set, and union. Similarly, constructs to define list types are array, set of, and record of.

TTCN-3 allows programmers to use the concept of data subtyping. Subtyping means restricting the values of a type to a subset of all values allowed by the original type. For example, given the set of all integers, one can create a subtype of all unsigned numbers that can be represented with 16 bits. Such a subtype is useful in representing port numbers while testing using TCP. Two examples of subtyping are shown in Figure 10.27, where TCPPort is a new user-defined type. A variable of type TCPPort can take on values in the range 0, . . . , 65535. Similarly, IPUserProtocol is a subtype of charstring. A variable of type IPUserProtocol can take on values from the given set and not outside the set.

A PDU in a communication protocol can be defined using a record type. In some protocols, their PDUs are simply called *messages*. TTCN-3 allows one to

```
type integer TCPPort ( 0 .. 65535 ); // a 16 bit unsigned number
type charstring IPUserProtocol ( ``TCP'', ``UDP'', ``OSPF'',
                                 ``RIP'' );
```

Figure 10.27 Definitions of two subtypes.

define a PDU field of arbitrary bit length. For example, one can define PDU fields of 1 bit, 4 bits, and 6 bits which are found in the packet header of IPv4 (Internet Protocol version 4). Often protocols put a limit on the length of some of the PDU fields. Such limits are easily expressed using the length attribute of variables.

One creates instances of those types by using concrete values after defining a PDU or a message type. Such concrete instances have two applications: (i) send messages to a remote protocol entity and (ii) receive messages having the desired values. Concrete instances of message types are called *templates* in TTCN-3. TTCN-3 allows parameterization of templates for easily creating messages. An example of a template is shown in Figure 10.28, where MyMessage is a message type and SFCRequest is an instance of MyMessage. The MyMessage type consists of four fields. The response field can be *omitted* while creating an instance of the template.

An example definition of a response message of type MyMessage is shown in Figure 10.29. One can specify what message is expected to be received from a SUT by defining a response message. The identification field can be used to associate a response message with a request message. A "?" value of the input field in the SFCResponse tells us that the field can contain any value, which is ignored. The response field in the received SFCResponse carries the actual value expected to be received from the SUT.

10.11.3 Ports and Components

A test infrastructure may consist of one or more test components, where a test component is an entity that can send and/or receive messages (templates). There is

```
template MyMessage SFCRequest( Identification id, Input Ival) := {
     identification := id,
     msgtype        := Request,
     input          := Ival,
     response       := omit // ``omit'' is a keyword
}
```

Figure 10.28 Parameterized template for constructing message to be sent.

```
template MyMessage
     SFCResponse( Identification id, Response Rval) := {
     identification := id,
     msgtype        := Response,
     input          := ?, // This means the field can contain any
                       value
     response       := Rval
}
```

Figure 10.29 Parameterized template for constructing message to be received.

Figure 10.30 Testing (a) square-root function (SRF) calculator and (*b*) port between tester and SRF calculator.

a need for communication between test components and between the SUT and the test components. The points at which communication takes place are called ports in TTCN-3. A port is modeled as an infinite FIFO (first in–first out) queue from the viewpoint of a receiver. Two kinds of communication semantics are associated with a port: *message* semantics and *procedure call* semantics. One can specify the kinds of messages or calls a port handles and the *input–output* direction of the port. A port can be an *in* (input) port, an *out* (output) port, or *in–out* (both input and output) port. We explain the testing of a square-root function calculator (SFC) in terms of a test component and the SFC component, as shown in Figure 10.30*a*. A port between the two components is shown in Figure 10.30*b*. The declaration of an in–out port type that handles messages of type MyMessage is shown in Figure 10.31. Figure 10.32 illustrates the attachment of test component SFCTester with the SFCServerPort. The SFC component is assumed to be running on a port called SFCServerPort.

10.11.4 Test Case Verdicts

A test designer wants to conclude something after having executed a test case. For example, two simple conclusions are whether the SUT has *passed* or *failed* the test. If a SUT behaves as expected, then it is natural to say that it has passed the test; otherwise the test has failed. Thus, pass and fail are two obvious test

```
type port SFCPort message { // The SFCPort type has a ``message''
                            // semantics
      inout MyMessage        // The SFCPort type is of inout type
                             // handling
                           // messages of type MyMessage
}
```

Figure 10.31 Defining port type.

```
type component SFCTester {
     port SFCPort SFCServerPort
}
```

Figure 10.32 Associating port with component.

verdicts. However, often the test designer may not be in a position to conclusively say whether the system has passed or failed a test. In such a case, the test designer assigns an *inconclusive* test verdict, which means that further tests need to be conducted to refine the inconclusive verdict into either a pass or a fail verdict.

TTCN-3 provides a mechanism to record test verdicts. Associated with each test component there is an implicitly defined variable of type verdicttype. The initial, default value of the implicitly defined test verdict variable is *none*. A test designer can assign new values to the test verdict variable by calling the operation setverdict. For example, the calls setverdict(Pass), setverdict(Fail), and setverdict(Inconc) assign verdicts pass, fail, and inconclusive, respectively. There is a fourth test verdict, namely, *error*, which is assigned by the run time system. The run time assigns the test verdict error to the test verdict variable when a run time error occurs. For example, dividing a numeric value by zero leads to a run time error. TTCN-3 does not allow a test designer to explicitly set the value of the verdict variable to error, that is, the operation setverdict(error) is not allowed in TTCN-3. The value of the verdict assigned so far in a test component can be retrieved with the operation getverdict.

10.11.5 Test Case

A simple test case running on the SFCTester component is shown in Figure 10.33. We assume a message passing semantic of communication between the test component SFCTester and the SUT, namely, SFC. SFCTester sends a request message to the SFC and waits for a response message from SFC. Since a faulty SFC may not generate a response message, SFCTester must exit from a possible infinite wait. Therefore, we define a timer, namely, responseTimer, by using the keyword *timer*. Next, the test case sends a message, namely, SFCMessage, and starts a timer with

```
// A test case description with alternative behavior
testcase  SFCtestcase1() runs on SFCTester {
     timer responseTimer; // Define a timer
     SFCPort. send (SFCRequest(7, 625));
     responseTimer.start(5.0);

     alt { // Now handle three alternative cases ...
           // Case 1: The expected result of computation
           // is received.
           [] SFCPort.receive(SFCResponse(7, 25)) {
                 setverdict(pass);
                 responseTimer.stop;
              }

           // Case 2: An unexpected result of computation
           // is received.
           [] SFCPort.receive {
                 setverdict(fail);
                 responseTimer.stop;
              }
           // Case 3: No result is received within a reasonable
           // time.
           [] responseTimer.timeout {
                 setverdict(fail);
              }
     }
     stop;
} // End of test case
```

Figure 10.33 Test case for testing SRF calculator.

a duration of 5.0 s. SFCMessage contains two important fields, namely, identifier and input. We are interested in calculating the square root of input, which has been given the value of 625. In the given example, the identifier takes on an arbitrary, but known, value of 7.

After sending a request to the SFC component and starting a timer, the test component waits for the expected message to arrive from the SFC component. At this point, three different situations can arise. First, the SFC component can correctly respond with the expected result of 25 in a message having an identifier value of 7. Second, the SFC component responds with an incorrect value; for example, the identifier field can have an incorrect value or the response field can have a value *not* equal to 25. Third, the SFC component may fail to produce a response.

In the first case, SFCTester records a pass test verdict. In the second case, the test component records a fail test verdict. In the third case, the test component comes out of an infinite wait and records a fail test verdict. If one of the first two alternatives occur, the test component naturally stops the timer, whereas in the third alternative the test component gets a timeout. The alternative behaviors have been

```
module ExampleTestModule2 {
      // Define variables and constants to be used ...
      // Define templates to be used ...
      // Define ports to be used ...
      // Associate test components with ports ...
      // Define test cases, such as SFCtestcase1 ...

     control {
           execute( SFCtestcase1() );
     }
}
```

Figure 10.34 Executing test case.

expressed by using the ***alt*** construct, and individual alternative behaviors have been explicitly represented by using the [ ] symbol.

Finally, the test case SFCtestcase1 can be executed by having an ***execute*** statement in the **control** portion of a test module, as shown in Figure 10.34.

10.12 EXTENDED FSMS

Two conceptual components of a software system are *flow of control* and *manipulation of data*. A FSM model is useful at describing the former but has no provision for specifying the latter. Though there are many systems which can be conveniently and accurately modeled as FSMs, many systems in the real world require us to specify the associated computations while a system makes transitions from state to state. The associated computations can take on the following forms:

- Manipulate local variables.
- Start and stop timers.
- Create instances of processes.
- Compare values and make control flow decisions.
- Access databases.

In this section we will frequently refer to the FSM model of a telephone PBX shown in Figure 10.8. There is a need to record the start time of a call, and this can be done by noting the time when the FSM moves to the talk state. Constructs to start and stop timers are essential to the specification of real-time systems. Manipulation of local variables and conditional jumps are central to repeatedly executing a sequence of state transitions for a certain number of times. Accessing a database is essential to logging values of local variables for business purposes, such as billing and maintenance.

Therefore, there is a need to augment the basic structure of a state transition with capability to perform additional computations, such as updating values of variables, manipulating timers, and making decisions. Such an *extension* of an FSM

results in an *extended finite-state machine* (EFSM). Processes in the Specification and Description Language (SDL) [20, 21] are EFSMs. SDL processes are built around the following basic concepts:

- *System*, which is described hierarchically by elements called systems, blocks, channels, processes, services, signals, and signal routes
- *Behavior*, which is described using am extension of the FSM concept
- *Data*, which are described using the concept of abstract data types with the addition of a notion of program variable and data structure
- *Communication*, which is asynchronous via channels that are infinite queues

An SDL specification can be written in two different forms: *SDL/GR* and *SDL/PR*. SDL/GR is a graphical syntax which shows most of the language constructs in flow-chart-like graphical form. Data definitions can only be textually represented. On the other hand, SDL/PR is written in textual form for machine processing. The "PR" in SDL/PR stands for *processing*. A one-to-one mapping is defined between the two forms. We show the structure of a state transition in an FSM in Figure 10.35*a*. This state transition specifies that if the FSM—the complete one has not been shown here—receives input *a* in state *A*, the machine produces an output *b* and moves to state *B*. The SDL/GR representation of the same transition is shown in Figure 10.35*b*. This example shows that one can easily represent an FSM as an SDL process. The state transition of Figure 10.35*b* has been extended in Figure 10.35*c* by including a *task* block that starts a timer. We show two state transitions—one from state *A* to state *B* and the other from

Figure 10.35 Comparison of state transitions of FSM and EFSM.

state *A* to state *C*—that include *task* blocks and a *decision* box (Figure 10.35*d*). To summarize the discussion of an EFSM, an EFSM is similar to an FSM with *computations* associated with every state transition. In the following, we give an EFSM model of a real-life system.

Example: Door Control System. Let us consider a *door control* system, as illustrated in Figure 10.36. A door is equipped with an electromechanical unit so that it can be opened and closed by sending electrical signals to the unit from a control panel. The user keys in a four-digit number to access a door. The door control unit compares the user-supplied number with a programmed number. If the two numbers match, then the control system turns the green light on, sends a signal to the electromechanical unit to open the door, and starts a timer called DoorOpen. If the two numbers do not match, then the red light goes on for 5 seconds followed by a welcome message to enter the PIN.

The door unit detects whether or not someone has passed through the door. If someone passes through the door, the door unit sends a Passed signal to the control unit. If no one passes through the door, the door unit sends a NotPassed signal to the control unit. If the control unit receives neither a Passed nor a NotPassed signal, it will eventually receive a timeout from the DoorOpen timer. The door unit must produce an appropriate signal irrespective of whether a user passes through the door or not. If the door unit fails to produce these signals, then the control unit assumes that the door unit is faulty, makes the yellow light turn on, displays a message saying that users are not welcome, and waits in the idle state for the necessary repair work to be done.

Figure 10.36 Controlled access to a door.

If the control unit receives a Passed or a NotPassed signal while the green light is on, the DoorOpen timer is stopped, the green light is switched off, a signal is sent to the door to close, and, finally, the system readies itself to handle another user request. A user could change his or her mind while entering the PIN by pressing the cancel button. Moreover, the same effect of cancellation can be achieved by abandoning the PIN entry process which is detected by a timer.

The above description of a door control system has been specified in SDL in Figures 10.37, 10.38, and 10.39. The system-level diagrams are found in Figure 10.37, and the behavior of the door control system can be found in the process diagrams of Figures 10.38 and 10.39. The reader may be reminded that we have specified the behavior of the door control system as an EFSM in the form of an SDL process shown in Figures 10.38 and 10.39.

Figure 10.37 SDL/GR door control system.

Figure 10.38 Door control behavior specification.

Figure 10.39 Door control behavior specification.

10.13 TEST GENERATION FROM EFSM MODELS

This section explains the mechanisms to generate test cases from an EFSM model of a system. Specifically, we generate test cases from an SDL process. Assume that we have an EFSM model E and a program P_E that correctly implements E. Our task is to generate test cases from E to test program P_E. Ideally, we want to generate a test suite T_E from E so that by testing P_E with T_E, we verify whether P_E implements the functions specified in E. Theoretically, in general, it is impossible to design such a test suite which, when applied to P_E, can reveal all the faults with P_E. In practice, however, we want to design T_E with the goal of verifying that P_E *behaves as expected for the commonly used input sequences*. This goal is achieved by designing T_E in two phases:

Phase 1: Identify a set of state transition sequences such that each sequence of state transitions represents a common *use sequence*.

Phase 2: Design a test case from each state transition sequence identified above.

In **phase 1**, the main task is to *identify* sequences of state transitions with two objectives. The first objective is whether or not an implementation P_E supports the functions represented by the chosen sequences. The second objective is to reveal faults with the implementation. Here, once again, state transition sequences are chosen to meet some coverage criteria. We remind the reader that a coverage criterion provides us with a systematic method for selecting sequences of state transitions. In addition, a coverage criterion gives us the minimum number of sequences we need to identify. While selecting state transition sequences, we must do the following:

- Perform tests to ensure that the SUT produces expected sequences of outcomes in response to input sequences.
- Perform tests to ensure that the SUT takes the right actions when a timeout occurs.
- Perform tests to ensure that the SUT has appropriately implemented other task blocks, such as resource allocation and deallocation, database accesses, and so on.

As discussed earlier in this chapter, there are two well-known coverage criteria for selecting state transition sequences, namely, state coverage and transition coverage. The former criterion means identifying a set of transition tours such that every state is visited at least once. The latter criterion means identifying a set of transition tours such that every state transition is visited at least once. **Phase 2**, that is, the process of generating a test case from a transition sequence, is explained in the following by means of an example.

Example: Test Generation from Door Control System. We show an EFSM model of a door control system in Figures 10.38 and 10.39. The control system has four states, namely, GETDIGIT, RED, DOOROPEN, and IDLE. After a sequence of initialization steps, the EFSM moves to its initial state GETDIGIT. Normal behavior of the system is restricted to the state transitions among the three states GETDIGIT, RED, and DOOROPEN. If the electromechanical door unit fails to respond to signals from the door control system, the EFSM moves from the DOOROPEN state to the IDLE state. The IDLE state can be thought of as an error state. Once faults with the door unit are fixed and the control unit is reset, the system moves back from the IDLE state to the GETDIGIT state. Therefore, by adding a sequence of tasks from the IDLE state to the START of the EFSM, we can make the EFSM strongly connected.

Assuming that the SDL process of Figures 10.38 and 10.39 is in state GETDIGIT, we design a test case that covers the transition tour shown in Figure 10.40. This transition tour executes the loop induced by digit inputs in state GETDIGIT while the read digits are incomplete. We assume that a user keys in an acceptable sequence of digits; that is, *analysis* of the digits in the decision box is *complete* and *valid*.

```
GETDIGIT ⇒ ... GETDIGIT ⇒ ... GETDIGIT ⇒ DOOROPEN ⇒ GETDIGIT.
```

Figure 10.40 Transition tour from door control system of Figures 10.38 and 10.39.

The reader may notice that control can flow from the DOOROPEN state to the GETDIGIT state in two ways, namely, by receiving Passed and NotPassed inputs, corresponding to someone passing through the door and not passing through the door, respectively. While generating a test case to cover the above transition tour, we assume that the Passed input occurs. In the following, we generate a test case using the three test design principles discussed in Section 10.5:

Step 1: We transform the inputs and outputs in a transition tour into outputs and expected inputs, respectively, to derive the core behavior of a test case. We obtain the ports, namely, DISPLAY, KEYPAD, GREENLIGHT, and DOOR, as shown in Figure 10.41 by analyzing the transition tour of Figure 10.40. The sequence of *inputs* and *outputs* found in Figures 10.38 and 10.39, corresponding to the transition tour shown in Figure 10.40, have been transformed into outputs and inputs, respectively, in the TTCN-3 notation of Figure 10.42. The reader may note that the test behavior shown in Figure 10.42 is in terms of the four ports identified in Figure 10.41.

If the test system observes that the SUT behaves as specified in Figure 10.42, then it assigns a Pass test verdict. We informally express a condition such that a test case outputs a number of digits in line 3 of Figure 10.42. We have obtained the test behavior shown in Figure 10.43 from Figure 10.42 by refining the if statement in line 3 of Figure 10.42 as follows. We initialize a counter count with value 0 in line 1 of

Figure 10.41 Testing door control system.

```
 1.   label label1  KEYPAD.send(digit);
 2.     DISPLAY.receive(Asterisk);
 3.      if (NOT enough number of digits) goto label1;
            // Not in TTCN-3 form yet.
 4.         DOOR.receive(Open);
 5.            GREENLIGHT.receive(On);
 6.               DOOR.send(Passed);
 7.                  GREENLIGHT.receive(Off);
 8.                     DOOR.receive(Close);
 9.                        DISPLAY.receive(Welcome);
10.                           setverdict(Pass);
```

Figure 10.42 Output and input behavior obtained from transition tour of Figure 10.40.

```
 1.   count := 0; // Count is of type integer.
 2.   label label1 KEYPAD.send( digit );
 3.       DISPLAY.receive( Asterisk );
 4.         count := count + 1;
 5.           if (count < 4) goto label1;
 6.           else {
 7.               DOOR.receive( Open );
 8.                  GREENLIGHT.receive( On );
 9.                     DOOR.send( Passed );
10.                        GREENLIGHT.receive( Off );
11.                           DOOR.receive (Close);
12.                              DISPLAY.receive(Welcome);
13.                                  setverdict(Pass);
14.           };
```

Figure 10.43 Test behavior obtained by refining if part in Figure 10.42.

Figure 10.43 and increment the counter in line 4 after the test behavior outputs a digit. The informal condition part of the if statement in line 7 of Figure 10.42 has been refined into a concrete condition in Figure 10.43 (line 5) by assuming that the door control system accepts a sequence of digits of length 4.

Step 2: We augment a test behavior to prepare itself to receive events *other* than the expected events. This is done by including an *any* event, denoted by a "?," as an alternative event to each expected event. When we apply this transformation step to the test behavior shown in Figure 10.43, we obtain the test behavior shown in Figure 10.44. For example, the any event denoted by a "?" in line 20 of Figure 10.44 is designed to match with any event which does not match the expected event specified in line 18. The any events in lines 23, 26, 29, 32, and 36 are alternative events to the expected events in lines 16, 14, 11, 9, and 4, respectively.

```
 1.   count := 0; // Count is of type integer.
 2.   label label1 KEYPAD.send( digit );
 3.   alt {
 4.       [] DISPLAY.receive( Asterisk );
 5.          count := count + 1;
 6.          if (count < 4) goto label1;
 7.          else {
 8.                 alt {
 9.                    [] DOOR.receive( Open );
10.                      alt {
11.                           [] GREENLIGHT.receive( On );
12.                              DOOR.send( Passed );
13.                              alt {
14.                                    [] GREENLIGHT.receive( Off );
15.                                      alt {
16.                                          [] DOOR.receive (Close);
17.                                             alt {
18.                                                 [] DISPLAY.receive(Welcome);
19.                                                    setverdict(Pass);
20.                                                 [] DISPLAY.receive(?);
21.                                                    setverdict(Fail);
22.                                             }
23.                                          [] DOOR.receive(?);
24.                                             setverdict(Fail);
25.                                      }
26.                                    [] GREENLIGHT.receive(?);
27.                                       setverdict(Fail);
28.                                 }
29.                           [] GREENLIGHT.receive(?);
30.                              setverdict(Fail);
31.                       }
32.                    [] DOOR.receive(?);
33.                       setverdict(Fail);
34.                 }
35.              } // end of else
36.       [] DISPLAY.receive(?);
37.          setverdict(Fail);
38.   }
```

Figure 10.44 Test behavior that can receive unexpected events (derived from Figure 10.43).

Step 3: We augment a test behavior with timers so that a test system does not enter into a deadlock in case the SUT produces no output, that is, before waiting to receive an expected event, the test behavior starts a timer. The corresponding timeout input event is specified as an alternative to the expected input and the any event explained before. When we apply this transformation step to the test behavior shown in Figure 10.44, we obtain the test behavior shown in Figure 10.45. Finally, the test behavior is augmented with test verdicts. A Pass verdict is assigned if the system under test behaves as expected, and this is shown in line 10 of Figure 10.42. The Pass verdict in line 10 of Figure 10.42 can be found in line 13 in Figure 10.43, line 19 in Figure 10.44, and line 27 in Figure 10.45, as we

```
count := 0; // Count is of type integer.
label label1 KEYPAD.send( digit );
Timer1.start(d1);
alt {
    [] DISPLAY.receive( Asterisk );
       Timer1.stop;
       count := count + 1;
       if (count < 4) goto label1;
       else {
              Timer2.start(d2);
              alt {
                 [] DOOR.receive( Open );
                    Timer2.stop; Timer3.start(d3);
                  alt {
                       [] GREENLIGHT.receive( On );
                          Timer3.stop;
                          DOOR.send( Passed );
                          Timer4.start( d4 );
                          alt {
                               [] GREENLIGHT.receive( Off );
                                  Timer4.stop; Timer5.start( d5 );
                                  alt {
                                      [] DOOR.receive (Close);
                                         Timer5.stop; Timer6.start( d6 );
                                         alt {
                                             [] DISPLAY.receive(Welcome);
                                                Timer6.stop; setverdict(Pass);
                                             [] DISPLAY.receive(?);
                                                Timer6.stop; setverdict(Fail);
                                             [] Timer6.timeout;
                                                setverdict(Inconc);
                                          }
                                      [] DOOR.receive(?);
                                         Timer5.stop; setverdict(Fail);
                                      [] Timer5.timeout;
                                         setverdict(Inconc);
                                   }
                                [] GREENLIGHT.receive(?);
                                   Timer4.stop; setverdict(Fail);
                                [] Timer4.timeout;
                                   setverdict(Inconc);
                             }
                       [] GREENLIGHT.receive(?);
                          Timer3.stop; setverdict(Fail);
                       [] Timer3.timeout;
                          setverdict(Inconc);
                  }
                 [] DOOR.receive(?);
                    Timer2.stop; setverdict(Fail);
                 [] Timer2.timeout;
                    setverdict(Inconc);
              }
           }
    [] DISPLAY.receive(?);
       Timer1.stop; setverdict(Fail);
    [] Timer1.timeout;
    setverdict(Inconc);
}}
```

Figure 10.45 Core behavior of test case for testing door control system (derived from Figure 10.44.)

go on making the test case more and more complete. We assign a Fail verdict when the test system receives an unexpected event in the form of any events and an Inconclusive verdict when a timeout occurs.

10.14 ADDITIONAL COVERAGE CRITERIA FOR SYSTEM TESTING

We discussed two coverage criteria, namely, state coverage and state transition coverage, to select test cases from FSM and EFSM models of software systems in Sections 10.5 and 10.13. Those two criteria focused on sequences of events, possibly including internal events, occurring at PCOs. In this section, we explain some more coverage criteria in line with the concepts of functional testing.

The reader may recall from Chapter 9 on functional testing that we identify the domains of input and output variables and select test data based on special values from those domains. For example, if an output variable takes on a small number of discrete values, then test cases are designed to make the SUT produce all those output values. If an output variable takes on values from a contiguous range, test data are selected such that the SUT produces the external points and an interior point in the specified range.

In line with the above concept of functional testing, in the following, we explain some coverage criteria for *event-driven* systems modeled as FSMs or EFSMs. First, we identify the PCOs, also referred to as ports—points where a SUT interacts with the external world. Next, we apply the following coverage criteria to select test cases:

PCO Coverage: Select test cases such that the SUT receives an event at each input PCO and produces an event at each output PCO.

Sequences of Events at PCOs: Select test cases such that *common* sequences of inputs and outputs occur at the PCOs. By a common sequence we mean sequences commonly found in the uses of the system.

Events Occurring in Different Contexts: In many applications, a user produces an event for the system by pressing a button, for example. Here, a button represents an input PCO, and pressing the button represents an event. However, the semantics of pressing a button, that is, interpretations of events at a PCO, depend on *data contexts* used by the system. For a given context, test data are selected such that all events, both desired and undesired, occur in the context.

Inopportune Events: A system is expected to discard invalid, or erroneous, events. On the other hand, inopportune events are normal events which occur at an inappropriate time.

Example: Automated Teller Machine. Let us consider the user interface of an ATM system as shown in Figure 10.46. A user selects one of the transaction options and specifies a transaction amount using buttons B1 through B6. The meaning of a

Figure 10.46 User interface of ATM.

Figure 10.47 Binding of buttons with user options.

button changes as the message in the display area changes. For example, a user can choose a transaction option, such as Deposit or Withdraw, by pressing buttons B2 or B4, respectively, as shown in Figure 10.47. However, as shown in Figure 10.48, when it comes to selecting an amount, buttons B2 and B4 represent options $40 and $100, respectively. Moreover, all the buttons for the *amount* context shown in Figure 10.48 do not represent the same *type* of data. For example, buttons B1 through B5 represent discrete, integer values, whereas button B6 gives the user an option to specify other values. In the context shown in Figure 10.47, buttons B3 and B5 are undefined, and, thus, those are potential sources of undesirable (erroneous) events. Test cases must be selected to observe how the system responds to undefined events. The OK button produces a normal event while the user is entering a PIN after inserting the cash card. The OK event essentially tells the system that the user

Figure 10.48 Binding of buttons with cash amount.

has completely entered a PIN. However, it may not be meaningful to press OK in the context shown in Figure 10.48. Therefore, pressing the OK button produces an inopportune event for the system to handle. Test cases need to be selected to consider inopportune events in addition to normal (valid) and abnormal (invalid or erroneous) events.

10.15 SUMMARY

This chapter began with the classification of software systems in terms of *stateless* and *state-oriented* systems. A stateless system does not memorize the previous inputs, and, therefore, its response to a new input does not depend on the previous inputs. On the other hand, a state-oriented system memorizes the sequence of inputs it has received so far in the form of a *state*. Next, we examined the concept of *ports* in the context of software testing. State-oriented systems were modeled as FSMs.

We explained two broad methods for generating test sequences from an implementation of an FSM—one *without* state verification and one *with* state verification. One can design weak test sequences in the form of transition tours without using state verification. On the other hand, one can design stronger test sequences by performing state verification with one of three kinds of sequences, namely, *unique input–output sequences*, *distinguishing sequences*, and *characterizing sequences*.

We examined four test architectures: *local*, *distributed*, *coordinated*, and *remote* with the test generation methods in place. These abstract test architectures are described in terms of controllable inputs given to an IUT and observable outputs from the IUT. The concept of *points of control and observation* (PCOs) is introduced. A PCO is a point of interaction called a port, which is accessible to a test entity, between the test entity and an IUT.

We provided a brief introduction to Testing and Test Control Notation Version 3 (TTCN-3), which is a common notation to specify test cases for a communication system. Specifically, the following features are described with examples: *module*, *data types*, *templates*, *ports*, *components*, and *test cases*.

Finally, we introduced the concept of an extended finite-state machine (EFSM), which has the capability to perform additional computations such as updating variables, manipulating timers, and making decisions. The concept of a *process* in SDL allows one to specify a module in the form of an EFSM. Finally, we presented the ways to generate test cases from an EFSM model.

LITERATURE REVIEW

An excellent collection of papers on FSM-based testing can be found in the book edited by Richard J. Linn and M. Umit Uyar [11]. Many articles referenced in this chapter have been reprinted in the book.

Those interested in knowing more about EFSM-based testing may refer to the excellent book by B. Sarikaya (*Principles of Protocol Engineering and Conformance Testing*, Ellis Horwood, Hemel Hempstead, Hertfordshire, 1993). Sarikaya explained formal specification languages Estelle, SDL, LOTOS, TTCN (the original Tree and Tabular Combined Notation), and Abstract Syntax Notation One (ASN.1) in the first part of his book. In the second part of the book, he explains the generation of transition tours from an unified model of Estelle, SDL, and LOTOS using the concepts of *control flow* and *data flow*.

In the past decade, researchers have proposed several techniques for generating test cases from nondeterministic FSMs. In the following we list the commonly referenced ones:

A. Alur, C. Couroubetis, and M. Yannakakis, "Distinguishing Tests for Nondeterministic and Probabilistic Machines," in *Proceedings of the 27th Annals of the ACM Symposium on Theory of Computing*, Las Vegas, Nevada, ACM Press, New York, 1995, pp. 363–372.

R. M. Hierons, "Applying Adaptive Test Cases to Nondeterministic Implementations," *Information Processing Letters*, Vol. 98, No. 2, April 2006, pp. 56–60.

R. M. Hierons and H. Ural, "Reducing the Cost of Applying Adaptive Test Cases," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Vol. 51, No. 1, January 2007, pp. 224–238.

G. Luo, G. v. Bochmann, and A. Petrenko, "Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method," *IEEE Transactions on Software Engineering*, Vol. 20, No. 2, February 1994, pp. 149–162.

P. Tripathy and K. Naik, "Generation of Adaptive Test Cases from Nondeterministic Finite State Models," in *Proceeding of the Fifth International*

Workshop on Protocol Test Systems, G. v. Bochmann, R. Dssouli, and A. Das, Eds., Montreal, North-Holland, Amsterdam, 1992, pp. 309–320.

F. Zhang and T. Cheung, "Optimal Transfer Trees and Distinguishing Trees for Testing Observable Nondeterministic Finite-State Machines," *IEEE Transactions on Software Engineering*, Vol. 29, No. 1, January 2003, pp. 1–14.

REFERENCES

1. S. Naito and M. Tsunoyama. Fault Detection for Sequential Machine by Transition Tours. In *Proceedings of the 11th IEEE Fault Tolerant Computer Symposium*, Los Alamitos, CA, IEEE Computer Society Press, 1981, pp. 238–243.
2. B. Sarikaya and G. v. Bochmann. Some Experience with Test Sequence Generation. In *Proceedings of Second International Workshop on Protocol Specification, Testing, and Verification*, North Holland, Amsterdam, The Netherlands, 1982, pp. 555–567.
3. E. P. Hsieh. Checking Experiments for Sequential Machines. *IEEE Transactions on Computers*, October 1971, pp. 1152–1166.
4. K. K. Sabnani and A. T. Dahbura. A Protocol Testing Procedure. *Computer Networks and ISDN System*, Vol. **15**, 1988, pp. 285–297.
5. K. Naik. Efficient Computation of Unique Input/Output Sequences in Finite-State Machines. *IEEE/ACM Transactions on Networking*, August 1997, pp. 585–599.
6. G. Gonenc. A Method for the Design of Fault Detection Experiments. *IEEE Transactions on Computers*, June 1970, pp. 551–558.
7. F. C. Hennie. Fault-Detecting Experiments for Sequential Circuits. In *Proceedings of the 5th Annual Symposium on Switching Circuit Theory and Logical Design*, Princeton University, Princeton, IEEE Press, New York, Lenox Hill Station, November 1964, pp. 95–110.
8. A. Gill. State-Identification Experiments in Finite Automata. *Information and Control*, Vol. **4**, 1961, pp. 132–154.
9. Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, New York, 1978.
10. T. S. Chow. Testing Software Designs Modeled by Finite State Machines. *IEEE Transactions on Software Engineering*, May 1978, pp. 178–187.
11. R. J. Linn and M. U. Uyar, Ed. *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press, Los Alamitos, CA, 1994.
12. D. Sidhu and T. Leung. Fault Coverage of Protocol Test Methods. In *Proceeding of the IEEE INFOCOM*, IEEE Press, New Orleans, LA, March 1988, pp. 80–85.
13. A. T. Dahbura and K. K. Sabnani. An Experience in the Fault Coverage of a Protocol Test. In *Proceeding of the IEEE INFOCOM*, IEEE Press, New Orleans, LA, March 1988, pp. 71–79.
14. Information Processing Systems. *OSI Conformance Testing Methodology and Framework*. ISO/IEC JCT 1/SC 21 DIS 9646, Part 3, February 1989. International Organization for Standardization, available at http://www.standardsinfo.net/.
15. Information Processing Systems. *OSI Conformance Testing Methodology and Framework*. ISO/IEC JCT 1/SC 21 DIS 9646, Parts 4–5, March 1989. International Organization for Standardization, available at http://www.standardsinfo.net/.
16. Information Processing Systems. *OSI Conformance Testing Methodology and Framework*. ISO/IEC JCT 1/SC 21 DIS 9646, Parts 1–2, November 1988. International Organization for Standardization, available at http://www.standardsinfo.net/.
17. R. J. Linn, Jr. Conformance Testing for OSI Protocols. *Computer Networks and ISDN Systems*, Vol. **18**, 1989/1990, pp. 203–219.
18. D. Rayner. OSI Conformance Testing. *Computer Networks and ISDN Systems*, Vol. **14**, 1987, pp. 79–98.
19. C. Willcock, T. Deiss, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN-3*. Wiley, New York, 2005.

20. F. Belina, D. Hogrefe, and A. Sarma. *SDL—With Application from Protocol Specification*. Prentice-Hall, Upper Saddle River, NJ, 1991.
21. CCITT. *Specification and Description Language*, Recommendation z.100, CCITT SG X, ITU, Geneva, Switzerland, 1992.

Exercises

1. Considering the FSM of Figure 10.17 discussed in this chapter, provide a test sequence table, similar to Table 10.12, for the state transition $(D, B, b/y)$.
2. What are the fundamental differences between UIO sequence and distinguishing sequence methods of state verification?
3. Consider the FSM $G = <S, I, O, A, \delta, \lambda>$, shown in Figure 10.49, where $S = \{A, B, C\}$ is the set of states, $I = \{a, b, c\}$ is the set of inputs, $O = \{e, f\}$ is the set of outputs, A is the initial state, $\delta : S \times I \rightarrow S$ is the next-state function, and $\lambda : S \times I \rightarrow O$ is the output function.
   (a) Generate a distinguishing sequence for the FSM, if it exists.
   (b) Generate characterizing sequences for the FSM.
   (c) Generate UIO sequence(s) for each state of the FSM, if those exist.
   (d) Compare the distinguishing sequence with the UIO sequences generated from the FSM. Are there any similarities and/or differences between the two kinds of sequences?
4. Consider the FSM $H = <S, I, O, A, \delta, \lambda>$ of Figure 10.50, where $S = \{A, B, C, D, E, F, G\}$ is the set of states, $I = \{\text{ri}, a, c, x, z\}$ is the set

Figure 10.49 FSM G.

Figure 10.50 FSM H.

Figure 10.51 FSM K.

of inputs, $O = \{\text{null}, b, f, d\}$ is the set of outputs, A is the initial state, $\delta : S \times I \rightarrow S$ is the next-state function, and $\lambda : S \times I \rightarrow O$ is the output function.

(a) Generate UIO sequence(s) for each state of the FSM, if those exist.

(b) Generate test sequences for each transition of the FSM using UIO sequences for state verification. Assume that a reset input ri will always bring the FSM back to its initial state A. The FSM produces a null output in response to the ri input.

(c) Represent the following two test cases in TTCN-3 form with verdicts: $(B, C, c/d)$ and $(A, B, a/b)$.

5. Consider the FSM $K = <S, I, O, A, \delta, \lambda>$ of Figure 10.51, where $S = \{A, B, C\}$ is the set of states, $I = \{a, b\}$ is the set of inputs, $O = \{x, y\}$ is the set of outputs, A is the initial state, $\delta : S \times I \rightarrow S$ is the next-state function, and $\lambda : S \times I \rightarrow O$ is the output function.

   (a) Show that the FSM K does not possess a distinguishing sequence.

   (b) Can you generate a UIO sequence for each state of this FSM? Justify your answer.

   (c) Generate characterizing sequences for the FSM.

6. Consider the nondeterministic FSM NFSM $= <S, I, O, A, \delta, \lambda>$ of Figure 10.52, where $S = \{A, B, C, D, E, F, G\}$ is the set of states, $I = \{\text{ri}, a, c, x, z\}$ is the set of inputs, $O = \{\text{null}, b, f, d\}$ is the set of outputs, A is the initial

Figure 10.52 Nondeterministic FSM.

state, $\delta : S \times I \rightarrow S$ is the next-state function, and $\lambda : S \times I \rightarrow O$ is the output function. NFSM is nondeterministic means that for some input $a_k \in I$ there are more than one transitions with different outputs defined for some state.

(a) Generate UIO sequence(s) for each state of the NFSM, if those exist.

(b) Do you think the generated UIO sequences can uniquely identify the states? Justify your answer.

(c) Devise a methodology to uniquely identify the states of the NFSM. (*Hint:* The tree structure will identify the state of the NFSM uniquely.) Using your methodology, generate UIO trees for each state of the NFSM.

(d) Generate test cases for the following transitions: (a) $(F, G, x/d)$, (b) $(B, C, c/d)$, and (c) $(A, B, a/b)$.

(e) Represent the following two test cases in TTCN-3 form with verdicts: $(B, C, c/d)$ and $(A, B, a/b)$.

7. Discuss the effectiveness of the error detection capabilities among the abstract test architectures presented in this chapter.

8. Design a test component to test the system, represented in Figure 10.8, from the standpoint of a local phone alone. In your test component, consider the partial transition tour OH–AD–RNG–TK.

9. Assuming that a called (remote) phone is far away from the calling (local) phone, explain the difficulty you will encounter while designing a test case in TTCN-3 from the following transition tour in Figure 10.8: OH–AD–RNG–TK–LON–TK–RON–AD–OH.

10. Explain the concept of test case verdicts.

CHAPTER 11

System Test Design

Many things difficult to design prove easy to perform.
— *Samuel Johnson*

11.1 TEST DESIGN FACTORS

The central activity in test design is to identify inputs to and the expected outcomes from a system to verify whether the system possesses certain features. A feature is a set of related requirements. The test design activities must be performed in a planned manner in order to meet some technical criteria, such as effectiveness, and economic criteria, such as productivity. Therefore, we consider the following factors during test design: (i) coverage metrics, (ii) effectiveness, (iii) productivity, (iii) validation, (iv) maintenance, and (v) user skill. In the following, we give motivations for considering these factors.

Coverage metrics concern the extent to which the DUT is examined by a test suite designed to meet certain criteria. Coverage metrics lend us two advantages. First, these allow us to quantify the extent to which a test suite covers certain aspects, such as functional, structural, and interface of a system. Second, these allow us to measure the progress of system testing. The criteria may be path testing, branch testing, or a feature identified from a requirement specification.

Each test case is given an identifier(s) to be associated with a set of requirements. This association is done by using the idea of a coverage matrix. A coverage matrix $[A_{ij}]$ is generated for the above idea of coverage [1]. The general structure of the coverage matrix $[A_{ij}]$ is represented as shown in Table 11.1, where T_i stands for the ith test case and N_j stands for the jth requirement to be covered; $[A_{ij}]$ stands for coverage of the test case T_i over the tested element N_j. The complete set of test cases, that is, a test suite, and the complete set of tested elements of the coverage matrix are identified as $T_c = \{T_1, T_2, .., T_q\}$ and $N_c = \{N_1, N_2, .., N_p\}$, respectively.

A structured test case development methodology must be used as much as possible to generate a test suite. A structured development methodology also minimizes maintenance work and improves productivity. Careful design of test cases

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

TABLE 11.1 Coverage Matrix $[A_{ij}]$

Test Case Identifier	Requirement Identifier			
	N_1	N_2	...	N_p
T_1	A_{11}	A_{12}	...	A_{1p}
T_2	A_{21}	A_{22}	...	A_{2p}
T_3	A_{31}	A_{32}	...	A_{3p}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
T_q	A_{q1}	A_{q2}	...	A_{qp}

in the early stages of test suite development ensures their maintainability as new requirements emerge. The correctness of the requirements is very critical in order to develop effective test cases to reveal defects. Therefore, emphasis must be put on identification and analysis of the requirements from which test objectives are derived. Test cases are created based on the test objectives. Another aspect of test case production is validation of the test cases to ensure that those are reliable. It is natural to expect that an executable test case meets its specification before it is used to examine another system. This includes ensuring that test cases have adequate error handling procedures and precise pass–fail criteria. We need to develop a methodology to assist the production, execution, and maintenance of the test suite. Another factor to be aware of is the potential users of the test suite. The test suite should be developed with these users in mind; the test suite must be easy to deploy and execute in other environments, and the procedures for doing so need to be properly documented. Our test suite production life cycle considers all six factors discussed above.

11.2 REQUIREMENT IDENTIFICATION

Statistical evidence gathered by Vinter [2] demonstrates the importance of requirements captured in the development of embedded real-time system projects. Vinter analyzed 1000 defects during his studies, out of which 23.9% of the defect reports stemmed from requirement issues, functionality 24.3%, component structure (the code) 20.9%, data 9.6%, implementation 4.3%, integration 5.2%, architecture 0.9%, testing 6.9%, and other 4.3%. Within those defect reports that were associated with requirements problems, Vinter argued that 48% could be classified as "misunderstanding." Typically, disagreement existed over the precise interpretation of a particular requirement. Missing constraints constitute 19% of the defect reports that stemmed from requirements problem while changed requirements account for 27%. A further 6% were classified as "other" issues. These statistical studies have inspired practitioners to advocate a new vision of requirements identification.

Requirements are a description of the needs or desires of users that a system is supposed to implement. There are two main challenges in defining requirements. First is to ensure that the right requirements are captured, which is essential for

meeting the expectations of the users. Requirements must be expressed in such a form that the users and their surrogates can easily review and confirm their correctness. Therefore, the "form" of a requirement is crucial to the communication between users (and their surrogates) and the representatives of a software development organization. Second is to ensure that the requirements are communicated unambiguously to the developers and testers so that there are no surprises when the system is delivered. A software development team may not be in charge of collecting the requirements from the users. For example, a team of marketing people may collect the requirements. In such a case, there may not be a direct link between the ultimate users and the technical teams, such as development team, system integration team, and system test team. It is undesirable for the teams to interpret the requirements in their own ways. There are two severe consequences of different teams interpreting a requirement in different ways. First, the development team and the system test team may have conflicting arguments about the quality of the product while analyzing the requirements before delivery. Second, the product may fail to meet the expectations of the users. Therefore, it is essential to have an unambiguous representation of the requirements and have it made available in a centralized place so that all the stakeholders have the same interpretation of the requirements [3].

We describe a formal model to capture the requirements for review and analysis within an organization in order to achieve the above two goals. Figure 11.1 shows a state diagram of a simplified requirement life cycle starting from the submit state to the closed state. This transition model provides different phases of a requirement, where each phase is represented by a state. This model represents the life of a requirement from its inception to completion through the following states: submit, open, review, assign, commit, implement, verification, and finally closed. At each of these states certain actions are taken by the owner, and the requirement is moved to the next state after the actions are completed. A requirement may be moved to the decline state from any of the states open, review, assign, implement, and verification for several reasons. For example, a marketing manager may decide

Figure 11.1 State transition diagram of requirement.

that the implementation of a particular requirement may not generate revenue. Therefore, the marketing manager may decline a requirement and terminate the development process.

The state transition diagram showing the life cycle of an individual requirement can be easily implemented using a database system with a graphical user interface (GUI). One can customize any existing open-source tracking system to implement the requirement model described here [4]. The idea behind implementing the state transition diagram is to be able to track the requirements as these flow through the organization [5, 6]. A requirement schema is designed with different tabbed panes using commonly used GUI and database. Subsequently, requirements can be stored and queries generated for tracking and reporting the status of the requirements. A list of fields of the schema is given in Table 11.2. It is necessary to implement a secure access system for different users of the database. Customers

TABLE 11.2 Requirement Schema Field Summary

Field Name	Description
requirement_id	A unique identifier associated with the requirement
title	Title of the requirement; one-line summary of the requirement
description	Description of the requirement
state	Current state of the requirement; can take a value from the set {Submit, Open, Review, Assign, Commit, Implement, Verification, Closed, Decline}
product	Product name
customer	Name of customer who requested this requirement
note	Submitter's note; any additional information the submitter wants to provide that will be useful to marketing manager or director of software engineering
software_release	Assigned to a software release; software release number in which the requirement is desired to be available for end customer
committed_release	Software release number in which requirement will be available
priority	Priority of requirement; can take value from the set {high, normal}
severity	Severity of requirement; can take value from the set {critical, normal}
marketing_justification	Marketing justification for existence of requirement
eng_comment	Software engineering director's comment after review of requirement; comments useful when developing functional specification, coding, or unit testing
time_to_implement	Estimated in person-week; time needed to implement requirement, including developing functional specification, coding, unit, and integration testing
eng_assigned	Assigned to an engineer by software engineering director in order to review requirement
functional_spec_title	Functional specification title
functional_spec_name	Functional specification filename
functional_spec_version	Latest version of functional specification

TABLE 11.2 (*Continued*)

Field Name	Description
decline_note	Explanation of why requirement is declined
ec_number	Engineering change (EC) document number
attachment	Attachment (if any)
tc_id	Test case identifier; multiple test case identifiers can be entered; these values may be obtained automatically from test factory database
tc_results	Test case result; can take value from the set {Untested, Passed, Failed, Blocked, Invalid}; can be automatically obtained from test factory database
verification_method	Verification method; T—by testing; A—by analysis; D—by demonstration; I—by inspection
verification_status	Verification state (passed, failed, incomplete) of requirement.
compliance	Compliance; can take value from the set {compliance, partial compliance, noncompliance}
testing_note	Notes from test engineer; may contain explanation of analysis, inspection, or demonstration given to end customer by test engineer
defect_id	Defect identifier; value can be extracted from test factory database along with test results. If the tc_results field takes the value "failed," then the defect identifier is associated with the failed test case to indicate the defect that causes the failure.

may be given restricted access to the database, so that they can see the status of their requirements within the organization. Customers can generate a traceability matrix from the requirement database system, which gives them confidence about test coverage [7]. A traceability matrix allows one to find a two-way mapping between requirements and test cases as follows [8]:

- From a requirement to a functional specification to specific tests which exercise the requirements
- From each test case back to the requirement and functional specifications

A traceability matrix finds two applications: (i) identify and track the functional coverage of a test and (ii) identify which test cases must be exercised or updated when a system evolves [9]. It is difficult to determine the extent of coverage achieved by a test suite without a traceability matrix. The test suite can contain a sizable number of tests for the features of a system, but without the cross-referencing provided by a traceability matrix it is difficult to discern whether a particular requirement has been adequately covered. The following definition by Gotel and Finkelstein [10] sums up the general view of requirements traceability (p. 97):

> The requirements traceability is the ability to describe and follow the life of a requirement, in both forward and backward direction, i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases.

Submit State A new requirement is put in the submit state to make it available to others. The owner of this state is the submitter. A new requirement may come from different sources: customer, marketing manager, and program manager. A program manager oversees a software release starting from its inception to its completion and is responsible for delivering it to the customer. A software release is the release of a software image providing new features. For example, one can release an OS to customers every eight months by adding new features and using an appropriate numbering scheme, such as OS release 2.0, OS release 2.1, and so on. Usually, the requirements are generated from the customers and marketing managers.

A defect filed by a test engineer may become a requirement in a future release because it may become apparent that the reported defect is not really a true defect and should be treated as an enhancement request. In case of a dispute between the test engineers and the developers, the program manager may make a final decision in consultation with relevant people. The program manager can submit a new requirement based on the issue raised in the defect if he or she decides that the disputed defect is, in fact, an enhancement request. The following fields of the schema given in Table 11.2 are filled out when a requirement is submitted:

requirement_id: A unique identifier associated with the requirement.

priority: A priority level of the requirement—high or normal.

title: A title for the requirement.

submitter: The submitter's name.

description: A short description of the requirement.

note: Some notes on this requirement, if there are any.

product: Name of the product in which the requirement is desired.

customer: Name of the customer who requested this requirement.

The priority level is an indication of the order in which requirements need to be implemented. Requirements prioritization [11] is an important aspect in a market-driven requirements engineering process [12, 13]. All the high-priority requirements should be considered for implementation before the normal priority requirements. The submitter can assign a priority level to a requirement that defines the requirement's level of importance. The marketing manager can move the state from submit to open by assigning the ownership to himself.

Open State In this state, the marketing manager is in charge of the requirement and coordinates the following activities.

- Reviews the requirement to find duplicate entries. The marketing manager can move the duplicate requirement from the open state to the decline state with an explanation and a pointer to the existing requirement. Also, he or she may ensure that there are no ambiguities in the requirement and, if there is any ambiguity, consult with the submitter and update the description and the note fields of the requirement.
- Reevaluates the priority of the requirement assigned by the submitter and either accepts it or modifies it.

- Determines the severity of the requirement. There are two levels of severity defined for each requirement: normal and critical. The severity option provides a tag for the upper management, such as the director of software engineer, in order to review the requirement. If the severity level is critical, then it is a flag to the director to complete the review as soon as possible. Assignment of a severity level is made independent of the priority level.
- Suggests a preferred software release for the requirement.
- Attaches a marketing justification to the requirement.
- Moves the requirement from the open state to the review state by assigning the ownership to the director of software engineering.
- The marketing manager may decline a requirement in the open state and terminate the development process, thereby moving the requirement to the decline state with a proper explanation.

The following fields may be updated by the marketing manager, who is the owner of the requirement in the open state:

priority: Reevaluate the priority—high or normal—of this requirement.

severity: Assign a severity level—normal or critical—to the requirement.

decline_note: Give an explanation of the requirement if declined.

software_release: Suggest a preferred software release for the requirement.

marketing_justification: Provide a marketing justification for the requirement.

description: Describe the requirement, if there is any ambiguity.

note: Make any useful comments, if there is a need.

Review State The director of software engineering is the owner of the requirement in the review state. A requirement stays in the review state until it passes through the engineering process as explained in the following. The software engineering director reviews the requirement to understand it and estimate the time required to implement this. The director thus prepares a preliminary version of the functional specification for this requirement. This scheme provides a framework to map the requirement to the functional specification which is to be implemented. The director of software engineering can move the requirement from the review state to the assign state by changing the ownership to the marketing manager. Moreover, the director may decline this requirement if it is not possible to implement. The following fields may be updated by the director:

eng_comment: Comments generated during the review are noted in this field. The comments are useful in developing a functional specification, generating code, or performing unit-level testing.

time_to_implement: This field holds the estimated time in person-weeks to implement the requirement.

attachment: An analysis document, if there is any, including figures and descriptions that are likely to be useful in the future development of functional specifications.

functional_spec_title: The name of the functional specification that will be written for this requirement.

functional_spec_name: The functional specification filename.

functional_spec_version: The latest version number of the functional specification.

eng_assigned: Name of the engineer assigned by the director to review the requirement.

Assign State The marketing manager is the owner of the requirement in the assign state. A marketing manager assigns the requirement to a particular software release and moves the requirement to the commit state by changing the ownership to the program manager, who owns that particular software release. The marketing manager may decline the requirement and terminate the development process, thereby moving the requirement to the decline state. The following fields are updated by the marketing manager: decline_note and software_release. The former holds an explanation for declining, if it is moved to the decline state. On the other hand, if the requirement is moved to the commit state, the marketing manager updates the latter field to specify the software release in which the requirement will be available.

Commit State The program manager is the owner of the requirement in the commit state. The requirement stays in this state until it is committed to a software release. The program manager reviews all the requirements that are suggested to be in a particular release which is owned by him. The program manger may reassign a particular requirement to a different software release by consulting with the marketing manager, the software engineering director, and the customer. The requirement may be moved to the implement state by the program manager after it is committed to a particular software release. All the functional specifications should be frozen after a requirement is committed, that is, exited from the commit state. It is important to stabilize and freeze the functional specification for test design and development. The test engineers must complete the review of the requirement and the relevant functional specification from a testability point of view. Next, the test engineers can start designing and writing test cases for this requirement, as discussed in Section 11.6. The only field to be updated by the program manager, who is the owner of the requirement in the commit state, is committed_release. The field holds the release number for this requirement.

Implement State The director of software engineering is the owner of the requirement in the implement state. This state implies that the software engineering group is currently coding and unit testing the requirement. The director of software engineering may move the requirement from the implement state to the verification

TABLE 11.3 Engineering Change Document Information

EC number	A unique number
Requirement(s) affected	Requirement ID(s) and titles
Problem/issue description	Brief description of issue
Description of change required	Description of changes needed to original requirement description
Secondary technical impact	Description of impact EC will have on system
Customer impacts	Description of impact EC will have on end customer
Change recommended by	Name of the engineer(s)
Change approved by	Name of the approver(s)

state after the implementation is complete and the software is released for system testing, The director can assign an EC number and explain that the requirement is not doable in its current definition. The EC document is attached to the requirement definition. An outline of an EC document is given in Table 11.3. The director can also decline the requirement, if it is technically not possible to implement it, and move the requirement to the decline state. The following fields may be updated by the director, since he or she is the owner of a requirement in the implement state:

decline_note: An explanation of the reasons the requirement is declined if it is moved to the decline state.

ec_number: The EC document number.

attachment: The EC document.

Verification State The test manager is the owner of the requirement in the verification state. The test manager verifies the requirement and identifies one or more methods for assigning a test verdict: (i) testing, (ii) inspection, (iii) analysis, and (iv) demonstration. If testing is a method for verifying a requirement, then the test case identifiers and their results are provided. This information is extracted from the test factory discussed in Section 11.6. Inspection means review of the code. Analysis means mathematical and/or statistical analysis. Demonstration means observing the system in a live operation. A verdict is assigned to the requirement by providing the degree of compliance information: full compliance, partial compliance, or noncompliance. A testing note is included if a method other than testing is used for verification. The notes may contain an explanation of the analysis, inspection, or demonstration given to the customer.

The requirement may get an EC number from the test manager as a testing note. The EC document specifies any deficiency in the implementation of the requirement. A deviation or an error discovered at this stage can rarely be corrected. It is often necessary to negotiate with the customer through an EC document to the requirement. The program manager coordinates this negotiation activity with the customer. The test manager may decline the implementation with an EC number, explaining that the implementation is not conforming to the requirement, and

move to the decline state. The test manager may move the requirement to the closed state after it has been verified and the value of the verification_status field set to "passed." The following fields are updated by the test manager since he or she is the owner of the requirement at the verification state:

decline_note: The reasons to decline this requirement.

ec_number: An EC document number.

attachment: The EC document.

verification_method: Can take one of the four values from the set {Testing, Analysis, Demonstration, Inspection}.

verification_status: Can take one of the three values from the set {Passed, Failed, Incomplete}, indicating the final verification status of the requirement.

compliance: Can take one of the three values from the set {compliance, partial compliance, noncompliance}, which indicate the extent to which the software image complies with the requirements.

tc_id: The test case identifiers that cover this requirement.

tc_results: The test case results for the above tests. It can take on one of the five values from the set {Untested, Passed, Failed, Blocked, Invalid}. These values are extracted from the test factory database, which is discussed in Section 11.7.

defect_id: A defect identifier. If the tc_results field takes the value "failed," then the defect identifier is associated with the failed test case to indicate the defect that causes the failure. This value is extracted from test factory database.

testing_note: May hold an explanation of the analysis, inspection, or demonstration given to the end customer by the test engineer.

Closed State The requirement is moved to the closed state from the verification state by the test manager after it is verified.

Decline State In this state, the marketing department is the owner of the requirement. A requirement comes to this state because of some of the following reasons:

- The marketing department rejected the requirement.
- It is technically not possible to implement this requirement and, possibly, there is an associated EC number.
- The test manager declines the implementation with an EC number.

The marketing group may move the requirement to the submit state after reviewing it with the customer. The marketing manager may reduce the scope of the requirement after discussing it with the customer based on the EC information and resubmit the requirement by moving it to the submit state.

11.3 CHARACTERISTICS OF TESTABLE REQUIREMENTS

System-level tests are designed based on the requirements to be verified. A test engineer analyzes the requirement, the relevant functional specifications, and the standards to determine the testability of the requirement. The above task is performed in the commit state. Testability analysis means assessing the static behavioral characteristics of the requirement to reveal test objectives. One way to determine the requirement description is testable is as follows:

- Take the following requirement description: The system must perform X.
- Then encapsulate the requirement description to create a test objective: Verify that the system performs X correctly.
- Review this test objective by asking the question: Is it workable? In other words, find out if it is possible to execute it assuming that the system and the test environment are available.
- If the answer to the above question is yes, then the requirement description is clear and detailed for testing purpose. Otherwise, more work needs to be done to revise or supplement the requirement description.

As an example, let us consider the following requirement: The software image must be easy to upgrade/downgrade as the network grows. This requirement is too broad and vague to determine the objective of a test case. In other words, it is a poorly crafted requirement. One can restate the previous requirement as: The software image must be easy to upgrade/downgrade for 100 network elements. Then one can easily create a test objective: Verify that the software image can be upgraded/downgraded for 100 network elements. It takes time, clear thinking, and courage to change things.

In addition to the testability of the requirements, the following items must be analyzed by the system test engineers during the review:

- **Safety:** Have the safety-critical requirements [14] been identified? The safety-critical requirements specify what the system shall *not* do, including means for eliminating and controlling hazards and for limiting any damage in the case that a mishap occurs.
- **Security:** Have the security requirements [15], such as confidentiality, integrity, and availability, been identified?
- **Completeness:** Have all the essential items been completed? Have all possible situations been addressed by the requirements? Have all the irrelevant items been omitted?
- **Correctness:** Are the requirements understandable and have they been stated without error? Are there any incorrect items?
- **Consistency:** Are there any conflicting requirements?
- **Clarity:** Are the requirement materials and the statements in the document clear, useful, and relevant? Are the diagrams, graphs, and illustrations

clear? Have those been expressed using proper notation to be effective? Do those appear in proper places? Is the writing style clear?

- **Relevance:** Are the requirements pertinent to the subject? Are the requirements unnecessarily restrictive?
- **Feasibility:** Are the requirements implementable?
- **Verifiable:** Can tests be written to demonstrate conclusively and objectively that the requirements have been met? Can the functionality of the system be measured in some way that will assess the degree to which the requirements are met?
- **Traceable:** Can each requirement be traced to the functions and data related to it so that changes in a requirements can lead to easy reevaluation?

Functional Specification A functional specification provides:

i. A precise description of the major functions the system must fulfill the requirements, description of the implementation of the functions, and explanation of the technological risks involved

ii. External interfaces with other software modules

iii. Data flow such as flowcharts, transaction sequence diagrams, and FSMs describing the sequence of activities

iv. Fault handling, memory utilization and performance estimates

v. Any engineering limitation, that is, inferred requirements that will not be supported

vi. The command line interface or element management system to provision/configure the feature in order to invoke the software implementation related to this feature

Once again, the functional specification must be reviewed from the point of view of testability. The characteristics of testable functional specifications are outlined in Table 11.4. The functional specifications are more likely to be testable if they satisfy all the items in the Table 11.4. Common problems with functional specifications include lack of clarity, ambiguity, and inconsistency. The following are the objectives that are kept in mind while reviewing a functional specification [16]:

- **Achieving Requirements:** It is essential that the functional specification identifies the formal requirements to be achieved. One determines, by means of review, whether requirements have been addressed by the functional specification.
- **Correctness:** Whenever possible, the specification parts should be compared directly to an external reference for correctness.
- **Extensible:** The specification is designed to easily accommodate future extensions that can be clearly envisioned at the time of review.
- **Comprehensible:** The specification must be easily comprehensible. By the end of the review process, if the reviewers do not understand how the system works, the specification or its documentation is likely to be flawed.

TABLE 11.4 Characteristics of Testable Functional Specifications

Purpose, goals, and exception are clearly stated. Address the right objectives.
Contain the requirements and standards with which this document complies.
Clearly stated operating environment. For what hardware, OS, and software release the feature is targeted. Minimum hardware configuration that supports this application.
Clearly list the major functions which the system must perform.
Clearly define the success criteria which the system must fulfill to be effective.
Provide an understandable, organized, and maintainable model of the processes, and or data or objects, using a standard structured method and the principle of functional decomposition.
Use standard and clearly defined terminology (key words, glossary, syntax and semantics).
Display a heavy use of model and graphics (e.g., SDL-GR, finite-state model), not primarily English narrative.
Document the assumptions.
The document should have a natural structure/flow and with each atomic feature labeled with an identifier for easy cross-referring to specific test cases.
Should have standard exception handling procedure, consistent error messages, and on-line help functions.
External interfaces, such as CLI, MIBs, EMS, and Web interface are defined clearly.
Clearly stated possible trade-offs between speed, time, cost, and portability.
Performance requirements are defined, usually in terms of packets per second, transactions per second, recovery time, response time, or other such metrics.
Scaling limits and resource utilization (CPU utilization, memory utilization) are stated precisely.
Documentation of unit tests.

Note: SDL-GR, Specification and Description Language Graphical Representation

Such specifications and documentations need to be reworked to make them more comprehensible.

- **Necessity:** Each item in the document should be necessary.
- **Sufficiency:** The specification should be examined for missing or incomplete items. All functions must be described as well as important properties of input and output data such as volume and magnitude.
- **Implementable:** It is desirable to have a functional specification that is implementable within the given resource constraints that are available in the target environment such as hardware, processing power, memory, and network bandwidth. One should be able to implement a specification in a short period of time without a technological breakthrough.
- **Efficient:** The functional specification must optimize those parts of the solution that contribute most to the performance (or lack thereof) of the system. The reviewers have the discretion of rejecting the specification on the ground of ineffectiveness in specifying efficiency requirements.
- **Simplicity:** In general, it is easier to achieve and verify requirements stated in the form of simple functional specifications.

- **Reusable Components:** The specification should reuse existing components as much as possible and be modular enough that the common components can be extracted to be reused.
- **Consistency with Existing Components:** The general structure of the specification should be consistent with the choices made in the rest of the system. It should not require the design paradigm of a system to be changed for no compelling reason.
- **Limitations:** The limitations should be realistic and consistent with the requirements.

11.4 TEST OBJECTIVE IDENTIFICATION

The question "What do I test?" must be answered with another question: "What do I expect the system to do?" We cannot test the system comprehensively if we do not understand it. Therefore, the first step in identifying the test objective is to read, understand, and analyze the functional specification. It is essential to have a background familiarity with the subject area, the goals of the system, business processes, and system users for a successful analysis.

Let us consider our previously revised requirement: The software image must be easy to upgrade/ downgrade for 100 network elements. The test engineer needs to ask one question: What do I need to know to develop a comprehensive set of test objectives for the above requirement? An inquisitive test engineer may also ask the following questions:

- Do we have to upgrade the software image sequentially on each of the network elements or at the same time on all the 100 elements? Or, do we proceed in a batch of 20 elements at a time?
- What is the source of the upgrade? Will the source be on an element management server?
- What does "easy" mean here? Does it refer to the length of time, say, 200 seconds, taken by the upgrade process?
- Can we have a mix of old and new software images on different network elements on the same network? In other words, is a new software image compatible with the old image?
- If we support old and new software images on different network elements on the same network, then the EMS should be capable of managing two versions of a software installed on the same network. Is it possible for an EMS to manage network elements running different software versions?
- To what release will the software be downgraded? Suppose the software image is upgraded to the nth release from the $(n-1)$th release. Now, if the software image is to be downgraded, to what release should it be downgraded—$(n-1)$th or $(n-2)$th release?

- While a system is being upgraded, do we need to observe the CPU utilization of the network elements and the EMS server? What is the expected CPU utilization?

We critically analyze requirements to extract the inferred requirements that are embedded in the requirements. An inferred requirement is one that a system is expected to support but is not explicitly stated. Inferred requirements need to be tested just like the explicitly stated requirements. As an example, let us consider the requirement that the system must be able to sort a list of items into a desired order. One obvious test objective is: Verify that the system can sort an unsorted list of items. However, there are several unstated requirements not being verified by the above test objective. Many more test objectives can be identified for the requirement:

- Verify that the system produces the sorted list of items when an already sorted list of items is given as input.
- Verify that the system produces the sorted list of items when a list of items with varying length is given as input.
- Verify that the number of output items is equal to the number of input items.
- Verify that the contents of the sorted output records are the same as the input record contents.
- Verify that the system produces an empty list of items when an empty list of items is given as input.
- Check the system behavior and the output list by giving an input list containing one or more empty (null) records.
- Verify that the system can sort a list containing a very large number of unsorted items.

The test objectives are put together to form a test group or a subgroup after they have been identified. A set of (sub)groups of test cases are logically combined to form a larger group. A hierarchical structure of test groups as shown in Figure 11.2 is called a test suite. It is necessary to identify the test groups based on test categories and refine the test groups into sets of test objectives. Individual test cases are created for each test objective within the subgroups; this is explained in the next section with an example. Test groups may be nested to an arbitrary depth. They may be used to aid system test planning and execution, which are discussed in Chapters 13 and 13, respectively.

11.5 EXAMPLE

The Frame Relay Forum (FRF) defines two kinds of frame relay (FR)/asynchronous transfer mode (ATM) interworking scenarios: network interworking [17] and service interworking [18]. These two interworking functions provide a means by which two technologies, namely ATM and FR, can interoperate. Simply stated, network

Figure 11.2 Test suite structure.

Interworking provides a transport between two FR devices (or entities). Service interworking enables an ATM user to transparently interwork with an FR user, and neither knows that the other end uses a different technology.

Suppose that one of the requirements is to support service interworking as described in FRF.8 [18] on a switch. The director of software engineering develops a functional specification after the requirements are approved by the marketing manager. The test group develops a test category based on the requirements and the functional specification. Since the actual functional specification is not available, we assume the following for simplicity:

- The term *FrAtm* refers to the software component that provides the FR-ATM permanent virtual connection (PVC) service interworking functionality.
- FrAtm supports a variety of ATM cell-based physical interfaces on the ATM side: OC3, E3, DS3, E1, and DS1.
- FrAtm supports a variety of frame-based physical interfaces on the FR side: V.11, V.35, DS1, E1, DS3, and E3.
- The subcomponents of FrAtm are as follows: local management interface (LMI) and data-link connection identifier (DLCI).
- FrAtm software components are being implemented on an FR and ATM switch. In other words, both FR and ATM functionality is available on the same switch.

Let us briefly analyze the service interworking functionality before we develop different categories of tests. Figure 11.3 illustrates the service interworking between FR and ATM. Service interworking applies when (i) an FR service user

Figure 11.3 Service interworking between FR and ATM services.

interworks with an ATM service user, (ii) the ATM service user performs no frame relaying specific functions, and (iii) the frame relaying service user performs no ATM service-specific functions. Broadband-customer premise equipment (B-CPE) has no knowledge that a distant device is attached to an FR network. As shown in Figure 11.3, an FR user sends traffic on a PVC through the FR network to an interworking function (IWF), which then maps it to an ATM PVC. The FR PVC address–ATM PVC address mapping and other options are configured by the network management system associated with the IWF. Again, the IWF can be extended to the networks as shown, but it is more likely to be integrated into the ATM network switch or FR switch. Note that there is always one ATM PVC per FR PVC in the case of service internetworking. The IWF can be explained using a protocol stack model described in Figure 11.3. This protocol stack uses a "null" service-specific convergence sublayer (SSCS) for describing the IWF. This SSCS provides interfaces using standard primitives to Q.922 DL core on one side and to AAL5 (ATM adaptation layer) CPCS (common part convergence sublayer) on the other side within the IWF. Figure 11.4 shows the transformation of FR to ATM cells.

Frame Formatting and Delimiting

- **FR to ATM:** The FR frame is mapped into AAL5 PDU; the frame flags, inserted zero bits, and CRC-16 are stripped. The Q922 frame header is removed and some of the fields of the header are mapped into the ATM cell header fields.
- **ATM to FR:** The message delineation provided by AAL5 is used to identify frame boundaries; insert zero bits, CRC-16, and flags. Protocol fields and functions of the ATM AAL5 PDU are translated into the protocol fields and function of the FR frame.

Figure 11.4 Transformation of FR to ATM cell.

Discard Eligibility and Cell Loss Priority Mapping

- **FR to ATM:** Mode 1 or mode 2 may be selected per PVC at subscription time; the default is mode 1 operation:

  Mode 1: The discard eligibility (DE) field in the Q.922 core frame shall be mapped to the ATM CLP field of every cell generated by the segmentation process of the AAL5 PDU containing the information of that frame.

  Mode 2: The ATM CLP of every cell generated by the segmentation process of the AAL5 PDU containing the information of that frame shall be set to a constant value (either 0 or 1) configured at the service subscription time.

- ATM to FR: Mode 1 or 2 may be selected per PVC at subscription time; the default is mode 1 operation:

  Mode 1: If one or more cells of a frame have their CLP fields set, the IWF shall set the DE field of the Q.922 core frame.

  Mode 2: The DE field of the Q.922 core frame shall be set to a constant value (either 0 or 1) configured at service subscription time.

Forward Congestion Indication Mapping

- FR to ATM: Mode 1 or 2 may be selected per PVC at subscription time; the default is mode 1 operation:

  Mode 1: The FECN (Forward Explicit Congestion Notification) field in the Q.922 core frame shall be mapped to the ATM EFCI (Explicit Forward Congestion Indication) field of every cell generated by the segmentation process of the AAL5 PDU containing the information of that frame.

  Mode 2: The FECN field in the Q.922 core frame shall not be mapped to the ATM EFCI field of cells generated by the segmentation process of the AAL5 PDU containing the information of that frame. The EFCI field is always set to "congestion not experienced."
- ATM to FR: If the EFCI field in the last cell of a segmented frame received is set to "congestion experienced," the IWF will set the FECN core frame to "congestion experienced."

Backward Congestion Indication Mapping

- FR to ATM: BECN (Backward Explicit Congestion Notification) is ignored.
- ATM to FR: BECN of the Q.922 core frame is set to 0.

Command/Response Field Mapping

- FR to ATM: The C/R bit of the Q.922 core frame is mapped to the least significant bit in the common part convergence sublayer user-to-user (CPCS_UU) field of the CPCS PDU.
- ATM to FR: The least significant bit in the CPCS_UU field of the CPCS PDU is mapped to the C/R bit of the Q.922 core frame.

DLCI Field Mapping There is a one-to-one mapping between the Q.922 DLCI and the VPI/VCI (Virtual Path Identifier/Virtual Circuit Identifier) field in the ATM cells. The mapping is defined when the PVC is established. The association may be arbitrary or systematic.

Traffic Management Frame relay quality-of-service (QoS) parameters (CIR, B_c, B_e) will be mapped to ATM QoS parameters (PCR, SCR, MBS) using method 1, one-to-one mapping of the ATM Forum B-ICI (BISDN-Inter Carrier Interface) specification, which is given in Table 11.5. The value of the frame size variable used in the calculations is configurable per PVC. Also configurable per PVC will be CDVT.

TABLE 11.5 Mapping of FR QoS Parameters to ATM QoS Parameters

$PCR_{0+1} = AR/8 \times [OHA(n)]$
$SCR_0 = CIR/8 \times [OHB(n)]$
$MBS_0 = [B_c/8 \times (1/(1 - CIR/AR)) + 1] \times [OHB(n)]$
$SCR_1 = EIR/8 \times [OHB(n)]$
$MBS_1 = [B_e/8 \times (1/(1 - EIR/AR)) + 1] \times [OHB(n)]$
$CDVT = 1/PCR_{0+1}$
where n - number of user information octets in frame
AR - access line rate
CIR - committed information rate, B_c/T (bits/s)
EIR - excess information rate, B_e/T (bits/s), $CIR + EIR < AR$
B_c - committed burst size (bits)
B_e - excess burst size (bits)
T - measurement interval
PCR - peak cell rate (cells/s)
SCR - sustained cell rate (cells/s)
MBS - maximum burst size (number of cells)
CDVT - cell delay variation tolerance
$|X|$ - smallest integer greater than or equal to X
$OHA(n) = |(n + h_1 + h_2)/48|/(n + h_1 + h_2)$, overhead factor for access rate (cells/byte)
h_1 = FR header size (octets), 2-, 3:, or 4-octet headers
h_2 = FR HDLC overhead size of CRC-16 and flags (4 octets)
$OHB(n) = |(n + h_1 + h_2)/48|/n$, overhead factor for committed/excess rate (cells/byte) and subscript $0+1$, 0, or 1 applied to PCR, SCR, or MBS implies the parameter values for $CLP = 0 + 1$ cell stream, $CLP = 0$ cell stream, and $CLP = 1$ cell stream, respectively

Note: This method characterizes FR traffic using three generic cell rate algorithms (GCRAs) described in Appendix B of the ATM Forum UNI specification, 1993. Frames are with n user interface bytes.

PVC Mapping

- FR to ATM: When the IWF is notified that an FR PVC changes state from active to inactive, it will begin sending AIS (Alarm Indication Signal) F5 OAM (Operation, Administration, and Management) cells on the corresponding ATM PVC. An AIS F5 OAM cell is sent once per second until the state of the PVC changes from inactive to active.
- ATM to FR: When the IWF is notified that an ATM PVC changes state from active to inactive, it will set the FR PVC status bit to inactive. *Note:* An ATM PVC is considered inactive if (i) AIS or RDI (Remote Defect Indication) OAM cells are received or (ii) the ILMI (Interim Local Management Interface) MIB variable atmVccOperStatus is either localDown or end2endDown or (iii) the ATM interface is down. ATM responds to received AIS cells by sending RDI cells.

Upper Layer User Protocol Encapsulation The network provider can configure one of the following two modes of operation for each pair of interoperable FR and ATM PVCs regarding upper protocol encapsulations:

- Mode 1—transparent mode: The upper layer encapsulation methods will not be mapped between ATM and FR standards.
- Mode 2—translation mode: Encapsulation methods for carrying multiple upper layer user protocols (e.g., LAN to LAN) over FR PVC and an ATM PVC conform to the standard RFC 1490 [19] and RFC 1483 [20], respectively. The IWF shall perform mapping between the two encapsulations due to the incompatibilities of the two methods. This mode supports the interworking of the internetworking (routed and/or bridged) protocol.

Fragmentation and Reassembly

- FR to ATM: When fragmented packets are received on an FR PVC by the IWF, reassembly should be performed while forwarding the assembled frame to the ATM PVC.
- ATM to FR: Fragmentation should be performed on the received CPCS PDU before forwarding them to the FR PVC if the CPCS PDU is greater than the maximum frame size supported on the FR PVC.

FR–ATM PVC Service Interworking Test Category The FR–ATM PVC service interworking requirement is divided into six main categories: (i) functionality, (ii) robustness, (iii) performance, (iv) stress, (v) load and stability, and (vi) regression. The complete structure of the test categories is given in Figure 11.5.

Functionality Tests Functionality tests are designed to verify the FrAtm software as thoroughly as possible over all the requirements specified in the FRF.8 document. This category has been further subdivided into six different functional subgroups. For each identified functional subgroup, test objectives are formed that adequately exercise the function. Within each subgroup test objectives are designed to cover the valid values. The following major functionalities of the FrAtm feature must be tested adequately: (1) configuration and monitoring tests, (2) traffic management tests, (3) congestion tests, (4) service interworking function (SIWF) translation mode mapping tests, (5) alarm tests, and (6) interface tests:

1. Configuration and Monitoring Tests: Tests are designed to verify all the configurable attributes of the FrAtm component using the CLI command. These configurable attributes are implementation dependent and should be defined in the functional specification. Configuration tests configure the FrAtm using CLI. Monitoring tests verify the ability to use the CLI to determine the status and functioning of FrAtm. Statistics counters are validated for accuracy by using known amounts of traffic.
2. Traffic Management Tests: Tests are designed to verify mapping between ATM traffic parameters PCR_{0+1}, SCR_{0+1}, and MBS and FR rate enforcement parameters CIR, B_c, and B_e. Mapping between these parameters is subject to engineering consideration and is largely dependent on the balance between acceptable loss probabilities and network costs.

Figure 11.5 FrAtm test suite structure.

3. Congestion Tests: Various tests are designed to verify the extent of the FrAtm congestion control mechanism. The test objectives of each test are as follows:
   - Verify that the long-term maximum rate for the committed burst is CIR.
   - Verify that the long-term maximum rate for the excess burst is EIR.
   - Verify that DE = 0 and DE = 1 frames are counted toward the committed burst and excess burst, respectively.
   - Verify that FrAtm sets the BECN bit on the FrAtm to signal local congestion to the FR user.
4. Service Interworking Function Translation Mode Mapping Tests: Tests are designed to verify the SIWF translation mode mapping from FR to ATM and vice versa with the following objectives:
   - Verify that for frames from FR to ATM the IWF replaces the RFC 1490 encapsulation with the RFC 1483 encapsulation header.
   - Verify that for frames from ATM to FR the IWF replaces the RFC 1483 encapsulation with the RFC 1490 encapsulation header.
   - In the FR-to-ATM direction, verify that the FECN bit on a given frame maps directly into the EFCI bit of every cell that constitutes a frame if the EFCI attribute is configured as “preserve”; otherwise, the EFCI bit of every cell generated is set to zero.

- In the ATM-to-FR direction, verify that the EFCI bit of the last cell constituting a given frame is mapped directly into the FECN bit of that frame.
- In the FR-to-ATM direction, verify that the DE bit of a given frame maps directly into the CLP bit of every cell that constitutes that frame if so configured.
- In the ATM-to-FR direction, verify that the CLP-to-DE mapping can be configured with values "preserve," "always 0," and "always 1."
- In the FR-to-ATM direction, verify that the command/response (C/R) bit of the FR frame is mapped directly to the least significant bit of UU data in the CPCS of AAL5 encapsulation.
- In the ATM-to-FR direction, verify that the least significant bit of the UU data in the CPCS is mapped directly to the C/R bit of the constituting FR frame.
- In the FR-to-ATM direction, verify that the IWF will send AIS F5 OAM cells on the ATM PVC when the FR PVC changes state from active to inactive. Also, verify that the OAM cells are sent once per second until the state of the PVC changes from inactive to active.
- In the ATM-to-FR direction, verify that the IWF will set the FR PVC status bit to inactive when the ATM PVC changes state from active to inactive. Verify that RDI cells are sent corresponding to the received AIS cells.
- In the FR-to-ATM direction, verify that the IWF reassembles the fragmented packets received on a FR PVC and forwards it to the ATM PVC.
- In the ATM-to-FR direction, verify that the fragmentation ID is performed by the IWF on the received CPCS PDU before forwarding them to the FR PVC. Ensure that the CPCS PDU size is greater than the maximum frame size supported on the FR PVC.

5. Alarm Tests: Tests are designed to make sure that various alarms are generated for the FR–ATM service. The test objectives of each test are as follows:

- Verify that FrAtm generates appropriate alarms per PVC for individual DLCI.
- Verify that FrAtm generates state change notification (SCN) alarms when the operational state changes. The state change can occur when the link comes up or goes down. The state change can also occur by locking and unlocking FrAtm.
- Verify that FrAtm generates alarms to indicate the receipt of unexpected messages.
- Verify that alarms are generated to indicate the failure to allocate storage to create the service or to create subcomponents such as DLCIs.

6. Interface Tests: Verify that FrAtm supports V.11, V35, DS1, E1, DS3, and E3 on the FR side and OC3, E3, and DS3 interfaces on the ATM side.

Robustness Tests One verifies the robustness of the FrAtm implementations to error situations:

FrAtm Tests: Verify that (i) the FrAtm software component can handle continuous lock/unlock without any crash, (ii) the DLCI software component can handle multiple lock/unlock commands generated at high rate, and (iii) the DLCI software component can handle lock/unlock while traffic is going through the switch without any crash.

Boundary Value Tests: Verify that FrAtm handles boundary values and values below and values above the valid boundary for a subset of configurable attributes. For example, if the maximum number of DLCI configurable is 1024, then try to configure 1025 DLCI subcomponents and verify that the 1025th DLCI subcomponent is not created.

Performance Tests Tests are designed to measure data delay and throughput across different interfaces on both the FR and the ATM sides. Tests are conducted to measure the delay and throughput over a full range of ATM interface cards and FR interface cards for frame sizes 64, 128, 256, 512, 1024, 2048, and 4096 bytes.

Stress Tests Tests are designed to observe and capture the behavior of the FR–ATM services under various types of load profiles. The test objectives of each test are as follows:

- Verify that FrAtm works without any hiccups when it is stressed for 48 hours or more while lock/unlock activities and data transfer activity are going on.
- Verify that FrAtm works without any hiccups when the maximum number of DLCIs on the FR side using one FR card and the maximum number of ATM VCCs (Virtual Channel Connection) are configured.

Load and Stability Tests Tests are designed to simulate a real customer configuration by sending real data traffic. For these tests, setting up the customer environment in the laboratory is challenging and expensive. An example of this test is to verify that a user file can be transferred correctly from an FR network to an ATM network by using the FR–ATM PVC service interworking feature.

Regression Tests New test cases are not designed; rather, tests are selected from the groups explained before. In our opinion, a subset of test cases from each subgroup may be selected as regression tests. In addition, test cases from FR and ATM must be selected and executed to ensure that the previously supported functionality works with the new software component FrAtm.

Figure 11.6 State transition diagram of a test case.

11.6 MODELING A TEST DESIGN PROCESS

Test objectives are identified from a requirement specification, and one test case is created for each test objective. Each test case is designed as a combination of modular components called test steps. Test cases are clearly specified so that testers can quickly understand, borrow, and reuse the test cases.

Figure 11.6 illustrates the life-cycle model of a test case in the form of a state transition diagram. The state transition model shows the different phases, or states, in the life cycle of a test case from its inception to its completion through the following states: create, draft, review, deleted, released, update, and deprecated. Certain actions are taken by the "owner" of the state, and the test case moves to a next state after the actions are completed. In the following, the states are explained one by one. One can easily implement a database of test cases using the test case schema shown in Table 11.6. We refer to such a database of test cases as a test factory.

Create State A test case is put in this initial state by its creator, called the owner or creator, who initiates the design of the test case. The creator initializes the following mandatory fields associated with the test case: requirement_ids, tc_id, tc_title, originator_group, creator, and test_category. The test case is expected to verify the requirements referred to in the requirement_ids field. The originator_-group is the group who found a need for the test. The creator may assign the test case to a specific test engineer, including himself, by filling out the eng_assigned field, and move the test case from the create to the draft state.

Draft State The owner of this state is the test group, that is, the system test team. In this state, the assigned test engineer enters the following information: tc_author, objective, setup, test_steps, cleanup, pf_criteria, candidate_for_automation, automation_priority. After completion of all the mandatory fields, the test engineer may reassign the test case to the creator to go through the test case. The

TABLE 11.6 Test Case Schema Summary

Field	Description
tc_id	Test case identifier assigned by test author (80 characters)
tc_title	Title of test case (120 characters)
creator	Name of person who created test case
status	Current state of the record: create, draft, review, released, update, deleted, and deprecated
owner	Current owner of test case
eng_assigned	Test engineer assigned to write test procedure
objective	Objective of test case (multiline string)
tc_author	Name of test case author (user name)
originator_group	Group that originates test (performance testing group, functional testing group, scaling testing group, etc.)
test_category	Test category name (performance, stress, interoperability, functionality, etc.)
setup	List of steps to perform prior to test
test_steps	List of test steps
cleanup	List of posttest activities
pf_criteria	List of pass/fail criteria
requirement_ids	List of references to requirements ID from requirement database
candidate_for_automation	If test can be/should be automated
automation_priority	Automation priority.
review_action	Action items from review meeting minutes
approver_names	List of approver names

test case stays in this state until it is walked through by the creator. After that, the creator may move the state from the draft state to the review state by entering all the approvers' names in the approver_names field.

Review and Deleted States The owner of the review state is the creator of the test case. The owner invites test engineers and developers to review and validate the test case. They ensure that the test case is executable, and the pass–fail criteria are clearly specified. Action items are created for the test case if any field needs a modification. Action items from a review meeting are entered in the review_actions field, and the action items are executed by the owner to effect changes to the test case. The test case moves to the released state after all the reviewers approve the changes. If the reviewers decide that this is not a valid test case or it is not executable, then the test case is moved to the deleted state. A review action item must tell to delete this test case for a test case to be deleted.

Released and Update States A test case in the released state is ready for execution, and it becomes a part of a test suite. On the other hand, a test case in the update state implies that it is in the process of being modified to

enhance its reusability, being fine tuned with respect to its pass–fail criteria, and/or having the detailed test procedure fixed. For example, a reusable test case should be parameterized rather than hard coded with data values. Moreover, a test case should be updated to adapt it to changes in system functionality or the environment. One can improve the repeatability of the test case so that others can quickly understand, borrow, and reuse it by moving a test case in the released–update loop a small number of times. Also, this provides the foundation and justification for the test case to be automated. A test case should be platform independent. If an update involves a small change, the test engineer may move the test case back to the released state after the fix. Otherwise, the test case is subject to a further review, which is achieved by moving it to the review state. A test case may be revised once every time it is executed.

Deprecated State An obsolete test case may be moved to a deprecated state. Ideally, if it has not been executed for a year, then the test case should be reviewed for its continued existence. A test case may become obsolete over time because of the following reasons. First, the functionality of the system being tested has much changed, and due to a lack of test case maintenance, a test case becomes obsolete. Second, as an old test case is updated, some of the requirements of the original test case may no longer be fulfilled. Third, reusability of test cases tend to degrade over time as the situation changes. This is especially true of test cases which are not designed with adequate attention to possible reuse. Finally, test cases may be carried forward carelessly long after their original justifications have disappeared. Nobody may know the original justification for a particular test case, so it continues to be used.

11.7 MODELING TEST RESULTS

A test suite schema can be used by a test manager to design a test suite after a test factory is created. A test suite schema, as shown in Table 11.7, is used to group test cases for testing a particular release. The schema requires a test suite ID, a title, an objective, and a list of test cases to be managed by the test suite. One also identifies the individual test cases to be executed (test cycles 1, 2, 3 and/or regression) and the requirements that the test cases satisfy. The idea here is to gather a selected number of released test cases and repackage them to form a test suite for a new project.

Test engineers concurrently execute test cases from a selected test suite on different test beds. The results of executing those test cases are recorded in the test factory database for gathering and analyzing test metrics. In a large, complex system with many defects, there are several possibilities of the result of a test execution, not merely passed or failed. Therefore, we model the results of test execution by using a state transition diagram as shown in Figure 11.7, and the corresponding schema is given in Table 11.8. Figure 11.7 illustrates a state diagram of a test case result starting from the untested state to four different states: passed, failed, blocked, and invalid.

TABLE 11.7 Test Suite Schema Summary

Field	Description
test_suite_id	Unique identifier assigned by originator
test_suite_title	Test suite title, one-line title for test suite
test_suite_objective	Objective of test suite, short description.
tests	Reference list of test case identifiers
test_id	Test case ID, selected
test_title	Test case title, read only, filled in when test case is created
test_category	Category name (performance, stress, interoperability, functionality, etc.)
tester	Engineer responsible for testing
sw_dev	Software developer responsible for this test case who will assists test engineer in execution of test case
priority	Priority of test case
requirement_ids	Requirement identifier, read only, filled in when test case is created
cycle 1–3	Check box to indicate test is cycle 1, 2, or 3 test case
regression	Check box to indicate test is regression test case

TABLE 11.8 Test Result Schema Summary

Field	Description
tc_id	Reference to test case identifier record
test_title	Test case title, read only, filled in when result is created
test_category	Test category name (performance, stress, interoperability, functionality, etc.)
status	State of test case result: passed, failed, blocked, invalid or untested; initially status of test case is "untested"
run date	Date test case was run
time	Time spent in executing test case
tester	Name of person who ran test
release	Software release (03.00.00)
build	Software integration number (1–100).
defect_ids	List of defects which cause test to fail; value can come from bug tracking database
test_suite	Test suite this results pertains to

The execution status of a test case is put in its initial state of untested after designing or selecting a test case. If the test case is not valid for the current software release, the test case result is moved to the invalid state. In the untested state, the test suite identifier is noted in a field called test_suite_id. The state of the test result, after execution of a test case is started, may change to one of the following states:

Figure 11.7 State transition diagram of test case result.

passed, failed, invalid, or blocked. A test engineer may move the test case result to the passed state from the untested state if the test case execution is complete and satisfies the pass criteria.

If the test execution is complete and satisfied the fail criteria, a test engineer moves the test result to the failed state from the untested state and associates the defect with the test case by initializing the defect_ids field. The test case must be reexecuted when a new build containing a fix for the defect is received. If the reexecution is complete and satisfies the pass criteria, the test result is moved to the passed state.

The test case result is moved to a blocked state if it is not possible to completely execute it. If known, the defect number that blocks the execution of the test case is recorded in the defect_ids field. The test case may be reexecuted when a new build addressing a blocked test case is received. If the execution is complete and satisfies the pass criteria, the test result is moved to the passed state. On the other hand, if it satisfies the fail criteria, the test result is moved to the failed state. If the execution is unsuccessful due to a new blocking defect, the test result remains in the blocked state and the new defect that blocked the test case is listed in the defect_ids field.

11.8 TEST DESIGN PREPAREDNESS METRICS

Management may be interested to know the progress, coverage, and productivity aspects of the test case preparation work being done by a team of test engineers. Such information lets them (i) know if a test project is progressing according to schedule and if more resources are required and (ii) plan their next project more accurately. The following metrics can be used to represent the level of preparedness of test design.

Preparation Status of Test Cases (PST): A test case can go through a number of phases, or states, such as draft and review, before it is released as a valid and useful test case. Thus, it is useful to periodically monitor the progress of test design by counting the test cases lying in different states of design—create, draft, review, released, and deleted. It is expected that all the planned test cases that are created for a particular project eventually move to the released state before the start of test execution.

Average Time Spent (ATS) in Test Case Design: It is useful to know the amount of time it takes for a test case to move from its initial conception, that is, create state, to when it is considered to be usable, that is, released state. This metric is useful in allocating time to the test preparation activity in a subsequent test project. Hence, it is useful in test planning.

Number of Available Test (NAT) Cases: This is the number of test cases in the released state from existing projects. Some of these test cases are selected for regression testing in the current test project.

Number of Planned Test (NPT) Cases: This is the number of test cases that are in a test suite and ready for execution at the start of system testing. This metric is useful in scheduling test execution. As testing continues, new, unplanned test cases may be required to be designed. A large number of new test cases compared to NPT suggest that initial planning was not accurate.

Coverage of a Test Suite (CTS): This metric gives the fraction of all requirements covered by a selected number of test cases or a complete test suite. The CTS is a measure of the number of test cases needed to be selected or designed to have good coverage of system requirements.

11.9 TEST CASE DESIGN EFFECTIVENESS

The objectives of the test case design effectiveness metric is to (i) measure the "defect revealing ability" of the test suite and (ii) use the metric to improve the test design process. During system-level testing, defects are revealed due to the execution of planned test cases. In addition to these defects, new defects are found during testing for which no test cases had been planned. For these new defects, new test cases are designed, which are called test case escaped (TCE). Test escapes occur because of deficiencies in the test design process. This happens because the test engineers get new ideas while executing the planned test cases. A metric commonly used in the industry to measure test case design effectiveness is the test case design yield (TCDY), defined as

$$\text{TCDY} = \frac{\text{NPT}}{\text{NPT} + \text{number of TCE}} \times 100\%$$

The TCDY is also used to measure the effectiveness of a particular testing phase. For example, the system integration manager may want to know the TCDY value for his or her system integration testing.

11.10 SUMMARY

This chapter began with an introduction to six factors that are taken into consideration during the design of test cases: coverage metrics, effectiveness, productivity, validation, maintenance, and user skill. Then we discussed the reasons to consider these factors in designing system tests.

Next, we discussed the requirement identification process. We provided a state transition model to track the individual requirements as they flow through the organization. At each state of the transition model, certain actions are taken by the owner. The requirement is moved to a new state after the actions are completed. We presented a requirement schema that can be used to generate a traceability matrix. A traceability matrix finds two applications: (i) identifying and tracking the functional coverage of a test suite and (ii) identifying which test cases must be exercised or updated when a system undergoes a change. A traceability matrix allows one to find a mapping between requirements and test cases as follows:

- From a requirement to a functional specification to specific tests which exercise them
- From each test case back to the requirements and functional specifications

Next, we examined the characteristics of testable requirements and functional specification. We provided techniques to identify test objectives from the requirements and functional specifications. A requirement must be analyzed to extract the inferred requirements that are embedded in the requirement. An inferred requirement is anything that a system is expected to do but not explicitly stated. Finally, we showed the ways to create a hierarchical structure of test groups, called a test suite. As an example, we illustrated in detail the design of a system test suite for the FR–ATM serviceinterworking protocol.

Next, we provided a state transition model of a test case from its inception to completion. At each state of the transition model, certain actions are taken by the owner. The test case is moved to a new state after the actions are completed. We presented a test case schema to create a test factory that can be used to design a test suite and monitor the test case preparedness metrics.

Finally, we provided a metric used in the industry to measure test case design effectiveness, known as the test case design yield. The objectives of the test case effectiveness metric are to (i) measure the "defect revealing ability" of the test suite and (ii) use the metric to improve the test design process.

11.11 LITERATURE REVIEW

A good discussion of testing requirements is presented in the article by Suzanne Robertson, entitled "An Early Start to Testing: How to Test Requirements," which is reprinted in Appendix A of the book by E. Dustin, J. Rashka, and J. Paul (*Automated Software Testing: Introduction, Management, and Performance*, Addison-Wesley,

Reading, MA, 1999). The author describes a set of requirement tests that cover relevance, coherency, traceability, completeness, and other qualities that successful requirements must have.

The requirements traceability reference models described in the article by B. Ramesh and M. Jarke ("Towards Reference Models for Requirements Traceability," *IEEE Transactions on Software Engineering*, Vol.27, No.1, January 2001, pp. 58–93) is based on several empirical studies. Data collection for the work spanned a period of over three years. The main study comprised 30 focus group discussions in 26 organizations conducted in a wide variety of industries, including defense, aerospace, pharmaceuticals, electronics, and telecommunications. The participants had an average of 15.5 years of experience in several key areas of systems development, including software engineering, requirements management, software testing, system integration, systems analysis, maintenance, and software implementation. The participants are categorized into two distinct groups with respect to their traceability practices. These groups are referred to as low-end and high-end traceability users. Separate reference models for these two groups are discussed in the article.

The IEEE standard 829-1983 (*IEEE Standard for Software Test Documentation: IEEE/ANSI Standard*) provides templates for test case specifications and test procedure specifications. A test case specification consists of the following components: test case specification identifier, test items, input specification, output specification, special environment needs, special procedural requirements, and test case dependencies. A test procedure specification consists of the following components: test procedure specification identifier, purpose, specific requirements, and procedure steps.

Another approach to measuring test case effectiveness has been proposed by Yuri Chernak ("Validating and Improving Test-Case Effectiveness," *IEEE Software*, Vol. 18, No. 1, January/February 2001, pp. 81–86). The effectiveness metric called test case escaped is defined as

$$\text{TCE} = \frac{\text{number of defects found by test cases}}{\text{total number of defects}} \times 100\%$$

The total number of defects in the above equation is the sum of the defects found by the test cases and the defects found by chance, which the author calls "side effects." Chernak illustrates his methodology with a client–server application using a baseline TCE value (<75 for this case) to evaluate test case effectiveness and make test process improvements. Incomplete test design and incomplete functional specifications were found to be the main causes of test escapes.

Formal verification of test cases is presented in the article by K. Naik and B. Sarikaya, "Test Case Verification by Model Checking," *Formal Methods in Systems Design*, Vol. 2, June 1993, pp. 277–321. The authors identified four classes of safety properties and one liveness property and expressed them as formulas in branching time temporal logic. They presented a model checking algorithm to verify those properties.

REFERENCES

1. H. S. Wang, S. R. Hsu, and J. C. Lin. A generalized optimal path-selection model for the structure program testing. *Journal of Systems and Software*, July, 1989, pp. 55–62.
2. O. Vinter. *From Problem Reports to Better Products*. In *Improving Software Organizations: From Principles to Practice*, L. Mathiassen, J. Pries-Heje, and O. Ngwenyama, Eds. Addison-Wesley, Reading, MA, 2002, Chapter 8.
3. M. Glinz and R. J. Wieringa. Stakeholders in Requirements Engineering. *IEEE Software*, March/April 2007, pp. 18–20.
4. N. Serrano and I. Ciordia. Bugzilla, ITtracker, and Other Bug Trackers. *IEEE Software*, March/April 2005, pp. 11–13.
5. D. Jacobs. Requirements Engineering So Things Don't Get Ugly. *Crosstalk, the Journal of Defense Software Engineering*, October 2004, pp. 19–25.
6. P. Carlshamre and B. Regnell. Requirements Lifecycle Management and Release Planning in Market-Driven Requirements Engineering Process. In *Proceedings of the 11th International Workshop on Database and Expert System Applications*, Greenwich, UK, IEEE Computer Society Press, Piscataway, NJ, September 2000, pp. 961–966.
7. J. Bach. Risk and Requirements-Based Testing. *IEEE Computer*, June 1999, pp. 113–114.
8. B. J. Brown. Assurance of Quality Software. SEI Curriculum Module, SEI-CM-7-1.1, July 1987.
9. M. Lehman and J. Ramil. Software Evolution—Background, Theory, Practice. *Information Processing Letters*, October 2003, pp. 33–44.
10. O. C. Z. Gotel and A. C. W. Finkelstein. An Analysis of the Requirements Traceability Problem. In *Proceedings of First International Conference on Requirements Engineering*, Colorado Spring, IEEE Computer Society Press, Piscataway, NJ, 1994, pp. 94–101.
11. J. Karlsson and K. Ryan. A Cost-Value Approach for Prioritizing Requirements. *IEEE Software*, September/October 1997, pp. 67–74.
12. E. Carmel and S. Becker. A Process Model for Packaged Software Development. *IEEE Transactions on Engineering Management*, February 1995, pp. 50–61.
13. P. Soffer, L. Goldin, and T. Kuflik. A Unified RE Approach for Software Product Evolution. In *Proceedings of SREP'05*, Paris, Printed in Ireland by the University of Limerick, August 2005, pp. 200–210.
14. C. W. Johnson and C. M. Holloway. Questioning the Role of Requirements Engineering in the Causes of Safety-Critical Software Failures. In *Proceedings of the IET 1st International Conference on System Safety*, IEEE Computer Society Press, Piscataway, NJ, London, June 2006.
15. S. L. Pfleeger. A Framework for Security Requirements. *Computers & Security*, October 1991, pp. 515–523.
16. W. E. Howden. Validation of Scientific Programs. *ACM Computing Surveys*, June 1982, pp. 193–227.
17. Frame Relay/ATM PVC Network Inter-Working-Implementation Agreement, FRF.5. Frame Relay Forum, December 1994. Available at http://www.ipmplsforum.org/Approved/FRF.5/FRF5.TOC.shtml.
18. Frame Relay/ATM PVC Service Inter-Working-Implementation Agreement, FRF.8. Frame Relay Forum, April 1995. Available at http://www.ipmplsforum.org/Approved/FRF.8/FRF8.TOC.shtml.
19. Internet Engineering Task Force (IETF). Multiprotocol Interconnect over Frame Relay, RFC 1490. IETF, July 1993. Available at http://www.apps.ietf.org/rfc/rfc1490.html.
20. Internet Engineering Task Force (IETF). Multiprotocol Encapsulation over AAL, RFC 1483. IETF, July 1993. Available at http://www.apps.ietf.org/rfc/rfc1483.html.

Exercises

1. Explain the difference between coverage metrics and traceability matrix.
2. Explain the difference between requirement testability and software testability.

3. Justify the statement that software testability and fault tolerant are opposites to each other, and attaining both at the same time for the same piece of software is not feasible. When do you want high testability and when don't you during the life cycle of a critical software system? Justify your answer.
4. What are the differences between software testability and reliability? What is more important in a software: high testability or high reliability? Justify your answer.
5. In a software test project, the number of unit-, integration-, and system-level test cases specified are 250, 175, and 235, respectively. The number of test cases added during the unit, integration, and system testing phases are 75, 60, and 35, respectively. Calculate the TCDY for unit, integration, and system testing phases.
6. Prepare a checklist of items that will serve as the focal point for reviewing test cases.
7. Under what circumstances can the execution result of a test case be declared as blocked?
8. Implement the requirement model discussed in this chapter.
9. Implement the test factory model discussed in this chapter.
10. For your current test project, develop the test suite hierarchy and identify a test objective for each test case within each test (sub)group.
11. Develop detailed test cases using the schema defined in Table 11.6 as a template for the test objectives you identified in the previous exercise.

CHAPTER 12

System Test Planning and Automation

When planning for a year, plant corn. When planning for a decade, plant trees. When planning for life, train and educate people.
— *Chinese proverb*

12.1 STRUCTURE OF A SYSTEM TEST PLAN

A good plan for performing system testing is the cornerstone of a successful software project. In the absence of a good plan it is highly unlikely that the desired level of system testing is performed within the stipulated time and without overusing resources such as manpower and money. Moreover, in the absence of a good plan, it is highly likely that a low-quality product is delivered even at a higher cost and later than the expected date.

The purpose of system test planning, or simply test planning, is to get ready and organized for test execution. Starting a system test in an ad hoc way, after all the modules are checked in to the version control system, is ineffective. Working under deadline pressure, people, that is, test engineers, have a tendency to take shortcuts and to "just get the job done," which leads to the shipment of a highly defective product. Consequently, the customer support group of the organization has to spend a lot of time in dealing with unsatisfied customers and be forced to release several patches to demanding customers. Test planning is essential in order to complete system testing and ship quality product to the market on schedule. Planning for system testing is part of overall planning for a software project. It provides the framework, scope, resources, schedule, and budget for the system testing part of the project.

Test efficiency can be monitored and improved, and unnecessary delay can be avoided with a good test plan. The purpose of a system test plan is summarized as follows:

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

TABLE 12.1 Outline of System Test Plan

1. Introduction
2. Feature description
3. Assumptions
4. Test approach
5. Test suite structure
6. Test environment
7. Test execution strategy
8. Test effort estimation
9. Scheduling and milestones

- It provides guidance for the executive management to support the test project, thereby allowing them to release the necessary resources to perform the test activity.
- It establishes the foundation of the system testing part of the overall software project.
- It provides assurance of test coverage by creating a requirement traceability matrix.
- It outlines an orderly schedule of events and test milestones that are tracked.
- It specifies the personnel, financial, equipment, and facility resources required to support the system testing part of a software project.

The activity of planning for system testing combines two tasks: *research* and *estimation*. Research allows us to define the scope of the test effort and resources already available in-house. Each major functional test suite consisting of test objectives can be described in a bounded fashion using the system requirements and functional specification as references. The reader may refer to Chapter 11 for a discussion of the test suites described in a bounded fashion.

A system test plan is outlined in Table 12.1. We explain the ways one can create a test plan. The test plan is released for review and approval after the author, that is, the leader of the system test group, completes it with all the pertinent details. The review team must include software and hardware development staff, customer support group members, system test team members, and the project manager responsible for the project. The author(s) should solicit reviews of the test plan and ask for comments prior to the meeting. The comments can then be addressed at the review meeting. The system test plan must be completed before the software project is committed.

12.2 INTRODUCTION AND FEATURE DESCRIPTION

The introduction section of the system test plan describes the structure and the objective of the test plan. This section includes (i) test project name, (ii) revision

history, (iii) terminology and definitions, (iv) names of the approvers and the date of approval, (v) references, and (vi) summary of the rest of the test plan.

The feature description section summarizes the system features that will be tested during the execution of this test plan. In other words, a high-level description of the functionalities of the system is presented in this section. The feature description for the FR/ATM service interworking example is discussed in Chapter 11.

12.3 ASSUMPTIONS

The assumptions section describes the areas for which test cases will not be designed in this plan due to several seasons. First, the necessary equipment to carry out scalability testing may not be available. Second, it may not be possible to procure third-party equipment in time to conduct interoperability testing. Finally, it may not be possible to conduct compliance tests for regulatory bodies and environment tests in the laboratory. These assumptions must be considered while reviewing a system test plan.

12.4 TEST APPROACH

The overall test approach is an important aspect of a testing project which consists of the following:

- Lessons learned from past test projects are useful in focusing on problematic areas in the testing process. Issues discovered by customers that were not caught during system testing in past projects are discussed. For example, if a customer encountered a memory leak in the system in the past project, action must be taken to flush out any memory leak early in system test execution. An appropriate response may be needed to develop effective test cases by using sophisticated software tools such as memory leak detectors from the market.
- If there are any outstanding issues that need to be tested differently, for example, requiring specific hardware and software configuration, then these issues need to be discussed here.
- A test automation strategy for writing scripts is a topic of discussion.
- Test cases should be identified in the test factory that can be reused in this test plan.
- Outline should be prepared of the tools, formats, and organizing scheme, such as a traceability matrix, that will be used and followed during the test project.
- Finally, the first level of test categories, which are likely to apply to the present situation and as discussed in Chapter 8, are identified.

12.5 TEST SUITE STRUCTURE

Detail test groups and subgroups are outlined in the test suite structure section based on the test categories identified in the test approach section. Test objectives are created for each test group and subgroup based on the system requirements and functional specification discussed in Chapter 11. A traceability matrix is generated to make an association between requirements and test objectives to provide the highest degree of confidence. Note that at this stage only test suites along with the test objectives are identified but not the detail test cases. Identification of test objectives provides a clue to the total number of new test cases that need to be developed for this project. If some existing test cases, automated or manual, need to be run as regression tests, those test cases must be included in the test suite. This information is useful in estimating the time required to create the test cases and to execute the test suite, as discussed in Section 12.8. The test suite structure for the FR/ATM service interworking example is discussed in Chapter 11.

12.6 TEST ENVIRONMENT

It is necessary to plan for and design the test environment, also called a *test bed* or a *test laboratory*, to make the execution of system-level test cases effective. It is a challenge to design test environments which contain only a small proportion of the equipment and facilities used in actual operation because of budget limitations. The central idea in using a small proportion of equipment and facilities is *to do more with less*. The objective here is to achieve effective testing whereby most of the defects in the system are revealed by utilizing a limited quantity of resources. One has to be innovative in designing a test bed such that the test objectives are fulfilled by executing the test cases on the test bed. One must consider alternatives, or, at least, a scaled-down version of the deployment environment from the standpoint of cost-effectiveness. Efforts must be made to create a deployment environment by using simulators, emulators, and third-party traffic generation tools. Such tools are found to be useful in conducting scalability, performance, load, and stress testing. An emulator may not be an ideal substitute for real equipment, but as long as it satisfies the purpose, it is worth investing in. We explained in Chapter 8 that there are different categories of test cases designed at the system level. Therefore, multiple test environments are constructed in practice for the following reasons:

- To run scalability tests, we need more resources than needed to run functional tests.
- Multiple test beds are required to reduce the length of system testing time.

Preparing for a test environment is a great challenge in test planning. This is especially true in testing distributed systems and computer networks where a variety of equipment are connected through communication protocols. For example, such equipment includes user computers, servers, routers, base stations in a wireless

network, authentication servers, and billing servers. It may take several months to set up an effective test bed for large, complex systems. It requires careful planning, procurement of test equipment, and installation of the equipment in a test facility different from the software development facilities so that system testing is performed effectively. Developers have their own test environments to perform unit tests and integration tests. However, a separate, dedicated system test laboratory, different from the ones used in unit and integration testing, is essential for the following reasons:

- Test engineers need to have the ability to reconfigure a test environment.
- Test activities need not interfere with development activities or live operations.
- Increased productivity is achieved by having a dedicated test laboratory.

A central issue concerning setting up of a system test laboratory is the justification to procure the equipment. Note that there is a need to justify each item to be procured. A good justification for procuring the equipment can be made by answering the following questions:

- Why do we need this equipment?
- What will be the impact of not having this equipment?
- Is there an alternative to procuring this equipment?

The technical leader of the system test engineering group should gather some facts and perform some preparation activities in order to get answers to these questions. The following items are part of a good fact gathering process:

- Reviews the system requirements and the functional specification
- Participates in the review processes to better understand the system and raise potential concerns related to the migration of the system from the development environment to the deployment environment
- Documents his or her findings

The following preparation activities are conducted to support the development of a system test bed:

- Obtain information about the customer deployment architecture, including hardware, software, and their manufacturers. For example, the real deployment network diagram along with the software configuration is useful in designing a scaled-down version of the system in the laboratory. The manufacturer names will be handy in procuring the exact equipment for interoperability and compatibility testing.
- Obtain a list of third-party products to be integrated with the SUT. Identification of the external products is important because of the need for performing interoperability testing.
- List third-party test tools to be used to monitor, simulate, and/or generate real traffic. This traffic will be used as input to the SUT.

- Identify the third-party software tools to be used under licenses.
- Identify the hardware equipment necessary to support special features specified in the requirement/test objectives, such as high availability and backup/recovery exercises within the test environment.
- List the number of hardware copies to carry out system testing if the project involves new hardware.
- Analyze the functional, performance, stress, load, and scalability test objectives to identify elements of the test environment that will be required to support those tests.
- Identify the security requirements for the test environment. Ensure that the security test cases can be executed using the test environment and an intruder cannot disrupt the stress and stability tests that may be running overnight or over the weekends.
- List the small, but necessary networking gears that may be required to set up the test laboratory, such as switches, terminal servers, hubs, attenuators, splitters, personal computers, servers, and different kinds and sizes of cables to interconnect these gears.
- List any other accessories required to facilitate system testing, such as racks, vehicles, and special shielding to prevent radiation.

After the above fact gathering and researching activities, the team leader develops a schematic diagram of one or more test beds in terms of the following two items:

- High-level graphic layout of test architectures
- Table of types of equipment, their quantities, and their descriptions to support the test architecture

The equipment list should be reviewed to determine the equipment available in-house and those that need to be procured. The list of equipment to be procured constitutes a test equipment purchase list. The list needs to include quantities required, unit price information, including maintenance cost, and justification, as outlined in the template shown in Table 12.2. The team leader must specify, for each item in the justification column, the justification for the item and the impact it will have on system testing in terms of quality and a time-to-market schedule. The team leader may obtain a quote from the suppliers to get an accurate unit price. The test team leader needs to keep track of the equipment received and their installation after the budget is

TABLE 12.2 Equipment Needed to be Procured

Equipment to Procure	Quantity	Unit Price	Maintenance Cost	Justification

approved and the orders are placed. The leader needs to ensure that these activities are on track to meet the overall software project schedule.

12.7 TEST EXECUTION STRATEGY

It is important to have a proper game plan ready before the system test group receives a software system for the first time for performing system testing.

The game plan is in the form of a system test execution strategy to carry out the task [1]. We address the following concerns by putting a game plan in place before initiating system testing:

- How many times are the test cases executed and when?
- What does one do with the failed test cases?
- What happens when too many test cases fail?
- In what order are the test cases executed?
- Which test cases are to be run just before ending the system testing phase?

In the absence of a good system test execution strategy, it is highly unlikely that the desired system-level test is performed within the given time and without overusing resources.

Let us consider a simple execution strategy. Assume that a system test team has designed a test suite T for a system S. Due to the detection and removal of defects and the possibility of causing more defects, S is an evolving system that can be characterized by a sequence of builds $B_0, B_1, B_2, \ldots, B_k$, where each build is expected to have fewer defects than its immediate predecessor. A simple strategy to perform system testing is as follows: Run the test set T on B_0 to detect a set of defects D_0; let B_1 be a build obtained from B_0 by fixing defects D_0; run T on B_1 to detect a set of defects D_1; let B_2 be a build obtained from B_1 by fixing defects D_1; and this process can continue until the quality level of the system is at least the desired quality level. One can adopt such a simple execution strategy if all the test cases in T can be independently executed, that is, no test case is blocked due to a defect in the system. If all the reported defects are immediately fixed and no new defects are introduced, then system testing can be considered to be over by running T just twice: once on B_0 to detect D_0 and a second time on B_1 for confirmation. In the above discussion, we refer to running T, or a subset thereof, on a new build of a software SUT as a system test cycle.

However, the processes of test execution, defect detection, and fixing defects are intricately intertwined. The key characteristics of those processes are as follows:

- Some test cases cannot be executed unless certain defects are detected and fixed.
- A programmer may introduce new defects while fixing one defect, which may not be successful.
- The development team releases a new build for system testing by working on a subset of the reported defects, rather than all the defects.

- It is a waste of resources to run the entire test set T on a build if too many test cases fail.
- As the number of reported defects significantly reduces over a few iterations of testing, it is not necessary to run the entire test set T for regression testing—a carefully chosen subset of T is expected to be adequate.

Therefore, the simple test execution strategy explained above is not practical. An effective and efficient execution strategy must take into account those characteristics of test execution, defect detection, and defect removal.

We present a process model for a metric-based, multicycle test execution strategy which clearly defines an entry criterion to start system testing, how to prioritize execution of test cases, when to move from one test cycle to the next, when to rerun failed test cases, when to suspend a test cycle and initiate root cause analysis, and how to choose a subset of test cases in the final test cycle for regression testing. We characterize each test cycle by a set of six parameters: goals, assumptions, test execution, revert and extension criteria, action, and exit criteria. The idea behind a parameterized test cycle is twofold: (i) the quality level of a system is raised in a measurable, controlled, and incremental manner possibly from the initial low level to the final high level and (ii) the process of system testing is broken down into a sequence of repeatable steps. The parameter values in each test cycle allow the system test leader to effectively control the progress of system testing so that the system moves one step closer to the desired, final quality level.

12.7.1 Multicycle System Test Strategy

The idea of a multicycle test execution strategy with three cycles, for example, is illustrated in Figure 12.1*a*. Figure 12.1*b* shows how the quality of a product increases from test cycle to test cycle. In one test cycle, all the test cases in the entire test suite T, or a carefully chosen subset of T, are executed at least once. It is expected that a number of defects are fixed within the life-span of a test cycle so that the quality level of the system is raised by a certain amount, as illustrated in Figure 12.1*b*. The need for two root cause analyses, one by the development team and one by the system test team, is explained later in this section. The total number of test cycles to be used in individual test projects is a matter of a management decision based on the level of the delivered quality one wants to achieve, the extent to which a test suite covers the requirements of the system, and the effectiveness of defect fixes.

12.7.2 Characterization of Test Cycles

Each test cycle is characterized by a set of six parameters: goals, assumptions, test execution, actions, revert and extension criteria, and exit criteria. Appropriate values must be assigned to these parameters while preparing a concrete test execution strategy. In Section 12.7.6, we give real-life, representative instances of these parameters for three test cycles, where the third test cycle is the final one.

Figure 12.1 Concept of cycle-based test execution strategy.

Goals A system test team sets its own goals to be achieved in each test cycle. These goals are *ideal* in the sense that these are very high standard goals, and consequently, may not be achievable in a given test cycle. The motivation for setting ideal goals is that it is not known what weaker goal is desirable and achievable. Setting a weaker goal has the danger of discouraging the development and test teams from achieving stronger goals. Therefore, we aim to achieve ideal goals in each test cycle, but we are ready to exit from a test cycle even if the goals are not fully achieved. An exit criterion specifies the termination of a test cycle. Goals are specified in terms of the number of test cases to pass within a test cycle. Though goals can also be specified in terms of the number of defects remaining in the software product at the end of each system test cycle [2], predicting the remaining number of defects is a very difficult task [3].

Remark 1. The goal of a test cycle may be changed due to unforeseen circumstances. For example, a testing cycle is terminated prematurely because of the poor quality of the software. However, the system test group may proceed to complete the system testing cycle to gain hands-on experience and to be able to identify needs for additional test cases. In this situation, the specified goal of the test cycle can be changed to understanding and improving the effectiveness of the test cases.

Assumptions A SIT group and a system test group work in an autonomous but cooperative manner in an organization. The SIT group produces builds almost regularly, whereas the system test group selects builds for testing on its own schedule, say, once every two weeks. As a consequence, not all builds are accepted for system testing. In the assumption part, the system test group records its own assumption about when to select builds for system testing. Appropriate assumptions must be made in order to achieve the goals. For example, consider a goal that 90% of test cases should pass in a five-week test cycle and another assumption that the system test team will accept just one new build from the SIT group during the test cycle. Assume that at the end of the fourth week of testing the total number of passed, failed, and blocked test cases are at 70, 7, and 23%, respectively. A test case is said to be blocked if a defect prevents the execution of the test case. Therefore, the goal cannot be achieved by the end of the fifth week. The team can accept builds from the SIT group on a daily basis to avoid this situation. One has to spend a significant amount of time in setting up the test beds by installing a build every day to pick a daily build for system testing. Therefore, a test manager must make an appropriate assumption to allow them to achieve the desired goal.

Test Execution Test cases are simultaneously executed in multiple test environments by using the concept of test prioritization. Prioritization of test execution changes between test cycles. This offers a trade-off between prioritizing test cases just once for the entire duration of system testing and prioritizing test cases from build to build. Some basic ideas in test prioritization are as follows: (i) test cases that exercise basic functionalities have higher priority than the rest in the first test cycle; (ii) test cases that have failed in one test cycle have a higher priority in the following test cycle; and (iii) test cases in certain test groups have higher priority than the others. There are three differences between test prioritization studied in the past [4–6] and a new approach presented in this book, which are as follows:

1. Test prioritization has been considered largely for regression testing. In our test strategy, test cases are prioritized for development testing as well. Test cases are prioritized during system testing in order to exercise the likely vulnerable part of the code early so that developers will have more time to debug and fix the defects, if there are any. For example, if the defects are revealed toward the end of a test cycle, the developers may not have enough time to fix the defects; moreover, the system test team may not be able to validate the fixes in a very short time. Srivastava and Thiagarajan [7] present a test prioritization technique for regression testing during development. Their test prioritization algorithm is based on block-level differences between the compiled binary code of the previous version of a system and its current version to be tested. The prioritization algorithm tries to pick a test from the remaining tests that will cover the maximum number of the remaining impacted, that is, new or modified, blocks.

2. Test prioritization produces a single sequence of ordered test cases, while the need and opportunity for producing smaller, concurrent test execution sequences have been ignored. In our approach, simultaneity of test executions and *importance* of test cases have been factored into test prioritization. Multiple, concurrent test environments allow us to shorten the length of system testing. Kapfhammer [8] proposes the idea of running regression tests on multiple communicating machines.
3. It is highly unlikely that tens of engineers working on a test project have identical interest and expertise in all aspects of a system. For example, one person might be an expert in performance evaluation, whereas another person may be useful in user-interface testing. Test cases falling in their respective categories are separately prioritized.

Revert and Extension Criteria Each test cycle must be completed within a stipulated time. However, the duration of a test cycle may be extended under certain circumstances, and in the worst case, the test cycle must be restarted from the beginning. The conditions for prematurely terminating a test cycle and for extending a test cycle must be precisely stated. Essentially, the concept of a revert criterion is used to ensure that system testing contributes to the overall quality of the product at reduced cost. It may not be useful to continue a test cycle if it is found that the software is of too poor quality. On the other hand, a test cycle can be extended due to various reasons, such as (i) a need to reexecute all the test cases in a particular test group because a large fraction of the test cases within the group failed or (ii) a significantly large number of new test cases were added while test execution was in progress. The idea of a revert criterion discussed above is similar to the idea of stopping rules for testing [9].

Actions Two unexpected events may occur while a test cycle is in progress: (i) too many test cases fail and (ii) the system test team has to design a large number of new test cases during the test cycle. On the one hand, the development team tries to understand what went wrong during development when too many test cases fail. On the other hand, when too many new test cases have to be designed, the system test team tries to understand what went wrong in the test planning phase. The two kinds of events require different kinds of responses from the development team and the system test team, as explained below.

It is useful to alert the development team once the number of failed test cases reaches a predefined level. Consequently, the developers take an action in the form of root cause analysis (RCA). In this analysis, the developers study the software components to identify whether a defect was in new, old, rewritten, or refixed code, which is discussed in Chapter 13. Next, the following question is asked: Would it have been possible to detect the defect during an earlier phase in the development cycle, such as design review, code review, unit testing, or integration testing? If the answer is yes, then corrective actions are taken by updating the design specification, reviewing the code, and adding new test cases for unit testing and integration testing to improve the quality of these error-prone software

components. The detailed defect analysis is discussed in the system test execution Chapter 13.

Test engineers may find the need to design new test cases as test executions reveal defects and those defects are fixed. The total number of test cases keep increasing as testing continues. A significant increase in the number of test cases implies that the initial planning for test design was not very accurate, and this may adversely affect the test execution schedule. In addition, management may want to know the reason for the initial number of test cases being low. The system test team initiates an RCA when the total number of newly designed test cases crosses a certain threshold. The test team studies all the new test cases and categorizes them into different groups based on the functional requirements. Next, the functional specification and/or any other relevant documents are studied to understand why the test team was unable to identify the objectives of these new test cases in the first place.

Exit Criteria We know when a test cycle has completed by applying an exit criterion. Mere execution of all the test cases in a test suite, or a subset thereof, does not mean that a cycle has completed. There is a need to monitor some quality metrics associated with the test cycle. Examples of these metrics are explained in detail for three test cycles. Determining the exit criteria of the final test cycle is a complex issue. It involves the nature of products (e.g., shrink-wrap application versus an operating system), business strategy related to the product, marketing opportunities and timing, and customer requirements, to name just a few [10]. The exit criteria considered here takes the view that quality is an important attribute and that time-to-market with desirable qualities is a major goal.

12.7.3 Preparing for First Test Cycle

It is desirable to understand the following concepts in order to get ready for the first test cycle: (i) the different states of a known defect in the form of a life cycle from the new state to the closed state, (ii) assignment of test cases to test engineers, and (iii) an entry criterion telling us when to start the first test cycle. These concepts are explained below.

Life Cycle of a Defect The idea behind giving a life-cycle model to defects is to be able to track them from the time they are detected to the time they are closed. The life cycle of a defect is represented by a state transition diagram with five states: new, assigned, open, resolved, and closed. The states of defects are used in the description of individual test cycles. A defect is put in the new state by a test engineer, called a submitter, as soon as it is revealed by a failed test case. After a defect in the new state is reviewed and presumed to be a true defect, it is moved to the assigned state, which means that the defect has been assigned to a software developer. The software developer moves the defect from the assigned state to the open state as soon as he or she starts working on the defect. Once the developer is satisfied by means of unit testing that the defect has been fixed, the defect is moved to the resolved state. The submitter verifies the fix by executing

the associated failed test cases against the new build prepared by the development team after a defect moves to the resolved state. If the submitter is satisfied by means of system testing that a defect in the resolved state has been truly fixed, the defect is moved to the closed state, and the associated test cases are declared as passed. Otherwise, the defect is moved back to the open state, and the associated test cases are still considered to have failed. A detailed model of a defect life-cycle is discussed in Chapter 13.

Assignment of Test Cases Since no single engineer is going to execute all the test cases all by themselves, it is desirable to assign test cases to appropriate test engineers by considering their expertise and interest. Assignment of test cases to test engineers may be changed from test cycle to test cycle for three reasons: (i) when a test case is assigned to a different engineer, the test case is likely to be executed from a different perspective; (ii) an opportunity to learn something new makes a positive impact on employee morale; and (iii) knowledge about test cases is distributed throughout the test group, so if a test engineer is temporarily unavailable for the task, it is easy to find a competent replacement.

Entry Criteria for First Test Cycle The entry criteria for the first test cycle, given in Table 12.3, tell us when we should start executing system tests. The criteria consist of five groups which depend on five cross-functional groups: marketing, hardware, software, technical publication, and system testing. Each group has one or more condition items to be satisfied. System testing should not start until all the groups of the entry criteria are satisfied. Otherwise, a revert criterion may be triggered sooner during system test execution.

The first group of the entry criteria concerns the completion of the project plan. The marketing team provides a business justification for the proposed product and its requirements. The engineering team defines a development approach, an execution environment, a schedule of the milestones, and the possible risks involved, identifying the dependencies of the product on other elements. The software project plan must be reviewed and approved by the software, hardware, marketing, and technical publication groups before system testing is initiated.

System testing may include new hardware, and, therefore, the second group of the entry criteria is about adequate testing of any new hardware system or subsystem used by the software system. The pass-and-fail results obtained from executing test cases cannot be considered to be dependable without a stable hardware base. It is important to ensure that the hardware has gone through the following three phases: (i) planning and specification; (ii) design, prototype implementation, and testing; and (iii) integration with the software system.

The third group of the entry criteria is about completion of the development and testing work by the software development group. It consists of seven criteria, which provides evidence that the system is sufficiently stable to withstand the rigor of system testing. The stability of a system should be based on metrics and evidence, rather than statements from software managers to start system testing. For example, documentation of weekly passed-and-failed status of all the unit and integration test cases should be available for review. The unit test and system

TABLE 12.3 Entry Criteria for First System Test Cycle

Group	Criteria
1. Marketing	1. Project plan and/or system requirements document is complete and updated.
2. Hardware	2. All approved hardware versions for field deployment are available in-house. A list of these hardware versions should be provided.
	3. Hardware version control process is in place and documented.
	4. Hardware DVT plan is completed and results are available.
3. Software	5. All functional specifications (FSs) are complete and have been updated to be in sync with the implementation. A list of individual FSs, including version number and status, must be provided.
	6. All design specifications (DSs) are complete and have been updated to be in sync with the implementation. A list of DSs, including version number and status, must be provided.
	7. All code complete and frozen; code changes allowed for only defect fixing but not features.
	8. A software version control is in place.
	9. 100% unit tests are executed and passed.
	10. 100% system integration tests are executed and passed.
	11. Not more than two unique crashes have been observed during the last two weeks of integration testing.
4. Technical publication	12. A draft version of the user guide is available.
5. System testing	13. The system test plan is in place (reviewed and approved).
	14. Test execution working document is in place and complete.

integration test reports must be thoroughly reviewed by the system test team before the start of the first test cycle.

The fourth group of the entry criteria concerns the readiness of the user guides written by technical writers. Unless the user guides are ready, the system test engineers will not be able to verify the accuracy and usability.

The fifth group of the entry criteria relates to a system test plan. Detailed test cases are included in the test plan. One can estimate the quality of the software early in the system testing phase [11] by documenting the test cases. The system test plan must be reviewed and approved by the software, hardware, marketing, and technical publication groups before the start of system testing.

Cross-functional review meetings must be conducted to track the status of the readiness criteria, outlined in Table 12.3, at least four weeks prior to the start of the first test cycle. Representatives from the five groups must attend the cross-functional review meeting to provide the status in their respective areas. Any exceptions to these criteria must be documented, discussed, and agreed upon at the final cross-functional status review meeting.

12.7.4 Selecting Test Cases for Final Test Cycle

Though it is desirable to reexecute all the test cases in the final test cycle, a lack of time and additional cost may not allow the system test team to do so. The concept of regression testing is applied only in the final test cycle. In our approach, test cases are selected based on (i) the results of their prior execution; (ii) their membership in certain test groups: basic, functionality, robustness, interoperability, stress, scalability, performance, and load and stability; and (iii) their association with software components that have been modified.

Test cases are selected in three steps: In the first step, the test suite is partitioned into four different bins—red, yellow, green, and white—based on certain criteria which are described in the selection procedure given below. The red bin is used to hold the test cases that must be executed. The yellow bin is used to hold the test cases that are useful to execute. The green bin is used to hold the test cases that will not add any value to regression testing and thus can be skipped. The rest of the test cases are included in the white bin. In other words, the test cases for which no concrete decision can be made in the first step are put in the white bin. In the second step, test cases from the white bin are moved to the other bins by considering the software components that were modified during system testing and the test cases that are associated with those components. Finally, in the third step, the red and yellow bins are selected for regression testing. In the following, we present the test selection procedure:

Step 1: The test suite is partitioned into red, yellow, green, and white bins as follows:

- **Red:** The red bin holds the following kinds of test cases that *must* be executed:

  Test cases that failed at least once in the previous test cycles.

  Test cases from those test groups for which RCA was conducted by the development team in the previous test cycles.

  Test cases from the stress, scalability, and load and stability test groups. These test cases are more likely to reveal system crash defects. They are selected to ensure that the final build is stable and the probability of the system crashing at the customer site is extremely low.

  Test cases from the performance test category. The performance characteristic must be measured against the final build that is going to be released to the customers.

- **Yellow:** The yellow bin holds the test cases that are *useful* to execute. This bin includes those test cases whose objectives are similar to the objectives of the test cases in the red bin. For example, let a test case with the following objective be in the red bin: While software up-gradation is in progress, the CPU utilization should not exceed 60%. Then, a test case with the following objective is put in the yellow bin: Verify that software image can be upgraded to the nth release from the $(n-1)$th release in less than 300 seconds. The condition, that is, less

than 60% CPU utilization, tells us when to execute the upgradation test.

- **Green:** The green bin holds the test cases that will not add any value to regression testing and thus can be skipped. This bin includes those test cases whose objectives are implicitly covered by the execution of the test cases in the red and yellow bins. For example, if a test case with the objective "Verify that a software image can be upgraded to the nth release from the $(n-1)$th release in less than 300 seconds" is in the yellow bin, then a basic test case with the objective "The software can be upgraded to the nth release from the $(n-1)$th release" is included in the green bin.
- **White:** The test cases for which no concrete decision can be made in the first step are put in the white bin. This includes the rest of the test cases not falling in the red, yellow, or green bin.

Step 2: Test cases from the white bin are moved to the other bins by considering the software components that were modified during the system testing and the test cases that are associated with those components. The software developers identify all the software components that have been modified after the start of the first test cycle. Each test case from the white bin is mapped to the identified software components. This mapping is done by analyzing the objective of the test case and then checking whether the modified code of the identified software components is exercised by executing the test cases. The test cases that are mapped to more than one, one, or zero software components are moved to the red, yellow, or green bin, respectively.

Step 3: Test cases from the red and yellow bins are selected for regression testing as follows:

- All the test cases from the red bin are selected for the final test cycle.
- Depending on the schedule, time to market, and customer demand, test cases from the yellow bin are selected for the final test cycle.

Remark 2. It is useful to understand the selection strategy explained above. The red bin holds the test cases that must be selected in the final test cycle. The test cases falling in the "must execute" category are (i) the test cases which had failed in the previous test cycles; (ii) the test cases from the groups for which RCA was performed in the previous test cycles; (iii) test cases from the stress, scalability, and load and stability test groups; and (iv) the test cases which concern modified software components. We remind the reader that RCA for a test group is performed if too many test cases had failed from that group. We put emphasis on a test group by selecting its test cases in the final test cycle. Test cases from the stress, scalability, and load and stability groups are also included in the final test cycle by default, even though those test cases may not concern the modified components. The rationale for their inclusion is that one must run a final check on the stress, scalability, load, and stability aspects of software systems—such as

servers, Internet routers, and base stations in wireless communication—which are likely to serve thousands of simultaneous end users.

12.7.5 Prioritization of Test Cases

Prioritization of test cases means ordering the execution of test cases according to certain test objectives. Formulating test objectives for prioritization of individual test cases is an extremely difficult task. Here, we discuss test prioritization in terms of groups of test cases with common properties. An example of a test objective for prioritization is: Execute the maximum number of test cases without being blocked. Furthermore, a major concern from software developers is that test engineers report critical defects (e.g., defects related to system crash) toward the end of test cycles. This does not give them enough time to fix those defects. In addition, blocking defects need to be fixed earlier in the test cycles in order to execute the blocked test cases. Therefore, we need to prioritize the execution of tests to detect critical defects early in the test cycles. In a multicycle-based test execution strategy, it is desirable to have different test objectives for prioritization in different test cycles for three reasons: (i) initially the quality level of the system under test is not very high, (ii) the quality of the system keeps improving from test cycle to test cycle, and (iii) a variety of defects are detected as testing progresses. Below we explain test prioritization in individual test cycles.

Test Prioritization in Test Cycle 1

Principle. Prioritize the test cases to allow the maximum number of test cases to completely execute without being blocked.

Test engineers execute their assigned test cases in different test environments. Each engineer prioritizes the execution of their subset of test cases as follows:

- A high priority is assigned to the test cases in the basic and functionality test groups.
- A medium priority is assigned to the robustness and interoperability test groups.
- A low priority is assigned to the test cases in the following groups: documentation, performance, stress, scalability, and load and stability tests.

The basic tests give *prima facie* evidence that the system is ready for more rigorous tests. The functionality tests provide a comprehensive testing over the full range of the requirements within the capabilities of the system. Both of these test groups are given high priority in the first test cycle to ensure that any functionality defects are fixed first. Functionality defects can block the execution of other test group. Stress, performance, scalability, load, and stability tests need complex configurations across different platforms, operating systems, and database management systems. Execution of these tests depends on the outcome of the interoperability and robustness tests. Therefore, interoperability and robustness test cases are executed

next to flush out any issues that may block the execution of stress, performance, scalability, load, and stability tests.

Test Prioritization in Test Cycle 2

Principle. Test cases which failed in the previous test cycle are executed early in the test cycle.

In the second test cycle, the test cases are reassigned to the test engineers based on their interest and expertise. The process described in Section 12.7.4 is used to distribute all the test cases into three different bins: red, yellow, and green. In this step, we are not selecting a subset of the test suite. Instead, the idea of partitioning a test suite is further used in prioritizing test cases. Each test engineer prioritizes the execution of test cases in their subset as follows:

- A high priority is assigned to the test cases in the red bin.
- A medium priority is assigned to the test cases in the yellow bin.
- A low priority is assigned to the test cases in the green bin.

Test Prioritization in Test Cycle 3

Principle. Test prioritization is similar to that in the second test cycle, but it is applied to a selected subset of the test cases chosen for regression testing.

Once again, the test cases are reassigned based on interest and expertise among the test engineers. Then each test engineer prioritizes the execution of test cases in their assigned subset as follows:

- A high priority is assigned to the test cases in the red bin.
- A low priority is assigned to the test cases in the yellow bin.

The reader may recall from the discussion of Section 12.7.4 that the test cases in the green bin are not executed in the final test cycle.

12.7.6 Details of Three Test Cycles

As we move from test cycle to test cycle, new test cases may be included, and the revert criteria and the exit criteria are made more stringent so that the quality of a system improves, rather than deteriorates, as testing progresses. Note that we have given values of the parameters in a test cycle. The concrete values used in the test cycles are customizable according to the testing capabilities and needs of an organization. The values used in the paper are actual values used in real test projects. We have used concrete values in the description of test cycles to make the descriptions meaningful and to be able to observe improvement in quality from test cycle to test cycle.

Test Cycle 1 In this cycle, we try to detect most of the defects by executing all the test cases. The six characteristic parameters of the first test cycle are described in the following.

Goals We intend to execute all the test cases from the test suite and maximize the number of passed test cases. The goal is to ensure that 98% of the test cases have passed.

Assumptions The system test group accepts a software image once every week for the first four weeks and once every two weeks afterward. The possibility of some test cases being blocked is more in the first four weeks of the test cycle because of more priority 1 ("critical") defects being reported earlier in the test cycle due to the execution of higher priority test cases. Unless these defects are resolved quickly, the test execution rate may slow down. Therefore, it is useful to accept new software images every week. We have observed that software images become more stable after four weeks, and test execution is not blocked. Subsequently, the system test team may accept software images once every two weeks.

Test Execution Test cases are executed according to the prioritization strategy for this test cycle explained in Section 12.7.5.

Revert and Extension Criteria Essentially, if the number of failed test cases reaches 20% of the total number of test cases to be executed, the system test team abandons this test cycle. The test cycle is restarted when the development team claims that the defects have been fixed. Assume that there are 1000 test cases to be executed. If the system test team observes that 200 test cases, which is 20% of 1000, out of the first, say, 700 test cases have failed, there is no point in continuing with the test cycle. This is because the quality of the product is too low, and any further testing before the defects are fixed is a waste of testing resources. If more than two unique crashes are observed during the test cycle, the system test team runs regression tests after the crash defects are fixed. If the number of failed test cases for any group of test cases, such as functionality, performance, and scalability, reaches 20% of the number of test cases in the group during the test cycle, the system test team reexecutes all the test cases in that group in this cycle after the defects are presumed to be fixed. Consequently, the duration of the test cycle is extended. Similarly, if the number of new test cases increases by 10% of the system test cases, the test cycle is extended to document the additional test cases.

Action The software development group initiates an RCA during the test cycle if the total number of failed test cases reaches some preset values as shown in Table 12.4. For example, if 25% of all test cases executed fail in a single week from a single group, then the development team performs an RCA. The system test group initiates an RCA if the number of new test cases increases by 10% of the total number of test cases designed before the test cycle was started.

TABLE 12.4 Test Case Failure Counts to Initiate RCA in Test Cycle 1

	Test Case Failure Count (%)	
	Single Week	Cumulative Weeks
Single test group	25	20
All test groups	20	15

Exit Criteria The test cycle is considered to have completed when the following predicates hold: (i) new test cases are designed and documented for those defects that were not detected by the existing test case, referred to as *test case escapes*; (ii) all test cases are executed at least once; (iii) 95% of test cases pass; and (iv) all the known defects are in the closed state.

Remark 3. Test case escapes occur because of deficiencies in the test design process. They are identified when test engineers find defects or when they encounter conditions that are not described in the plan. This happens by accident or when a new test scenario occurs to test engineers while executing the planned test cases. As test engineers learn more about the product, they develop innovative ways to test the product and find new defects. These test cases had escaped from the test case design effort, which is also known as side effects [12].

Remark 4. A software development group may take more time to fix defects that cause the failure of 5% of test cases. In that case there is no point in waiting for the fix and indefinitely delaying the completion of the first test cycle until an additional 3% of test cases pass in order to achieve the stated goal of 98% passed test cases set out for the first test cycle. It is indeed the case that some defects take much more time to be fixed, and some are deferred until the next software release. It is advisable to exit from the first test cycle to start the second cycle for effective and efficient utilization of resources. In any case, our strategy in the second test cycle is to execute all the test cases, which includes those failed 5% of test cases as well.

Test Cycle 2 The fixes for the defects found in the first test cycle are verified in the second test cycle. One of the four possibilities occurs for each modification while fixing a defect: (i) the defect is fixed without introducing a new defect; (ii) the defect is fixed, but something working is broken; (iii) neither the defect is fixed nor new defects are introduced; and (iv) the defect is not fixed, but something else is broken. Moreover, all the entry criteria may not be satisfied before the start of the first test cycle, which may be resolved by cross-functionality groups during the execution of the first test cycle. Sometimes, a demanding customer may cause a last-minute feature check-in before the start of the second test cycle; otherwise, the customer will not deploy the product. It is desirable to reexecute every test case during the second test cycle to ensure that those changes did not adversely

impact the quality of the software due to those uncertainties. The six characteristic parameters of the second test cycle are described below.

Goals All test cases are executed once again to ensure that there is no collateral damage due to the fixes applied in the first test cycle. The number of passed test cases are maximized, ensuring that 99% of the test cases have passed at the end of the second test cycle.

Assumption The system test group accepts software images every two weeks. This is because the system is relatively stable after having gone through the first test cycle.

Test Execution Test cases are executed according to the prioritization strategy for this test cycle explained in Section 12.7.5.

Revert and Extension Criteria At any instant in the test cycle, if the total number of failed test cases reaches 10% of the total number of test cases to be executed, the system test team stops testing. The test cycle is restarted from the beginning after the development team claims that the defects have been fixed. If more than one unique crash is observed during this test cycle, the system test team runs regression tests after the crash defects are fixed. This means that the duration of the test cycle is extended. If the number of failed test cases reaches 10% of the total number of tests in a group during the test cycle, the system test team reexecutes all the test cases from that group in this test cycle after the defects are claimed to be fixed. Therefore, the duration of the test cycle needs to be extended. The test cycle is extended to document the additional test cases if the number of new test cases increases by 5% of the total number of test cases.

Action The software development group initiates an RCA during the test cycle if the counts of failed test cases reach some preset values as shown in Table 12.5. For example, if 15% of all test cases executed in a single week from a single group fail, then the development team performs an RCA.

The system test group initiates an RCA if the number of new test cases is increased by 5% of the total number of test cases before the start of the second test cycle. A specific explanation is given if new test cases are added during the first test cycle and also new test cases are added during the second test cycle.

TABLE 12.5 Test Case Failure Counts to Initiate RCA in Test Cycle 2

	Test Case Failure Count (%)	
	Single Week	Cumulative Weeks
Single test group	15	10
All test groups	10	5

Exit Criteria The test cycle is considered to have completed when the following hold: (i) new test cases are designed and documented for those defects that were not detected by the existing test case, (ii) all test cases are executed at least once, (iii) 98% of test cases pass, and (iv) all the known defects are in the closed state.

Remark 5. One of the exit criteria is that 98% of test cases must pass, which is higher than the first test cycle. Once again, there is no point for test engineers to wait for the fixes and indefinitely postpone the completion of the second test cycle until another 1% of the failed test cases are passed in order to achieve the 99% pass goal set before the start of the second test cycle.

Test Cycle 3 In the third and final test cycle, a selected subset of test cases of the original test suite is reexecuted to ensure that the software is stable before it is released to the customer. The objective of this test cycle is to execute all the selected test cases against a single software image.

Goals A selected subset of test cases from the test suite are reexecuted. The process of selecting test cases for this test cycle has been explained in Section 12.7.4. At the end of the cycle, the software is released to the customer. Therefore, 100% of the selected test cases should pass.

Assumption The system test group accepts just one software image at the beginning of the test cycle. In exceptional circumstances, the system test group may accept a second image during this cycle, but it should not be less than three weeks before the end of the test cycle.

Test Execution Test cases are executed according to the prioritization strategy for this test cycle explained in Section 12.7.5.

Revert and Extension Criteria At any instant during the test cycle, if the total number of failed test cases exceeds 5% of the total number of test cases to be executed, the system test team stops testing. Testing is also stopped if a single crash is observed. The test cycle restarts from the beginning after the development team claims that the defects have been fixed, since this is the final test cycle before the release. All the selected test cases need to be reexecuted to ensure that there is no collateral damage due to the fix. Therefore this test cycle can be terminated and restarted again, but not extended.

Exit Criteria The final test cycle is considered to have completed if all of these predicates hold: (i) all the selected test cases are executed; (ii) the results of all the tests are available; (iii) 98% of test cases pass; (iv) the 2% failed test cases should not be from stress, performance, scalability, and load and stability test groups; (v) the system does not crash in the final three weeks of testing; and (vi) the test report is completed and approved. The test report summarizes the test

results of all the three test cycles, performance characteristics, scaling limitations (if any), stability observations, and interoperability of the system.

Remark 6. One of the exit criteria is that 98% of test cases must pass in the third test cycle. One can ask: Why is it not 100%? This was the goal set before the start of the third test cycle. It must be noted that the exit criteria of the final test cycle are influenced by a trade-off among time, cost and quality. As Yourdon [13] has argued, sometimes less than perfect is good enough. Only business goals and priority determine how much less than perfect is acceptable. Ultimately, the exit criteria must be related to the business goals of the organization. Exit criteria are generally not based solely on quality; rather, they take into consideration the innovation and timeliness of the product to be released to the market. Of course, for any mission-critical application, the pass rate of system test cases must be 100%.

Our experience told us that fewer than three test cycles is not good enough to give us much confidence in a large software system unless everything works to perfection as planned—which is a rare case. By "perfection" we mean that (i) the product does not change apart from the defect fixes, (iii) the fixes work as intended, and (iii) the test engineers do not add any new test cases. Rarely are these three conditions satisfied. More than three test cycles can be scheduled with appropriate exit criteria in terms of increasing percentage of test cases passing in successive test cycles. However, more than three test cycles incurs a much higher cost, while delaying the launch of a product "just in time" to the market. Management does not want to delay the launch of a product unless it is a mission-critical project, where the goal is to have a zero-defect product. It is better to plan for three-cycle-based system testing but skip the third test cycle if there is no need. On the other hand, it is not desirable to budget for two test cycles and fall behind when the software is of low quality.

12.8 TEST EFFORT ESTIMATION

The system test group needs to estimate testing effort to produce a schedule of test execution. Intuitively, testing effort defines the amount of work that needs to be done. In concrete terms, this work has two major components:

- The number of test cases created by one person in one day
- The number of test case executed by one person in one day

The above components are also referred to as test productivity. Unfortunately, there is no mathematical formula to compute the test productivity numbers. Rather, the test effort data are gathered by measuring the test creation and execution time in real projects by taking a microscopic view of real testing processes. If productivity data are collected for many test projects, an average test productivity measure can be estimated. Our experience on this issue is discussed in this section.

In the planning stage it is natural to estimate the cost of the test and the time to complete the test. Together the two parameters *cost* and *time* are called *test effort*.

In most cases, estimation of the system test effort is combined with estimation of the entire software project. However, it is useful to separate the test effort estimate from the estimate of the entire project so that enough time is allocated to plan for the system testing from the beginning and conduct it as soon as the entry criteria are satisfied. The three key factors in estimating test effort are as follows:

1. Number of test cases to be designed for the software project
2. Effort required to create a detailed test case
3. Effort required to execute a test case and analyze the result of execution

Some other factors affecting the test effort are as follows:

- Effort needed to create test environments
- Effort needed to train test engineers on the details of the project
- Availability of test engineers for system testing as they are needed

We now discuss the ways to estimate the key factors affecting test effort. Test planners rely on their past experience in testing similar projects and a thorough knowledge of the current project to provide accurate estimation of the above three key items.

12.8.1 Number of Test Cases

It is useful to understand the commonly used term "test case" before it is used for estimation. The *granularity* of test cases has a significant impact on the estimation of the number of test cases needed. For example, one person may use the term test case to mean one atomic test step, which is just a single input–output interaction between the tester and the SUT. Another person may use the same term test case to mean hundreds of test steps. Our definition of test case is independent of the number of test steps needed to construct a test case. The granularity of a test case is tightly coupled with the test objective, or purpose. A simple test case typically contains 7–10 test steps, while a complex test case can easily contain 10–50 test steps. The test steps are atomic building blocks which are combined to form a test case satisfying a well-defined objective. We discuss several ways to estimate the number of test cases.

Estimation Based on Test Group Category It is straightforward to estimate the number of test cases after the test suite structure and the test objectives are created, as discussed in Section 12.5, by simply counting the number of test objectives. However, this will give an underestimation of the total number of test cases designed by the end of the system test execution cycle. This is because as system testing progresses with the initially estimated and designed test cases, test engineers want to observe additional behavior of the system due to their observations of some unforeseen failures of the system. The number of test cases used by the end of the system testing phase is more than its initial estimation. A more accurate estimation of the number of test cases for each group is obtained by adding a "fudge factor" of 10–15% to the total number of test objectives identified for the group. On the

one hand, underestimation of the number of test cases causes reduced allocation of resources to system testing at its beginning. However, it demands more resources when the need is felt, causing uncontrollable project delay. On the other hand, overestimation of the number of test cases leads to their inefficient utilization. For pragmatic reasons, for any moderate-size, moderate-risk testing project, a reasonable factor of safety, in the form of a fudge factor, needs to be included in the estimate in order to provide a contingency for uncertainties.

The test team leader can generate a table consisting of test (sub)groups and for each (sub)group the estimated number of test cases. An additional column may be included for time required to create and time required to execute the test cases, which are discussed later in Sections 12.8.2 and 12.8.3, respectively.

Example: FR–ATM PVC Service Interworking. Let us consider the FR–ATM PVC service interworking example discussed in Chapter 11. A test effort estimation is given in Table 12.6 based on the test categories and FrAtm structure discussed in Chapter 11. The first column is estimated based on the number of test objectives identified for the system. The procedure to obtain the estimations is given below. The estimation of the next two columns of Table 12.6 are discussed in Sections 12.8.2 and 12.8.3, respectively.

One test case is required to test the attribute for each configurable attribute of the FrAtm component. Therefore, we estimate that 40 test cases will be required to test the configuration functionality with our assumption of 40 configuration attributes. Similarly, 30 test cases need to be created in order to test the monitoring feature with our assumption that 30 monitoring attributes are available in the implementation of an FrAtm component. The configuration and monitoring attributes information should be available in the functional specification of the

TABLE 12.6 Test Effort Estimation for FR–ATM PVC Service Interworking

Test (Sub)Groups	Estimated Number of Test Cases	Person-Day to Create	Person-Day to Execute
Configuration	40	4	6
Monitoring	30	4	4
Traffic Management	3	2	3
Congestion	4	2	4
SIWF Translation Mode Mapping	12	4	4
Alarm	4	1	1
Interface	9	5	3
Robustness	44(4 + 40)	4	6
Performance	18	6	10
Stress	2	1.5	2
Load and stability	2	1.5	2
Regression	150(60 + 90)	0	15
Total	**318**	**35**	**60**

FrAtm component. The traffic management test cases are estimated based on the number of QoS parameters to be mapped from FR to ATM, which is approximately 3. The number of interface tests is estimated based on the number of RF and ATM cards supported by the FrAtm component, which is 9. One test case needs to be created for each interface card. Therefore, 9 test cases are estimated for the interface test group. Forty-four test cases are estimated for robustness group, 4 for the FrAtm subgroup and 40 for boundary value tests. One boundary value test case needs to be designed for each configuration attribute. Therefore, we estimated 40 test cases for boundary value functionality testing. Performance test cases are estimated based on the following assumptions:

- Delay measurement for FrAtm with 3 types of ATM cards. This requires us to design 3 test cases.
- Delay measurement for FrAtm with 6 types of FR cards. Therefore, we need to design 6 test cases.
- Throughput measurement using one FR card one at a time. There are 6 types of FR cards, and thus we need 6 test cases.
- Throughput measurement using one ATM card at a time. There are 3 types of ATM cards, and thus we need 3 test cases.

The estimated number of test cases for performance testing may change if the combinations of cards are considered with different objectives. Finally, a number of test cases for regression testing are selected from the existing FR and ATM test suites. For simplicity, we assume that 60 and 90 test cases are selected from the existing FR and ATM test suites, respectively.

Estimation Based on Function Points The concept of *function points* is used to estimate resources by analyzing a requirements document. This methodology was first proposed by A. J. Albrecht in 1979 [14, 15]. The function point method is becoming more popular in the industry, though it is felt that it is still in an experimental state. The central idea in the function point method is as follows: *Given a functional view of a system in the form of the number of user inputs, the number of user outputs, the number of user on-line queries, the number of logical files, and the number of external interfaces, one can estimate the project size in number of lines of code required to implement the system and the number of test cases required to test the system*.

Albrecht [14] has shown that it is useful to measure a software system in terms of the number of "functions" it performs and gave a method to "count" the number of those functions. The count of those functions is referred to as function points. The function point of a system is a weighted sum of the numbers of inputs, outputs, master files, and inquiries produced to or generated by the software. The four steps of computing the function point of a system are described below:

Step 1: Identify the following five types of components, also known as "user function types," in a software system and count them by analyzing the requirements document:

- Number of external input types (NI): This is the distinct number of user data or user control input entering the boundary of a system.
- Number of external output types (NO): This is the distinct number of user data or control output type that leaves the external boundary of the system.
- Number of external inquiry types (NQ): This is the distinct number of unique input–output combinations, where an input causes and generates an immediate output. Each distinct input–output pair is considered as an inquiry type.
- Number of logical internal file types (NF): This is the distinct number of major logical groups of user data or control information in the system. Each logical group is treated as a logical file. These groups of data are generated, used, and maintained by the system.
- Number of external interface file types (NE): This is the distinct number of files passed or shared between systems. Each major group of data or control that leaves the system boundary is counted as an external interface file type.

Step 2: Analyze the complexity of each of the above five types of user functions and classify those to three levels of complexity, namely simple, average, or complex. For example, the external input types are simple, the logical internal file types have average complexity, and the number of external interface file types are complex in nature.

A weighting factor is associated with each level of complexity for each type of user function. Two types of user functions with the same level of complexity need not have the same weighting factor. For example, simple external inputs may have a weighting factor of 3, whereas simple external inquiry types may have a weighting factor of 4. Let WFNI denote the weighting factor of external input types, WFNO denote the weighting factor of external output types, and so on. Now, the unadjusted, or crude, function point, denoted by UFP, is computed as

$$\begin{aligned} \text{UFP} = {} & \text{WFNI} \times \text{NI} + \text{WFNO} \times \text{NO} + \text{WFNQ} \times \text{NQ} \\ & + \text{WFNL} \times \text{NL} + \text{WFNE} \times \text{NE} \end{aligned}$$

Compute the unadjusted function point by using the form shown in Table 12.7.

Step 3: Albrecht [14] has identified 14 factors that affect the required development effort for a project. A grade—between 0 and 5—is assigned to each of the 14 factors for a certain project, where 0 = not present or no influence if present, 1 = insignificant influence, 2 = moderate influence, 3 = average influence, 4 = significant influence, and 5 = strong influence. The sum of these 14 grades is known as the processing complexity adjustment (PCA) factor. The 14 factors are listed in Table 12.8.

TABLE 12.7 Form for Computing Unadjusted Function Point

		Complexity (Use One Item in Each Row)			
No.	Identifier	Simple	Average	Complex	Total
1	NI	— × 3 = —	— × 4 = —	— × 6 = —	
2	NO	— × 4 = —	— × 5 = —	— × 7 = —	
3	NQ	— × 7 = —	— × 10 = —	— × 15 = —	
4	NF	— × 5 = —	— × 7 = —	— × 10 = —	
5	NE	— × 3 = —	— × 4 = —	— × 6 = —	
				Total(UFP)	

TABLE 12.8 Factors Affecting Development Effort

1. Requirement for reliable backup and recovery
2. Requirement for data communication
3. Extent of distributed processing
4. Performance requirements
5. Expected operational environment
6. Extent of on-line data entries
7. Extent of multiscreen or multioperation data input
8. Extent of on-line updating of master files
9. Extent of complex inputs, outputs, on-line queries, and files
10. Extent of complex data processing
11. Extent that currently developed code can be designed for reuse
12. Extent of conversion and installation included in the design
13. Extent of multiple installations in an organization and variety of customer organizations
14. Extent of change and focus on ease of use

Step 4: Now, we compute the function point (FP) of a system using the following empirical expression:

$$FP = UFP \times (0.65 + 0.01 \times PCA)$$

By multiplying the unadjusted function point, denoted by UFP, by the expression $0.65 + 0.01 \times PCA$, we get an opportunity to adjust the function point by $\pm 35\%$. This adjustment is explained as follows. Let us analyze the two extreme values of PCA, namely 0 and 70 ($= 14 \times 5$):

$$PCA = 0: \qquad FP = 0.65 \times UFP$$

$$PCA = 70: \qquad FP = UFP \times (0.65 + 0.01 \times 70) = 1.35 \times UFP$$

Therefore, the value of FP can range from $0.65 \times UFP$ to $1.35 \times UFP$. Hence, we adjust the value of FP within a range of $\pm 35\%$ by using intermediate values of PCA.

The function point metric can be utilized to estimate the number of test cases in the following ways:

- **Indirect Method:** Estimate the code size from the function points and then estimate the number of test cases from the code size.
- **Direct Method:** Estimate the number of test cases directly from function points.

Indirect Method The function points of a software system are computed by examining the details of the requirements of the system from the requirement database. At the time of such a computation, the programming language in which the system will be implemented may not be known. Implementation of the system in different programming languages will produce different measures of line of code (LOC). Therefore, the choice of a programming language has a direct impact on the LOC metric. Capers Jones [16] gave a relationship between function points and the code size of a software system. Specifically, he gave the number of LOC per function point for a number of programming languages as shown in Table 12.9.

Given the total number of function points for a system, one can predict the number of LOC for a software system by making an assumption of a programming language in implementing the system. At this point, there is a need to utilize one's experience in estimating the number of test cases to be designed for system testing. This estimation practice varies among organizations and even within the same organization for different categories of software systems.

The 30 year empirical study of Hitachi Software [11] gives a standard of one test case per 10–15 LOC. A system with 100 function points and implemented in C will produce a code size of $100 \times 128 = 12{,}800$ lines. Test cases numbering between 850 and 1280 are designed for system-level testing of the software system under consideration. It must be remembered that these test cases do not include those test cases needed for unit testing and integration testing. This is because the test engineers in the system test group must be unbiased and never reuse the test cases of the programmers. Instead, the system testing team designs the test cases from the scratch.

TABLE 12.9 Empirical Relationship between Function Points and LOC

Programming Language	Average LOC/FP
Assembly language	320
C	128
COBOL	106
FORTRAN	106
C++	64
Visual Basic	32

Direct Method Capers Jones [16] gave a direct relationship between function points and the total number of test cases created as

$$\text{Total number of test cases} = (\text{function points})^{1.2}$$

The number of test cases estimated above encompasses all forms of testing done on a software system, such as unit testing, integration testing, and system testing. It does not distinguish system test size from total testing effort.

Now it is useful to compare the test estimation method used by Hitachi Software [11] and the Caper Jones method [16]. The method used by Hitachi Software estimates the number of test cases by considering code size in number of LOC, whereas Caper Jones estimates the number of test cases by considering function points. We start with a certain number of function points and estimate the code size therefrom for a meaningful comparison. For example, we have considered 100 function points in the two preceding examples. Next, we derive the LOC information by assuming a certain programming language to be able to apply to the Hitachi Software method. Assuming that our target language is C, the estimated number of LOC to be produced in C is $100 \times 128 = 12{,}800$ LOC. The method adopted by Hitachi Software produces 850–1280 test cases to test a system with 12,800 LOC, whereas the Caper Jones method estimates 251 test cases. Note that there is a four- to five-fold difference between the number of test cases estimated by the two methods. The difference is interpreted as follows: The Japanese software industry (e.g., Hitachi Software) uses the term *test point* rather than *test case*, and a test point is similar to a *test step*. A test case normally contains between 1 and 10 test steps. The number of test cases estimated by using the Hitachi Software method represents the number of test steps (or test points). Now, if we assume that a test case contains an average of five test steps, the number of test cases estimated by the Hitachi Software method is between 170 and 256, and this range is closer to 251—the number estimated by the Caper Jones method.

12.8.2 Test Case Creation Effort

It is necessary to allocate time to create test cases after the test suite structure and the test objectives are identified. On average, the productivity of the manual test case creation is summarized in Table 12.10. The time represents the duration from the entry to the create state to the entry to the released state of the state transition diagram of a test case discussed in Chapter 11.

TABLE 12.10 Guidelines for Manual Test Case Creation Effort

Size of Test Case	Average Number of Test Cases per Person-Day
Small, simple test case (1–10 atomic test steps)	7–15
Large, complex test case (10–50 atomic test steps)	1.5–3

The activities involved in creating test cases have been discussed in detail in Chapter 11 and are summarized here:

- Reading and understanding the system requirements and functional specifications documents
- Creating the test cases
- Entering all the mandatory fields, including test steps, and pass–fail criteria
- Reviewing and updating the test case

The skill and effectiveness of a test engineer are significant factors that influence the estimation of test case creation. We assume in our guideline provided in Table 12.10 that a test engineer is skilled in developing test cases, that is, he or she has four to six years of prior experience in developing test cases in the relevant area of expertise. Caper Jones [17] estimated that test cases are created at a rate of 30–300 per person-month. In other words 1.5–15 test cases can be created by one person in one day. Our estimation of creating test cases for each group of FR–ATM service interworking is given in the third column of Table 12.6 based on the guidelines provided in Table 12.10.

12.8.3 Test Case Execution Effort

The time required to execute a test case depends on the complexity of the test case and the expertise of the executer on the subject matter of the system. Arguably, the test cases in the telecommunications area are the most complex ones and require understanding configuration of switches, routers, and different protocols. In any case, one must consider the following factors in order to estimate the work effort for manual test execution:

- Understanding the test cases if the test cases have not been created by the executer
- Configuring the test beds
- Executing the test steps and evaluating the test steps
- Determining the execution status (pass–fail) of the test case
- Logging of test results in the test factory database
- Collecting and analyzing the data if performance and stress–load test cases are being executed
- Updating the test case in the test factory, as discussed in Chapter 11
- In the event of a failure, performing the following tasks:

  Trying to reproduce the failure

  Executing different scenarios related to the failure observed

  Collecting details of the configuration, logs, and hardware configuration

  Filing a defect report in the defect tracking system

- If the software developers cannot reproduce a failure, following up with them in debugging and localizing the problem

Intuitively—and it can also be observed—the test execution rate is linearly proportional to the pass rate of test cases. In other words, if the test case failure rate is high, it slows down the test case execution rate. This is because a test engineer needs to help the developers in replicating the problem and in identifying the root cause of the problem by trial and error, which takes away a lot of time from execution of test cases.

We have summarized, based on our experience, the execution rate of test cases for both new and regression tests in Tables 12.11 and 12.12, respectively. The execution time for newly created test cases is different than the time for the regression test. The difference is due to the fact that the regression test cases were run before, and their test steps and pass–fail criteria were validated earlier. However, if a test engineer has not executed a particular group of regression tests before, he may have to spend more time in understanding the tests and in setting up the test bed in order to execute the assigned regression tests. As a consequence, the execution time for the regression test is almost the same as that of the newly created test case for an inexperienced engineer. Therefore, test automation must be considered for regression tests, which is discussed in a later section of this chapter.

Our estimation of test case execution for each group of FR–ATM service interworking is given in the fourth column of Table 12.6 based on the guidelines provided in Tables 12.11 and 12.12. It may be noted that if automated regression tests are available, the total execution time could have been reduced.

TABLE 12.11 Guidelines for Manual Test Case Execution Effort

Size of Test Case	Average Number of Test Cases per Person-Day
Small, simple test case (1–10 atomic test steps)	7–15
Large, complex test case (10–50 atomic test steps)	1.5–2.5

TABLE 12.12 Guidelines for Estimation of Effort to Manually Execute Regression Test Cases

	Average Number of Regression Test Cases per Person-Day	
Size of Test Case	Did Not Execute Test Cases Earlier	Test Cases Executed Earlier
Small, simple test case (1–10 atomic test steps)	10–15	10–20
Large, complex test case (10–50 atomic test steps)	1–2.5	2.5–5

12.9 SCHEDULING AND TEST MILESTONES

For reasons of economy and to meet contractual deadlines, it is important to outline a schedule and the milestones for the test project. Organizations are constrained to release products within the time frame recommended by their marketing and sales departments because of today's high competitiveness in the market. The system test group is constrained to distribute its testing effort to accommodate the marketing requirement to release the product within a prescribed time frame. One must consider different avenues to complete the system testing phase on time without much delay in product delivery and without compromising the quality of a product. In scheduling the activities in system testing, the leader of the system test group attempts to coordinate the available resources to achieve the projected productivity. The leader considers any interdependency among tasks and schedules the tasks in parallel whenever possible. The milestones, reviews, and test deliverables are specified in the test schedule to accurately reflect the progress of the test project.

Scheduling system testing is a portion of the overall scheduling of a software development project. It is merged with the overall software project plan after a schedule for system testing is produced. It is essential to understand and consider the following steps in order to schedule a test project effectively:

- Develop a detailed list of tasks, such as:
  - Procurement of equipment to set up the test environments
  - Setting up the test environments
  - Creation of detail test cases for each group and subgroup in the test factory
  - Creation of a test suite in the test factory
  - Execution of test cases during the system test cycles
  - Writing the final test report after completion of system testing
- List all the major milestones needed to be achieved during the test project, such as:
  - Review of the test plan with cross-functional team members
  - Completion of the approved version of the test plan
  - Review of the newly created test cases and the chosen test cases for regression testing
  - Completion of test cases; all test cases are in their released states so that those can be a part of a test suite, as discussed in Chapter 11
  - Creation of the test suite to be executed, as discussed in Chapter 11
  - Preparation of test environments
  - Date of entry criteria readiness review meeting
  - Official release of the software image to system test team
  - Start of first system test cycle

End of first system test cycle

Start of second system test cycle

End of second system test cycle

Start of final system test cycle

End of final system test cycle

Final test report delivery date

- Identify the interdependencies among the test tasks and any software milestones that may influence the flow of work, such as official release of the software image to the system test group. In addition, identify the tasks that can be performed concurrently. Some examples of concurrent implementations of test tasks are as follows:

  Test beds can be set up concurrently with the creation of test cases.

  Creation of test cases in different groups can be done concurrently.

  Test cases from different groups can be concurrently executed on different test beds.

- Identify the different kinds of resources and expertise needed to fulfill each task on the list.
- Estimate the quantity of resources needed for each task, such as equipment and human resources, as discussed earlier in this chapter.
- Identify the types and quantities of resources available for this testing project, such as:

  Human resources: persons available and their dates of availability, part- or full-time availability of each individual human resource, and area of expertise of each individual human resource

  Hardware/ software resources: availability of all the hardware/software to build the test environment as discussed in Section 12.6

- Assume that the rest of the resources will be available by certain dates.
- Allocate the resources to each task. This allocation is done task by task in sequence, starting backward from the end date of the test project.
- Schedule the start and end dates of each task. For example, the test execution task can be based on the estimates discussed in Section 12.8.3. Allow delays for unforeseen events, such as illness, vacation, training, and meetings.
- At this point insert the rest of the milestones into the overall schedule. You may have to adjust the resource allocation and the schedule dates.
- Determine the earliest and the latest possible start date and end date for each task by taking into account any interdependency between tasks identified earlier.
- Review the schedule for reasonableness of the assumptions. At this stage, it is a good idea to ask "what if" questions. Analyze and change the schedule through iterations as the assumptions change.

- Identify the conditions required to be satisfied for the schedule. In addition, identify the contingency plan to be prepared for a possibility that those conditions are not met. For example, if it is not possible to hire a full-time test engineer by a certain date, one should go for a contractor or move in other test engineers from a different project.
- Document the assumptions, such as (i) new test engineer must be hired by a certain date, (ii) new equipment must be available by a certain date, and (iii) space availability to set up the test environment.
- Review the schedule with the test team and get their feedback. The schedule may not be met without their active participation.

The test effort estimate and schedule need to go through a few rounds of iterations in order to make them reliable. A good software tool available in the market is valuable to juggle the test schedule. A major problem the test team leader may encounter from management is how to shorten the test case execution time. Some tasks might be completely eliminated by reducing the scope and depth of testing. If not possible, one may include skilled test professionals with high proficiency, ask people to work overtime, and add more people and test beds. Another way to reduce the execution time is to automate all the test cases upfront so that execution time is reduced drastically.

Gantt Chart A Gantt chart is often used to represent a project schedule that includes the duration of individual tasks, their dependencies, and their ordering. A typical Gantt chart graphically displays the start and end points of each task—the total duration needed to complete each task. As the project progresses, it displays the percentage of completion of each task. It allows the planner to assess the duration of a project, identify the resources needed, and lay out the order in which tasks need to be carried out. It is useful in managing the dependencies between tasks. It is widely used for project planning and scheduling in order to:

- Assess the time characteristics of a project
- Show the task order
- Show the link between scheduled tasks
- Define resources involved
- Monitor project completion

In a Gantt chart, each task takes one row. Dates run along the top in increments of days, weeks, or months, depending on the length of the project. The expected time for each task is represented by a horizontal rectangle or line. Tasks may overlap or run in sequence or in parallel. Often the project has important events, and one would want to show those events on the project timeline. For example, we may wish to highlight when a prototype is completed or the date of a test plan review. One can enter these on a Gantt chart as milestone events and mark them with a special symbol, often a diamond. The tasks should be kept to a manageable number (no more than 15 or 20) during the construction of a Gantt chart so that the chart fits on a single page. More complex projects

may require subordinate charts which detail the timing of all the subtasks of the main tasks. It often helps to have an additional column containing numbers or initials which identify the team member responsible for the task for team projects.

Remark. The idea of a Gantt chart was originally proposed in 1917 by the American engineer Henry L. Gantt. He developed the first Gantt chart to be used in production flow planning. Accepted as a commonplace project management tool today, it was an innovation of worldwide importance in the 1920s. Gantt charts have been successfully used in large construction projects such as the Hoover Dam project started in 1931 and the interstate highway network started in 1956.

Example: FR–ATM PVC Service Interworking. Let us consider the FR–ATM PVC service interworking example discussed in Chapter 11. A high-level test schedule for this test project is shown in Figure 12.2 using the Gantt chart. We made the following assumptions in planning the test schedule:

- Four test engineers: Alex, Rohan, Inu, and Lucy are available for this project from day 1, and Alex is the test team leader for this project.
- All four engineers are well trained to generate test cases in the test factory.
- All of them are knowledgeable in the area of FR and ATM protocol.
- It took five days for Alex to develop the test plan for this project.

ID	Task Name	Duration	Start	Finish
1	Development of the test plan	5 days	Jan 3	Jan 7
2	Review the test plan	0 days	Jan 7	Jan 7
3	Update the test plan	1 day	Jan 10	Jan 10
4	Test plan approved	0 days	Jan 10	Jan 10
5	Procurement of equipment	7 days	Jan 11	Jan 19
6	Set up the test environment	10 days	Jan 20	Feb 2
7	Creation of test cases in test factory	18 days	Jan 11	Feb 3
8	Review the test cases	0 days	Feb 3	Feb 3
9	Update the test cases in test factory	4 days	Feb 4	Feb 9
10	Test cases are in released state	0 days	Feb 9	Feb 9
11	Creation of test suite in test factory	1 day	Feb 10	Feb 10
12	Entrance criteria readiness meeting	1 day	Feb 11	Feb 11
13	Official release of S/W to system test	0 days	Feb 14	Feb 14
14	Start of first test cycle	0 days	Feb 14	Feb 14
15	First test cycle	15 days	Feb 14	Mar 4
16	End of first test cycle	0 days	Mar 4	Mar 4
17	Start of second test cycle	0 days	Mar 4	Mar 4
18	Second test cycle	15 days	Mar 7	Mar 25
19	End of second test cycle	0 days	Mar 25	Mar 25
20	Beta release criteria review meeting	0 days	Mar 25	Mar 25
21	Release the software to beta customer	1 days	Mar 28	Mar 28
22	Start of third test cycle	0 days	Mar 28	Mar 28
23	Third test cycle	5 days	Mar 29	Apr 4
24	End of third test cycle	0 days	Apr 4	Apr 4
25	Test report preparation	4 days	Apr 5	Apr 8
26	Release criteria review meeting	0 days	Apr 8	Apr 8
27	Release the software to the customer	1 day	Apr 13	Apr 13

Figure 12.2 Gantt chart for FR–ATM service interworking test project.

- No one has any plans to take a vacation during the system testing period.
- Test equipment is available for this test project.

The test environment setup and the creation of test cases are done concurrently. Lucy is assigned to set up the test beds whereas the rest of the engineers are assigned to design test cases. We have estimated approximately 35 person-days to create 168 test cases, excluding the regression test case as given in Table 12.6. Therefore, we have allocated 18 days to design the test cases distributed among three test engineers, Alex, Inu, and Rohan, with a combined 54 person-days. This includes training and understanding of the FR–ATM service internetworking protocol. The first test cycle is started after the entry criteria are verified at the readiness review meeting. According to our estimation, it will take 60 person-days to execute all the selected 318 test cases, including the regression tests. Therefore, we have allocated 15 days to execute all the test cases by four test engineers, Alex, Inu, Rohan, and Lucy, with a combined resource of 60 person-days. After the second test cycle, the software is scheduled to be released to beta customers. A subset of the test cases is selected for regression testing in the final test cycle. Therefore, only one week is allocated for this task. The software is released to the customers after the final test report is created and reviewed.

12.10 SYSTEM TEST AUTOMATION

It is absolutely necessary for any testing organization to move forward to become more efficient, in particular in the direction of test automation. The reasons for automating test cases are given in Table 12.13. It is important to think about automation as a strategic business activity. A strategic activity requires senior management support; otherwise it will most likely fail due to lack of funding. It should be aligned with the business mission and goals and a desire to speed up delivery of the system to the market without compromising quality. However, automation is a long-term investment; it is an on-going process. It cannot be achieved overnight; expectation need to be managed to ensure that it is realistically achievable within a certain time period.

TABLE 12.13 Benefits of Automated Testing

1. Test engineer productivity
2. Coverage of regression testing
3. Reusability of test cases
4. Consistency in testing
5. Test interval reduction
6. Reduced software maintenance cost
7. Increased test effectiveness

The organization must assess and address a number of considerations before test automation can proceed. The following prerequisites need to be considered for an assessment of whether or not the organization is ready for test automation:

- The system is stable and its functionalities are well defined.
- The test cases to be automated are unambiguous.
- The test tools and infrastructure are in place.
- The test automation professionals have prior successful experience with automation.
- Adequate budget has been allocated for the procurement of software tools.

The system must be stable enough for automation to be meaningful. If the system is constantly changing or frequently crashing, the maintenance cost of the automated test suite will be rather high to keep the test cases up to date with the system. Test automation will not succeed unless detailed test procedures are in place. It is very difficult to automate a test case which is not well defined to be manually executed. If the tests are executed in an ad hoc manner without developing the test objectives, detailed test procedure, and pass–fail criteria, then they are not ready for automation. If the test cases are designed as discussed in Chapter 11, then automation is likely to be more successful.

The test engineers should have significant programming experience. It is not possible to automate tests without using programming languages, such as Tcl (Tool command language), C, Perl, Python, Java, and Expect. It takes months to learn a programming language. The development of an automation process will fail if the testers do not have the necessary programming skills or are reluctant to develop it. Adding temporary contractors to the test team in order to automate test cases may not work. The contractors may assist in developing test libraries but will not be able to maintain an automated test suite on an on-going basis.

Adequate budget should be available to purchase and maintain new software and hardware tools to be used in test automation. The organization should keep aside funds to train the staff with new software and hardware tools. Skilled professionals with good automation background may need to be added to the test team in order to carry out the test automation project. Therefore, additional head count should be budgeted by the senior executive of the organization.

12.11 EVALUATION AND SELECTION OF TEST AUTOMATION TOOLS

A test automation tool is a software application that assists in the automation of test cases that would otherwise be run manually. Some tools are commercially available in the market, but for testing complex, imbedded, real-time systems, very few commercial test tools are available in the market. Therefore, most organizations build their own test automation frameworks using programming languages such as C and Tcl. It is essential to combine both hardware and software for real-time testing

tools. This is due to the fact that special kinds of interface cards are required to be connected to the SUT. The computing power of personal computers with network interface cards may not be good enough to send traffic to the SUT.

Test professionals generally build their own test tools in high-technology fields, such as telecommunication equipment and application based on IP. Commercial third-party test tools are usually not available during the system testing phase. For example, there were no commercially available test tools during the testing of the 1xEv-DO system described in Chapter 8. The second author of this book developed in-house software tools to simulate access terminals using their own products. However, we advocate that testers should build their own test automation tools only if they have no alternative. Building and maintaining one's own test automation tool from scratch are time-consuming tasks and an expensive undertaking. The test tool evaluation criteria are formulated for the selection of the right kind of software tool. There may be no tool that fulfills all the criteria. Therefore, we should be a bit flexible during the evaluation of off-the-self automation tools available in the market. The broad criteria for evaluating test automation tools have been classified into the following eight categories as shown in Figure 12.3.

1. *Test Development Criteria:* An automation test tool should provide a high-level, preferably nonproprietary, easy-to-use test scripting language such as Tcl. It should have the ability to interface and drive modules that can be easily written in, for example, C, Tcl, Perl, or Visual Basic. The tool must provide facility to directly access, read, modify, and control the internals of the automated test scripts. The input test data should be stored separately from the test script but easily cross-referenced to the corresponding test scripts, if necessary. The tool should have built-in templates of test scripts, test cases, tutorials, and demo application examples to show how to develop automated test cases. Finally, no changes should be made to the SUT in order to use the tool. The vendor's

Figure 12.3 Broad criteria of test automation tool evaluation.

recommended environment should match the real test laboratory execution environment.

2. *Test Maintenance Criteria:* The tool should possess a rich set of features, such as version control capability on test cases, test data, and migration of test cases across different platforms. The tool must provide powerful, easy-to-use facilities to browse, navigate, modify, and reuse the test suites. The tool should have the ability to select a subset of test cases to form a group for a particular test run based on one or more distinguishing characteristics. A tool needs to have features to allow modification and replication of test cases, easy addition of new test cases, and import from another. The tool should have the capability to add multiple tags to a test case and modify those tags so that the test case can be easily selected in a subgroup of test cases sharing a common characteristic.

3. *Test Execution Criteria:* An automation tool should allow test cases to be executed individually, as a group, or in a predefined sequence. The user should have the ability to check the interim results during the execution of a group of tests and exercise other options for the remainder of the tests based on the interim results. The user should have the option to pause and resume the execution of a test suite. The tool should have the facility to execute the test suite over the Internet. The tool should allow simultaneous execution of several test suites that can be distributed across multiple machines for parallel execution. This substantially reduces the time needed for testing if multiple test machines are available. The test tool should have a capability for monitoring, measuring, and diagnosing performance characteristics of the SUT. Finally, the tool should have the capability to be integrated with other software tools which are either in use or expected to be used.

4. *Test Results Criteria:* The test tool must provide a flexible, comprehensive logging process during execution of the test suite, which may include detailed records of each test case, test results, time and date, and pertinent diagnostic data. A tool should have the capability to cross-reference the test results back to the right versions of test cases. The test result log can be archived in an industry standard data format and the tool should have an effective way to access and browse the archived test results. The tool should provide query capability to extract test results, analyze the test status and trend, and produce graphical reports of the test results. Finally, the tool should have the capability to collect and analyze response time and throughput as an aid to performance testing.

5. *Test Management Criteria:* A tool should have the ability to provide a test structure, or hierarchy, that allows test cases to be stored and retrieved in a manner that the test organization wants to organize. The tool should have the capability to allocate tests or groups of tests to specific test engineers and compare the work status with the plan through graphic display. A tool needs to have authorization features. For example, a test script developer may be authorized to create and update the test scripts, while the test executer can only access them in the run mode. The tool should have the capability to send out emails with the test results after completion of test suite execution.

6. *GUI Testing Capability Criteria:* An automated GUI test tool should include a record/playback feature which allows the test engineers to create, modify, and run automated tests across many environments. These tools should have a capability to recognize and deal with all types of GUI objects, such as list boxes, radio buttons, icons, joysticks, hot keys, and bit-map images with changes in color shades and presentation fonts. The recording activity of the tool capturing the keystrokes entered by the test engineer can be represented as scripts in a high-level programming language and saved for future replay. The tools must allow test engineers to modify test scripts to create reusable test procedures to be played back on a new software image for comparison. The performance of a GUI test tool needs to be evaluated. One may consider the question: How fast can the tool record and playback a complex test scenario or a group of test scenarios?

7. *Vendor Qualification Criteria:* Many questions need to be asked about the vendor's financial stability, age of the vendor company, and its capability to support the tool. The vendor must be willing to fix problems that arise with the tool. A future roadmap must exist for the product. Finally, the maturity and market share of the product must be evaluated.

8. *Pricing Criteria:* Pricing is an important aspect of the product evaluation criteria. One can ask a number of questions: Is the price competitive? Is it within the estimated price range for an initial tool purchase? For a large number of licenses, a pricing discount can be negotiated with the vendor. Finally, the license must explicitly cap the maintenance cost of the test tool from year to year.

Tool vendors may guarantee the functionality of the test tool; however, experience shows that often test automation tools do not work as expected within the particular test environment. Therefore, it is recommended to evaluate the test tool by using it before making the decision to purchase it. The test team leader needs to contact the tool vendor to request a demonstration. After a demonstration of the tool, if the test team believes that the tool holds potential, then the test team leader may ask for a temporary license of the tool for evaluation. At this point enough resources are allocated to evaluate the test tool. The evaluator should have a clear understanding of the tool requirements and should make a test evaluation plan based on the criteria outlined previously. The goal here is to ensure that the test tool performs as advertised by the vendor and that the tool is the best product for the requirement. Following the hands-on evaluation process, an evaluation report is prepared. The report documents the hands-on experience with the tool. This report should contain background information, a tool summary, technical findings, and a conclusion. This document is designed to address the management concerns because eventually it has to be approved by executive management.

12.12 TEST SELECTION GUIDELINES FOR AUTOMATION

Test cases should be automated only if there is a clear economic benefit over manual execution. Some test cases are easy to automate while others are more cumbersome.

The general guideline shown in Figure 12.4 may be used in evaluating the suitability of test cases to be automated as follows:

Less Volatile: A test case is stable and is unlikely to change over time. The test case should have been executed manually before. It is expected that the test steps and the pass–fail criteria are not likely to change any more.

Repeatability: Test cases that are going to be executed several times should be automated. However, one-time test cases should not be considered for automation. Poorly designed test cases which tend to be difficult to reuse are not economical for automation.

High Risk: High-risk test cases are those that are routinely rerun after every new software build. The objectives of these test cases are so important that one cannot afford to not reexecute them. In some cases the propensity of the test cases to break is very high. These test cases are likely to be fruitful in the long run and are the right candidates for automation.

Easy to Automate: Test cases that are easy to automate using automation tools should be automated. Some features of the system are easier to test than other features, based on the characteristics of a particular tool. Custom objects with graphic and sound features are likely to be more expensive to automate.

Manually Difficult: Test cases that are very hard to execute manually should be automated. Manual test executions are a big problem, for example, causing eye strain from having to look at too many screens for too long in a GUI test. It is strenuous to look at transient results in real-time applications. These nasty, unpleasant test cases are good candidates for automation.

Boring and Time Consuming: Test cases that are repetitive in nature and need to be executed for longer periods of time should be automated. The tester's time should be utilized in the development of more creative and effective test cases.

Figure 12.4 Test selection guideline for automation.

12.13 CHARACTERISTICS OF AUTOMATED TEST CASES

The largest component of test case automation is programming. Unless test cases are designed and coded properly, their execution and maintenance may not be effective. The design characteristics of effective test cases were discussed in Chapter 11. A formal model of a standard test case schema was also provided in Chapter 11. In this section, we include some key points which are pertinent to the coding of test cases. The characteristics of good automated test cases are given in Figure 12.5 and explained in the following.

1. *Simple:* The test case should have a single objective. Multiobjective test cases are difficult to understand and design. There should not be more than 10–15 test steps per test case, excluding the setup and cleanup steps. Multipurpose test cases are likely to break or give misleading results. If the execution of a complex test leads to a system failure, it is difficult to isolate the cause of the failure.

2. *Modular:* Each test case should have a setup and cleanup phase before and after the execution test steps, respectively. The setup phase ensures that the initial conditions are met before the start of the test steps. Similarly, the cleanup phase puts the system back in the initial state, that is, the state prior to setup. Each test step should be small and precise. One input stimulus should be provided to the system at a time and the response verified (if applicable) with an interim verdict. The test steps are building blocks from reusable libraries that are put together to form multistep test cases.

3. *Robust and Reliable:* A test case verdict (pass–fail) should be assigned in such a way that it should be unambiguous and understandable. Robust test cases can ignore trivial failures such as one-pixel mismatch in a graphical display. Care should be taken so that false test results are minimized. The test cases must have built-in mechanisms to detect and recover from errors. For example, a test case

Figure 12.5 Characteristics of automated test cases.

need not wait indefinitely if the SUT has crashed. Rather, it can wait for a while and terminate an indefinite wait by using a timer mechanism.

4. *Reusable:* The test steps are built to be configurable, that is, variables should not be hard coded. They can take values from a single configurable file. Attention should be given while coding test steps to ensure that a single global variable is used, instead of multiple, decentralized, hard-coded variables. Test steps are made as independent of test environments as possible. The automated test cases are categorized into different groups so that subsets of test steps and test cases can be extracted to be reused for other platforms and/or configurations. Finally, in GUI automation hard-coded screen locations must be avoided.

5. *Maintainable:* Any changes to the SUT will have an impact on the automated test cases and may require necessary changes to be done to the affected test cases. Therefore, it is required to conduct an assessment of the test cases that need to be modified before an approval of the project to change the system. The test suite should be organized and categorized in such a way that the affected test cases are easily identified. If a particular test case is data driven, it is recommended that the input test data be stored separately from the test case and accessed by the test procedure as needed. The test cases must comply with coding standard formats. Finally, all the test cases should be controlled with a version control system.

6. *Documented:* The test cases and the test steps must be well documented. Each test case gets a unique identifier, and the test purpose is clear and understandable. Creator name, date of creation, and the last time it was modified must be documented. There should be traceability to the features and requirements being checked by the test case. The situation under which the test case cannot be used is clearly described. The environment requirements are clearly stated with the source of input test data (if applicable). Finally, the result, that is, pass or fail, evaluation criteria are clearly described.

7. *Independent and Self-Sufficient:* Each test case is designed as a cohesive entity, and test cases should be largely independent of each other. Each test case consists of test steps which are naturally linked together. The predecessor and successor of a test step within a test case should be clearly understood. It is useful to keep the following three independence rules while automating test cases:

- *Data value independent:* The possible corruption of data associated with one test case should have no impact on other test cases.
- *Failure independent:* The failure of one test case should not cause a ripple of failures among a large number of subsequent test cases.
- *Final state independent:* The state in which the environment is left by a test case should have no impact on test cases to be executed later.

We must take into consideration the characteristics outlined in this section during the development of test scripts. In addition, we must obey the syntax of the test case defined in the next section while implementing a test case.

12.14 STRUCTURE OF AN AUTOMATED TEST CASE

An automated test case mimics the actions of a human tester in terms of creating initial conditions to execute the test, entering the input data to drive the test, capturing the output, evaluating the result, and finally restoring the system back to its original state. The six major steps in an automated test case are shown in Figure 12.6. Error handling routines are incorporated in each step to increase the maintainability and stability of test cases.

Setup: The setup includes steps to check the hardware, network environment, software configuration, and that the SUT is running. In addition, all the parameters of the SUT that are specific to the test case are configured. Other variables pertinent to the test case are initialized.

Drive the Test: The test is driven by providing input data to the SUT. It can be a single step or multiple steps. The input data should be generated in such a way that the SUT can read, understand, and respond.

Capture the Response: The response from the SUT is captured and saved. Manipulation of the output data from the system may be required to extract the information that is relevant to the objective of the test case.

Determine the Verdict: The actual outcome is compared with the expected outcome. Predetermined decision rules are applied to evaluate any discrepancies between the actual outcome against the expected outcome and decide whether the test result is a pass or a fail. If a fail verdict is assigned to the test case, additional diagnostic information is needed. One must be careful in designing the rules for assigning a passed/failed verdict to a test case. A failed test procedure does not necessarily indicate a problem with the SUT—the problem could be a *false positive*. Similarly, a passed test procedure does not necessarily indicate that there is no problem with the SUT—the problem could be due to a *false negative*. The problems of false negative and false positive can occur due to several reasons,

Figure 12.6 Six major steps in automated test case.

such as setup errors, test procedure errors, test script logic errors, or user errors [18].

Log the Verdict: A detailed record of the results are written in a log file. If the test case failed, additional diagnostic information is needed, such as environment information at the time of failure, which may be useful in reproducing the problem later.

Cleanup: A cleanup action includes steps to restore the SUT to its original state so that the next test case can be executed. The setup and cleanup steps within the test case need to be efficient in order to reduce the overhead of test execution.

12.15 TEST AUTOMATION INFRASTRUCTURE

A test automation infrastructure, or framework, consists of test tools, equipment, test scripts, procedures, and people needed to make test automation efficient and effective. The creation and maintenance of a test automation framework are key to the success of any test automation project within an organization. The implementation of an automation framework generally requires an automation test group, as discussed in Chapter 16. The six components of a test automation framework are shown in Figure 12.7. The idea behind an automation infrastructure is to ensure the following:

- Different test tools and equipment are coordinated to work together.
- The library of the existing test case scripts can be reused for different test projects, thus minimizing the duplication of development effort.
- Nobody creates test scripts in their own ways.
- Consistency is maintained across test scripts.
- The test suite automation process is coordinated such that it is available just in time for regression testing.
- People understand their responsibilities in automated testing.

System to Be Tested: This is the first component of an automation infrastructure. The subsystems of the system to be tested must be stable; otherwise test automation will not be cost effective. As an example, the 1xEV-DO system described in Chapter 8 consists of three subsystems, BTS, BSC, and EMS. All three subsystems must be stable and work together as a whole before the start of an automation test project.

Test Platform: The test platform and facilities, that is, the network setup on which the system will be tested, must be in place to carry out the test automation project. For example, a procedure to download the image of the SUT, configuration management utilities, servers, clients, routers, switches, and hubs are necessary to set up the automation environment to execute the test scripts.

Figure 12.7 Components of automation infrastructure.

Test Case Library: It is useful to compile libraries of reusable test steps of basic utilities to be used as the building blocks of automated test scripts. Each utility typically performs a distinct task to assist the automation of test cases. Examples of such utilities are *ssh* (secure shell) from client to server, *exit* from client to server, response capture, information extraction, rules for verdicts, verdict logging, error logging, cleanup, and setup.

Automated Testing Practices: The procedures describing how to automate test cases using test tools and test case libraries must be documented. A template of an automated test case is useful in order to have consistency across all the automated test cases developed by different engineers. A list of all the utilities and guidelines for using them will enable us to have better efficiency in test automation. In addition, the maintenance procedure for the library must be documented.

Tools: Different types of tools are required for the development of test scripts. Examples of such tools are test automation tool, traffic generation tool, traffic monitoring tool, and support tool. The support tools include test factory, requirement analysis, defect tracking, and configuration management tools. Integration of test automation and support tools, such as defect tracking, is crucial for the automatic reporting of defects for failed test cases. Similarly, the test factory tool can generate automated test execution trends and result patterns.

Administrator: The automation framework administrator (i) manages test case libraries, test platforms, and test tools; (ii) maintains the inventory of templates; (iii) provides tutorials; and (iv) helps test engineers in writing test scripts using the test case libraries. In addition, the administrator provides tutorial assistance to the users of test tools and maintains a liaison with the tool vendors and the users.

12.16 SUMMARY

This chapter focused on the importance of a system test plan. We presented an outline of a system test plan which consists of introduction, feature description, assumptions, approach, test suite structure, test environment, execution strategy, test effort estimation, and scheduling and milestones. We discussed the processes to create such a plan.

The chapter presented an effective process model for performing software system testing. The key idea in this process model is to raise the quality of a system from its initial low level to the final high level in a number of controlled steps called test cycles. We characterized a test cycle in terms of six parameters: goals, assumptions, test execution, revert and extension criteria, action, and exit criteria. These parameters allow test managers to effectively control the progress of system testing so that the system moves one step closer to the final quality level. We explained an entry criterion that tells us when system testing should start. We presented new techniques for prioritization of test cases from test cycle to test cycle and selection of test cases for regression testing in the final test cycle. Typical values of the parameters were given for three-cycle-based test projects.

We discussed techniques, such as function points, to estimate the number of test cases for a test project with concrete examples. Guidelines to estimate the time required to create and execute system-level test cases were proposed. We presented ways of using a Gantt chart to create a system test schedule and milestones for the test project.

Next, we outlined the benefits of test automation: (i) test engineer productivity, (ii) coverage of regression testing, (iii) reusability of test cases, (iv) consistency in testing, (v) test interval reduction, (vi) reduction of software maintenance cost, and (vii) increased test effectiveness. The following five prerequisites must be considered while assessing whether or not a test organization is ready for test automation:

- The system is stable and the functionalities are well defined.
- The test cases that need to be automated are well defined.
- The test tools and infrastructure are in place.
- The test automation professionals have prior successful experience with automation.
- Adequate budget was allocated for the procurement of software tools.

We presented a process to evaluate and select third-party test tools available in the market. The test tool evaluation criteria are classified into eight categories: (i) test development, (ii) test maintenance, (iii) test execution, (iv) test results, (v) test management, (vi) GUI testing capability, (vii) vendor qualification and (viii) pricing. These criteria are formulated in order to select the right kind of software test tool. We advocated that test engineers should build their own test automation tools only if they have no alternative. Building and maintaining one's own test automation tool from scratch are time consuming and an expensive undertaking. There is no point in

reinventing the wheel if off-the-shelf test tools satisfying the criteria are readily available.

We provided general guidelines to be used in the selection of test cases that can be automated. The characteristics of automatable test cases are as follows: less volatile, repeatability, high risk, easy to automate, manually difficult to execute, and boring and time consuming. We discussed the characteristics of automated test cases: simple, modular, reusable, maintainable, documented, robustness, reliable, and independent and self-sufficient.

Finally, we described the structure of an automated test case, which consists of setup, driving the test, capturing the response, determining the verdict, logging the verdict, and cleanup. We explained the need of a test automation infrastructure. We provided six components of a test automation framework: (i) the system to be tested, (ii) the test platform, (iii) the test case library, (iv) the automated testing practices, (v) the tools, and (vi) the administrator.

LITERATURE REVIEW

A generic test plan template has been described in the IEEE standard 829-1983 (*IEEE Standard for Software Test Documentation: IEEE/ANSI Standard*). Other topics described in the standard are test design specifications, test case specifications, test procedure specifications, test item transmittal report, and test result reports. The following are the sections of a test plan as defined in the standard:

- **Test Plan Identifier:** A unique identifier, useful if all the documents are stored in a database.
- **Introduction:** Overall description of the test project. This includes references to all the relevant documents.
- **Items to be Tested:** List of entities (function, module, systems, and subsystems) to be tested. The list includes references to specifications and manuals.
- **Features to be Tested:** Identification of the features to be tested. Cross-reference them to the test design specifications.
- **Features Not to be Tested:** Which ones and why not.
- **Approach:** Description of the overall approach to testing. The standard also says that this section—and not the schedule section—is the place to identify constraints, including deadlines and availability of resources (human and equipment).
- **Pass/Fail Criteria:** A set of criteria to decide the test case has been passed or failed upon execution.
- **Suspension and Resumption Criteria:** List of anything that would cause testing to stop until it is fixed. What would have to be done to resume testing? What tests should be reexecuted at this point?

- **Test Deliverables:** List of all testing documents that will be written for this project.
- **Testing Tasks:** List of all tasks necessary to prepare and conduct testing. Show dependency between tasks, special skills needed to do them, who does each task, how much effort is involved, and when each will be done.
- **Test Environment:** Description of the hardware, software, testing tool, and lab facilities required to conduct the testing.
- **Responsibilities:** Name of the engineers (or groups) responsible for managing, designing, preparing, and executing the tests.
- **Staffing and Training Needs:** How many people are required? What training they need?
- **Scheduling:** List of all the milestones with dates and when all the resources will be needed.
- **Risks and Contingencies:** What are the highest risk assumptions in the test plan? What can go sufficiently wrong to delay the schedule, and what can be done about it?
- **Approvals:** Who has to approve this plan? Provide space for their signatures.

Cangussu et al. [2] have studied a general testing process by using a state-variable-based model. By mathematically quantifying the effects of parameter variations on the behavior of a system test process, the system test manager can estimate the impact of parameter variation on the test schedule.

Several regression test selection techniques have been proposed by researchers to reduce software maintenance costs. The test selection techniques proposed for regression testing can be broadly classified into two categories based on the kinds of information they utilize to perform the task: *specification* based and *code* based. The following articles discuss the selection of regression tests utilizing information from specification and code.

J. Bible, G. Rothermel, and D. S. Rosenblum "A Comparative Study of Coarse- and Fine-Grained Safe Regression Test-Selection Techniques," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 10, No. 2, 2001, pp. 149–183.

D. Binkley, "Semantics Guided Regression Test Cost Reduction," *IEEE Transactions on Software Engineering*, Vol. 23, No. 8, August 1997, pp. 498–516.

T. L. Graves, M. J. Harrold, J. M. Kim, A. Porter, and G. Rothermel, "An Empirical Study of Regression Test Selection Techniques," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 10, No. 2, 2001, pp. 184–208.

G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qui, "On Test Suite Composition and Cost-Effective Regression Testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 13, No. 3, 2004, pp. 277–331.

G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Transactions on Software Engineering*, Vol. 22, No. 8, August 1996, pp. 529–551.

G. Rothermel and M. J. Harrold, "Empirical Studies of a Safe Regression Test Selection Technique," *IEEE Transactions on Software Engineering*, Vol. 24, No. 6, June 1998, pp. 401–419.

In addition, several *code*-based techniques have been proposed by Rothermel, et al. [4] to prioritize regression tests by using information gathered from previous execution of test cases. Those techniques are fine grained in the sense that prioritization is done at the level of source code statements. Elbaum, et al. [5] have proposed 12 coarse-grained techniques for test prioritization at the function level for regression testing. They suggest that it may be desirable to change the priority of test cases from version to version. Test case prioritization and selection algorithms for regression testing with the goal of achieving modified condition/decision coverage have been studied by Jones and Harrold [6].

Computer-aided software engineering (CASE) tools are continuously evolving. Tool evaluation processes continue to evolve as the tools continue to mature and the technology advances. The following two articles give managers and tool evaluators a reliable way to identify tools that best fit their needs:

V. Mosley, "How to Assess Tools Efficiently and Quantitatively," *IEEE Software*, May 1992, pp. 29–32.

R. M. Poston and M. P. Sexton, "Evaluating and Selecting Testing Tools," *IEEE Software*, May 1992, pp. 33–42.

A voluminous survey of test automation tools can be found in Appendix B of the book by Dustin, et al. [18]. The book also outlines a number of excellent test automation tools to improve a testing process.

REFERENCES

1. P. Tripathy and K. Naik. A Multi-Cycle Based Execution Strategy for Software System Testing, TR:2007-L086. NEC Laboratories America Electronic TR/TN Distribution, Princeton, NJ, June 2007, pp. 1–45.
2. J. W. Cangussu, R. A. Decarlo, and A. P. Mathur. Using Sensitivity Analysis to Validate a State Variable Model of the Software Test Process. *IEEE Transactions on Software Engineering*, June 2003, pp. 430–443.
3. N. E. Fenton and M. Neil. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*, September/October 1999, pp. 675–689.
4. G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing Test Cases for Regression Testing. *IEEE Transactions on Software Engineering*, October 2001, pp. 929–948.
5. S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test Case Prioritization: A Family of Empirical Studies. *IEEE Transactions on Software Engineering*, February 2002, pp. 159–182.
6. J. A. Jones and M. J. Harrold. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. *IEEE Transactions on Software Engineering*, March 2003, pp. 195–209.

7. A. Srivastava and J. Thiagarajan. Effectively Prioritizing Tests in Development Environment. Paper presented at the International Symposium on Software Testing and Analysis (ISSTA), Roma, Italy, ACM Press, New York, 2002, pp. 97–106.
8. G. M. Kapfhammer. Automatically and Transparently Distributing the Execution of Regression Test Suites. Paper presented at the 18th International Conference on Testing Computer Software, 2001, pp. 1–16.
9. B. Littlewood and D. Wright. Some Conservative Stopping Rules for the Operational Testing of Safety-Critical Software. *IEEE Transactions on Software Engineering*, November 1997, pp. 673–683.
10. S. H. Kan, J. Parrish, and D. Manlove. In-Process Metrics for Software Testing. *IBM Systems Journal*, January 2001, pp. 220–241.
11. T. Yamaura. How to Design Practical Test Cases. *IEEE Software*, November/December 1998, pp. 30–36.
12. Y. Chernak. Validating and Improving Test-Case Effectiveness. *IEEE Software*, January/February 2001, pp. 81–86.
13. E. Yourdon. When Good Enough Software Is Best. *IEEE Software*, May 1995, pp. 79–81.
14. A. J. Albrecht. Measuring Application Development Productivity. Paper presented at the Process Joint SHARE/GUIDE/IBM Application Development Symposium, October 1979, pp. 34–43.
15. A. J. Albrecht and J. E Gaffney. Software Functions Source Lines Code and Development Efforts Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, November 1983, pp. 639–648.
16. C. Jones. *Estimating Software Costs*. McGraw-Hill, New York, 1998.
17. C. Jones. *Applied Software Measurement*. McGraw-Hill, New York, 1996.
18. E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley, Reading, MA, 1999.

Exercises

1. What is the purpose of having a system test plan ready before the start of system test cycles?
2. What are some essential items a tester should include in a system test plan?
3. What are the advantages and disadvantages of having separate unit, integration, and system test plans as opposed to an inclusive test plan containing all three in the same document?
4. Study the test plan proposed in the IEEE standard 829 standard described in the literature review section of this chapter. What are the essential items not addressed in this test plan?
5. Why are multiple test environments set up for the execution of system-level test cases? Why should these test environments not be shared for software development activities?
6. What are the parameters of the multicycle system test execution strategy? What is the idea behind the parameterized system test cycle?
7. Why must the entry criteria be satisfied before the start of the first system test cycle?
8. What are the objectives of the first, second, and the third system test cycles?
9. If the test case failure rate is high, why does it slow down the test execution rate?
10. Develop a system test plan for your current test project.

11. What are the advantages and disadvantages of early test automation, that is, for the first release of a product?
12. What are the advantages and disadvantages of late test automation, that is, after the first release of the product?
13. Develop evaluation criteria for the selection of a defect tracking tool.
14. Evaluate the ClearQuest defect tracking system by getting a trial copy of the tool from IBM using the criteria developed for exercise 13.
15. What are the components of a test automation infrastructure? What is the role of a test automation framework administrator?

CHAPTER 13

System Test Execution

Execute every act of thy life as though it were thy last.
— *Marcus Aurelius*

13.1 BASIC IDEAS

Preparing for and executing system-level tests are a critical phase in a software development process because of the following: (i) There is pressure to meet a tight schedule close to the delivery date; (ii) there is a need to discover most of the defects before delivering the product; and (iii) it is essential to verify that defect fixes are working and have not resulted in new defects. It is important to monitor the processes of test execution and defect fixing. To be able to monitor those test processes, we identify two key categories of metrics: (i) *system test execution status* and (ii) *defects status*. We provide a detailed *defect schema*, which includes a general FSM model of defects for ease of collecting those metrics. Analyzing defects is a crucial activity performed by the software development team while fixing defects. Therefore, in this chapter, we describe three types of defect analysis techniques: *causal, orthogonal*, and *Pareto methodology*. In addition, we provide three metrics, namely, *defect removal efficiency, spoilage*, and *fault seeding*, to measure test effectiveness. The objective of a test effectiveness metric is to evaluate the effectiveness of a system testing effort in the development of a product in terms of the number of defects escaped from the system testing effort.

The product is ready for beta testing at the customer site during the system testing. We provide a framework for beta testing and discuss how beta testing is conducted at the customer's site. Moreover, we provide a detailed structure of the system test execution report, which is generated before a product's *general availability* is declared.

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

13.2 MODELING DEFECTS

The key to a successful defect tracking system lies in properly modeling defects to capture the viewpoints of their many stakeholders, called cross-functionality groups. The cross-functionality groups in an organization are those groups that have different stakes in a product. For example, a marketing group, a customer support group, a development group, a system test group, and a product sustaining group are collectively referred to as cross-functionality groups in an organization. It is not enough to merely report a defect from the viewpoint of software development and product management and seek to understand it by means of reproduction before fixing it. In reality, a reported defect is an evolving entity that can be appropriately represented by giving it a life-cycle model in the form of a state transition diagram, as shown in Figure 13.1. The states used in Figure 13.1 are briefly explained in Table 13.1.

The state transition model allows us to represent each phase in the life cycle of a defect by a distinct state. The model represents the life cycle of a defect from its initial reporting to its final closing through the following states: new, assigned, open, resolved, wait, FAD, hold, duplicate, shelved, irreproducible, postponed, and closed. When a defect moves to a new state, certain actions are taken by the owner of the state. By "owner" of a state of a defect we mean the person or group of people who are responsible for taking the required actions in that state. Once the associated actions are taken in a state, the defect is moved to a new state.

Two key concepts involved in modeling defects are the levels of *priority* and *severity*. On one hand, a priority level is a measure of how soon the defect needs to be fixed, that is, urgency. On the other hand, a severity level is a measure of the extent of the detrimental effect of the defect on the operation of the product. Therefore, priority and severity assignments are separately done. In the following, four levels of defect priority are explained:

Figure 13.1 State transition diagram representation of life cycle of defect.

TABLE 13.1 States of Defect Modeled in Figure 13.1

State	Semantic	Description
N	New	A problem report with a severity level and a priority level is filed.
A	Assigned	The problem is assigned to an appropriate person.
O	Open	The assigned person is actively working on the problem to resolve it.
R	Resolved	The assigned person has resolved the problem and is waiting for the submitter to verify and close it.
C	Closed	The submitter has verified the resolution of the problem.
W	Watt	The assigned person is waiting for additional information from the submitter.
F	FAD	The reported defect is not a true defect. Rather, it is a function as designed.
H	Hold	The problem is on hold because this problem cannot be resolved until another problem is resolved.
S	Shelved	There is a problem in the system, but a conscious decision is taken that this will not be resolved in the near future. Only the development manager can move a defect to this state.
D	Duplicate	The problem reported is a duplicate of a problem that has already been reported.
I	Irreproducible	The problem reported cannot be reproduced by the assigned person.
P	Postponed	The problem will be resolved in a later release.

1. *Critical*: Defects at this level must be resolved as soon as possible. Testing should not progress until a known critical defect is fixed.
2. *High*: Defects at this level must be resolved with high priority. A subset of the system functionalities cannot be tested unless this level of defect is resolved.
3. *Medium*: Defects at this level will not have any impact on the progress of testing. Medium-priority defects stand by themselves.
4. *Low*: Defects at this level should be resolved whenever it is possible.

Intuitively, a critical-priority defect affects a large number of functionalities of a system, a high-priority defect affects a smaller subset of functionalities, a medium-priority defect affects an individual functionality, and a low-priority defect is considered to be a minor irritation. In the following, four levels of severity are explained:

1. *Critical*: A defect receives a "critical" severity level if one or more critical system functionalities are impaired by a defect with is impaired and there is no workaround.
2. *High*: A defect receives a "high" severity level if some fundamental system functionalities are impaired but a workaround exists.
3. *Medium*: A defect receives a "medium" severity level if no critical functionality is impaired and a workaround exists for the defect.

4. *Low*: A defect receives a "low" severity level if the problem involves a cosmetic feature of the system.

Intuitively, defects with high severity should receive a high priority in the resolution phase, but a situation may arise where a high-severity defect may be given a low priority. For example, let a system crash when the system date is set to December 32. Though a crash defect has severe consequences, an occurrence of December 32 as the system date is unlikely in normal circumstances. The said defect receives a critical severity but a low priority. In other words, if a critically severe defect occurs with an extremely low probability, then it may not be fixed immediately, that is, it may be treated with low priority.

Someone reporting a defect makes a preliminary assessment of the severity and priority levels of the defect. If there is a dispute among the different cross-functional groups concerning the priority and severity levels of a defect, an agreement can be reached in a regular review meeting. Note that the severity of a defect remains the same throughout its life cycle, whereas the priority level may change as the software product approaches its release date. A defect may be assigned a low priority at the beginning of system testing if it may not have any impact on the progress of testing, but it must be fixed before the software is released. For example, a spelling mistake in the GUI may be a low-priority defect at the beginning of system testing but its priority becomes high as the release date approaches.

In the following, we describe the states of a defect modeled as a state transition diagram, as shown in Figure 13.1, and explain how a defect moves from state to state. When a new defect is found, it is recorded in a defect tracking system, and the defect is put in its initial state new. The owner of this state is the software development manager. In this state, the following fields, explained as parts of a schema given in Table 13.2, are initialized: defect_id, submit_date, product, submitter, group, owner, headline, keywords, severity, priority, reproducible, category, software_version, build, description, h/w_configuration, s/w_configuration, attachments, notes, and number_tc_fail, tc_id.

The software development manger, who is the owner of the new state, moves the defect from the new state to the assigned state by changing the ownership either to a software developer or to himself. The software development manager may reject this defect by explaining why it is not a true defect and move the defect to the FAD state with a change in ownership back to the submitter. The development manager may change the state to the shelved state, which means that it is a true defect and a conscious decision has been made to the effect that this defect will not be fixed in the near future. All the relevant parties, especially the customer support group of the organization, must agree to shelve the defect before a defect is moved to the shelved state. There are several reasons to move a defect to the shelved state. For example, a defect may not have much impact on system operation in the customer's operational environment, and it may consume a lot of resources to fix it. In practice, very few defects land in the shelved state.

The assigned state means that someone has been assigned to fix the defect but the actual work required to fix the defect has not begun. The software development

TABLE 13.2 Defect Schema Summary Fields

Field Name	Description
defect_id	A unique, internally generated identifier for the defect
state	Current state of the defect; takes a value from the set {New, Assigned, Open, Resolved, Information, FAD, Hold, Duplicate, Shelved, Irreproducible, Postponed, Closed}
headline	One-line summary of defect
severity	Severity level of defect; takes a value from the set {critical, high, medium, low}
priority	Priority level of defect; takes a value from the set {critical, high, medium low}
submitter	Name of the person who submits the defect
group	Group affiliation of submitter; takes a value from the set {ST, SIT, software, hardware, customer support}
owner	Current owner of defect
reproducible	Says, in terms of yes or no, whether or not defect can be reproduced
crash	Says, in terms of yes or no, whether or not defect causes system to crash
keywords	Some common words that can be associated with this defect for searching purpose
product	Name of product in which defect was found
category	Test category name that revealed defect
software_version	Software version number in which defect was observed
build	Build number in which defect was observed
submit_date	Date of submission of defect
description	Brief description of defect
h/w_configuration	Description of hardware configuration of test bed
s/w_configuration	Description of software configuration of test bed
attachments	Attachments, in the form of log files, configuration files, etc., useful in understanding the defect
notes	Additional notes or comments, if there are any
number_tc_fail	Number of test cases failed or blocked because of defect
tc_id	List of test case identifiers of those test cases, from test factory database, which will fail because of defect
forecast_fix_version	Software version in which fix for defect will be available
forecast_build_number	Build number in which fix for defect will be available
actual_fix_version	Software version in which fix is actually available
actual_build_number	Build number in which fix is actually available
fix_description	Brief description of fix for defect
fix_date	Date when fix was checked in to code
duplicate_defect_id	Present defect considered to be duplicate of duplicate defect_id
requirement_id	Requirement identifier from requirement database that is generated as result of agreement that defect be turned into requirement

manager, or a developer, is the owner of the defect in the assigned state. The ownership may change to a different software developer who will fix the defect. The assigned software developer may move the defect from the assigned state to the open state as soon as the process of defect fixing begins. Note that only the software development manager moves a defect from the assigned state to one the following states: shelved, postponed, and FAD. The ownership is changed to the submitter when a defect is moved to the FAD state.

A software developer, the owner of the defect, is actively working to fix the defect in the open state. The developer initializes the forecast_fix_version and forecast_build_number fields of the defect schema given in Table 13.2 so that the submitter can plan and schedule a retest of the fix. A defect may lie in the open state for several weeks. The software developer moves the defect from this state to five possible next states—irreproducible, resolved, wait, hold, and duplicate—which are explained in Table 13.3 along with the actions that need to be taken when the state is changed.

If a software developer working on a defect is satisfied that the defect has been fixed, he or she moves the defect to a resolved state and changes the ownership

TABLE 13.3 State Transitions to Five Possible Next States from Open State

Possible *Next* State	Actions
Irreproducible	If the owner, i.e., a developer, cannot reproduce a defect, the defect is moved to irreproducible state with the ownership changed back to the submitter.
Wait	If the amount of information given by the submitter is not enough to understand and/or reproduce the defect, then the owner asks for more information on the defect from the submitter. Essentially, the developer is asking for more information about the defect in the notes field. The ownership field is changed back to the submitter.
Resolved	**1.** The ownership field is changed back to the submitter. **2.** The following fields need to be filled out: **3.** Actual_fix_version: Software version where the fix is available **4.** Actual_build_number: Build number where the fix is available. **5.** Fixed_description: Brief description of the fix. The developer must list all the files that are modified and what has been changed in the files in order to fix the problem. **6.** Fixed_date: Date the defect was fixed on and moved to the resolved state.
Hold	It has to be explained in the notes field why the defect was moved to this state. The hold state means that no firm decision has been made regarding the defect because the problem depends on some other issues, such as a software defect in a third-party product. Unless the other issues are resolved, the problem described in the defect report cannot be resolved.
Duplicate	If this defect is identical to, i.e., a duplicate of another defect, then the identifier of the original defect is recorded in the duplicate_defect_id field.

of the defect back to the submitter. A defect in the resolved state merely implies that the defect has been fixed to the satisfaction of the developer, and the associated tests need to be executed to further verify the claim in a wider context. The submitter moves the defect from the resolved state to the closed state by initializing the following fields of the defect schema after it is verified that the defect has been fixed: verifier, closed_in_version, closed_in_build, closed_date, and verification_description. Essentially, verification_description describes the details of the verification procedure. The closed state signifies that the defect has been resolved and verified. On the other hand, if the verifier is convinced that the defect has not been actually or completely fixed, the fix is rejected by moving the defect back to the assigned state, and the ownership of the defect is changed back to the responsible software developer. The verifier provides the following information while rejecting a fix:

- Explanation for rejection of fix
- Any new observation and/or problem encountered

A defect is moved to the FAD state by the development team if the team concludes that it is not a true defect and the system behaves as expected. The submitter, the owner of the defect in the FAD state, may involve the development team, the customer support group, the product management, and the project leadear to review defects in this state. There are three possible alternative outcomes of a defect review meeting:

- The reporting of the defect was due to a procedural error on the part of the submitter. The defect remains in the FAD state for this outcome. The idea behind keeping the defect in the FAD state, and not in the closed state, is to record the fact that the submitter had designated it as a defect.
- The reported defect is a true defect. The defect is moved to the assigned state with a note for the software development manager saying that this is a defect agreed to by all the parties.
- A suggestion is made that a new feature enhancement request be made to accommodate the system behavior perceived to be a "defect" at this moment. The submitter moves the defect to the assigned state with a note for the development manager saying that the defect may be resolved by creating a new requirement and submitting the requirement identifier in the requirement_id field of the defect schema.

If a defect cannot be fixed in a particular release, it is moved to the postponed state, where the development manager is the owner of the defect. Fixing a defect is postponed due to lack of time to fix it for a certain release. The defect is moved, after the release for its fix is known, to the assigned state, where the following fields of the defect schema are initialized: (i) forecast_build_number—the build in which a fix will be available for testing—and (ii) forecast_fix_version—the version in which a fix will be available.

All the outstanding defects that are still open are moved to the postponed state after the product is released to the customer, so that the defects can be scheduled

to be fixed in future software releases. The defects are moved to the assigned state after it is finalized in which release these defects will be fixed. The postponed state is useful in tracking and scheduling all the outstanding defects in the system that must be fixed in future releases.

If a developer finds that the information provided by the submitter is not adequate to fix a defect, the developer asks for additional information from the submitter by moving the defect from the open state to the wait state. In the wait state, the submitter is the owner of the defect. The submitter provides the information asked for by the developer by initializing the notes field of the defect schema, moves the defect to the assigned state, and changes the ownership back to the software developer.

The developer may try to reproduce the symptom to understand the defect. If the developer cannot reproduce the reported symptom, the defect is moved to the irreproducible state with the ownership changed back to the submitter. The submitter makes an effort to reproduce the defect, collect all the relevant information into the defect report, and move the defect to the assigned state if the defect is reproduced. If the submitter cannot reproduce the defect, the defect stays in the irreproducible state.

It may be observed that the defect is caused by an extraneous factor, such as a defect in a third-party product which is being used. The reported defect cannot be fixed unless the defect in the outside component is resolved. In such a case, the defect is put on hold by moving it to the hold state, where the software development manager is the owner of the defect. The software development manager moves the defect to either the resolved state or the open state depending on if the primary defect in the outside component has been resolved or is being fixed, respectively.

A defect in the duplicate state, where the submitter is the owner, means that the problem is a duplicate of a defect reported earlier. The original defect identifier is used to initialize the schema field duplicate_defect_id. Once the duplicate defect is in the closed state, the submitter must verify this defect. If the verification is successful, the submitter moves the defect to the closed state. On the other hand, if the verification fails, the submitter rejects the duplicity claim by moving the defect back to the assigned state, where the software developer is the owner of the defect. The verifier must give a reason for rejecting a defect as a duplicate defect. This information contains any observation and/or problem encountered while verifying the defect.

13.3 PREPAREDNESS TO START SYSTEM TESTING

The status of the entry criteria outlined in Chapter 13 must be tracked at least four weeks before the start of the first system test cycle. Any exceptions to these criteria must be noted and discussed at the weekly project status review meeting. We discuss the last item of the entry criteria, namely that the test execution working document is in place and complete. A framework of such a document is outlined in Table 13.4 and is explained subsequently. This working document is created, controlled, and tracked by the test team leader.

TABLE 13.4 Outline of Test Execution Working Document

1. Test engineers
2. Test cases allocation
3. Test beds allocation
4. Automation progress
5. Projected test execution rate
6. Execution of failed test cases
7. Development of new test cases
8. Trial of system image
9. Schedule of defect review meeting

The test engineers section contains the names of the test engineers assigned to this project as well as their availability and expertise in the specific areas of the project. The training requirement for each team members that may be necessary to successfully execute the test project is noted in detail. The action plan and training progress for each team member are tracked in this section. Any issue related to human resource availability or training is referred to the software project lead.

The test cases are allocated among test engineers based on their expertise and interest after the test suite is created in the test factory. The software development engineers and system test engineers are identified and included in the sw_dev and tester fields of the test cases, respectively. The idea here is to give the ownership of the test case execution to the individual test engineers. A test engineer reviews the test cases allocated to him or her to understand them and prioritize them in consultation with the software developers. Priority of a test case is set to one of the values from the set {high, medium, low}. If necessary, a test engineer may update the test cases to fine tune the test procedures and the pass–fail criteria as discussed in Section 11.6. The activities of prioritization and allocation of test cases among engineers and software developers are tracked in the test case allocation section, and this must be completed before the start of the system test cycle. This is achieved by calling several meetings with the team members of the software project.

The test beds allocation section of the document states the availability of the number and kinds of test beds and the ways these test beds are distributed so that the test cases are executed concurrently. A mapping is established between the available test beds and the subgroups of test cases from the test suite. The mapping is based on the configuration requirements to execute the groups of test cases. The test engineer responsible for executing a subgroup of test cases is then assigned to that test bed. The test engineer owns that particular test bed and must ensure that the test bed is up and ready to go before the start of system test cycles. Precaution must be taken to ensure that each test engineer is allocated the same test bed throughout the entire test execution cycle. The idea here is to maximize the execution efficiency of the engineer. If necessary, new test beds are created, or allocated from other projects, so that the idle time is minimized.

In the automation progress section of the document, a table is created to track the progress of the automation of the test cases and the availability of those automated test cases for execution. This is of particular interest in the case of regression test cases that were developed earlier and executed manually, but their automation is in progress. The test team leader interacts with the responsible test automation engineers to gather the information on the test automation progress. In case the automated test cases are available on time, that is, before the start of the test cycle, then the scheduled manual execution of these test cases may be skipped. Instead, the automated test cases can be executed.

In the projected test execution rate section of the document, a table is created to show the number of test cases that are anticipated to be executed on a weekly basis during a test cycle. The projected number of test cases is tracked against the actual execution of test cases during the test cycle. As an example, let us consider a test project name, Bazooka, for which the total number of test cases selected is 1592. The projected execution of 1592 test cases on a weekly basis for a 14-week test cycle is shown in Table 13.2 in the form of a cumulative chart. The execution of the 1592 test cases is expected to take 14 weeks; 25 test cases will be executed in the first week; 75 test cases will be executed in the second week; thus, 100 test cases will executed by the end of the second week. As the execution proceeds, the second row of Figure 13.2, namely, actually executed test cases, is updated. This gives the test group an idea of the progress of the testing. If the number of actually executed test cases falls far behind the projected number for several weeks, the test manager will know that the project is going to be delayed.

A strategy for the execution of failed test cases and verification of defect fixes is outlined in the execution of failed test cases section of the document. There are two possibilities here: (i) The failed test cases are executed in the next test cycle and (ii) the failed test cases are executed in the current cycle as soon

	wk1	wk2	wk3	wk4	wk5	wk6	wk7	wk8	wk9	wk10	wk11	wk12	wk13	wk14
■ Projected execution	25	100	150	250	400	600	800	950	1100	1250	1400	1500	1550	1592
□ Actually executed														

Figure 13.2 Projected execution of test cases on weekly basis in cumulative chart form.

as the corresponding fixes are available. In the first case, all the fixes, which are already integrated into the system, are subject to a regression test. No fixes are inserted into the system midway through a test cycle which are available at the beginning of a test cycle. In the second scenario, the fixes are checked in to the version control system, and a new build is created. The build is then sanity tested by the system integration group, and then it is released to the system testing group to be retested by running the test cases which failed earlier. If a test case now passes, then the fix has been verified to have resolved the defect. However, any collateral damage may not have been detected. A fix causing collateral damage is called a "bad fix," A handful of passed test cases around the fix may be reexecuted after a previously failed test case is observed to have passed irrespective of a fix that has caused collateral damage. The idea is to identify the high-impact region of the fix, select a set of previously passed test cases in consultation with the software developer who fixed the defect, and then execute those tests as a regression test. The following information is collected and analyzed in selecting the regression test cases:

- Parts of the software modules that were modified in order to incorporate the fix
- Functionality areas that are closely related to the fix and susceptible to defects
- Kinds of defects likely to be introduced because of the fix

Based on the above information, existing or new test cases are selected that are most likely to uncover any possible new defects that might have been introduced.

The test engineers learn more about the system and are in a better position to develop additional test cases during the first test cycle of testing. This is in particular important when a repeatable problem is observed and there are no test scenarios in the test suite. In this case, a new test case needs to be added to the test factory since the long-term goal is to improve the coverage and effectiveness of the test suite. However, a decision must be made regarding the addition of a test case into the test factory during the test cycle or after completion of the test cycle.

The development of new test cases section of the document clarifies the strategy for writing new test cases during the execution of test cases. It should be communicated to the test engineers before the start of the test cycle so that one of the following actions is taken:

- Develop new test cases during the test cycle.
- Include a note in the defect report saying that a test case needs to be developed later for this defect and continue with the execution of test cases as per schedule.

It is recommended to take the second approach, since it is difficult to find time to write the test cases during test execution unless it is in the schedule. Therefore, the test engineer should go back to the individual defects and add test cases to the test factory based on the defect report after completion of the test cycle.

Quite often the test engineers download a software image from the system integration group to their allocated test beds at least two or three weeks prior to the official start of the first test cycle. This allows the test engineer to be familiar with the software image and to ensure that the test bed is fully operational. This activity is tracked for each system test engineer in the trial of system image section of the working document, even though the build is not officially released to start system testing. There are several advantages of using the trial software image before the official start date of the system test cycle:

- The test engineers are trained to become familiar with the system so that test execution productivity is higher during the system test cycle.
- To ensure that the test bed is operational without any downtime at the beginning of the system test cycle.
- The basic test cases can be executed earlier to validate the test steps.

A schedule of the defect review meeting must be in place before the start of the system test cycle. The schedule is established for the entire test cycle duration before the start of the system test. The test leader may use a Gantt chart to represent a schedule and include it in the schedule the defect review meeting section of the test execution working document. The cross-functional team members from software development, hardware development, and the customer support groups are invited to this meeting.

13.4 METRICS FOR TRACKING SYSTEM TEST

The system test execution brings forth (three) different facets of software development. The developers would like to know the degree to which the system meets the explicit as well as implicit requirements. The delivery date cannot be precisely predicted due to the uncertainty in fixing the problems. The customer is excited to take the delivery of the product. It is therefore a highly visible and exciting activity. At this stage, it is desirable to monitor certain metrics which truly represent the progress of system testing and reveal the quality level of the system. Based on those metrics, the management can trigger actions for corrective and preventive measures. By putting a small but critical set of metrics in place executive management is in a position to know whether they are on the right track [1]. We make a difference between *statistics* and *in-process* metrics: Statistics are used for postproject analysis to gather experience and use it in future projects, whereas in-process metrics let us monitor the progress of the project while we still have an opportunity to steer the course of the project. By considering three large, real-life test projects, we categorize execution metrics into two classes:

- For monitoring test execution
- For monitoring defects

The first class of metrics concerns the process of executing test cases, whereas the second class concerns the defects found as a result of test execution. These

metrics need to be tracked and analyzed on a periodic basis, say, daily or weekly. It is important to gather valid and accurate information about the project for the leader of a system test group to effectively control a test project. One example of effective control of a test project is to precisely determine the time to trigger the revert criteria for a test cycle and initiate root cause analysis of the problems before more tests are performed. A test manager can effectively utilize the time of test engineers by triggering such a revert criterion, and possibly money, by suspending a test cycle on a product with too many defects to carry out a meaningful system test. Therefore, our objective is to identify and monitor the metrics while system testing is in progress so that the nimble decisions can be taken by the management team [2].

13.4.1 Metrics for Monitoring Test Execution

It is desirable to monitor the execution of system test cases of a large project involving tens of test engineers and taking several months. The system test execution for large projects is monitored weekly in the beginning and daily toward the finish. It is strongly recommended to use automated tools, such as a test factory, for monitoring and reporting the test case execution status. Different kinds of queries can be generated to obtain the *in-process* metrics after a database is in place. The following metrics are useful in successfully tracking test projects:

- **Test Case Escapes (TCE):** The test engineers may find the need to design new test cases during the testing, called test case escapes, as testing continues. The number of test case escapes is monitored as it rises. A significant increase in the number of test case escapes implies the deficiencies in the test design process, and this may adversely affect the project schedule.
- **Planned versus Actual Execution (PAE) Rate:** The actual number of test cases executed every week is compared with the planned number of test cases [3]. This metric is useful in representing the productivity of the test team in executing test cases. If the actual rate of execution falls far short of the planned rate, managers may have to take preventive measures so that the time required for system testing does not adversely affect the project schedule.
- **Execution Status of Test (EST) Cases:** It is useful to periodically monitor the number of test cases lying in different states—failed, passed, blocked, invalid, and untested—after their execution. It is also useful to further subdivide those numbers by test categories, such as basic, functionality, and robustness.

13.4.2 Test Execution Metric Examples

We give examples of test case execution metrics from a real-life test project called Bazooka. A total of 1592 test cases were designed to be executed. The projected execution of the 1592 test cases on a weekly basis for a 14-week test cycle is shown

Figure 13.3 PAE metric of Bazooka (PE: projected execution; AE: actually executed) project.

in Figure 13.3 in the form of a cumulative bar chart for the PAE metric. Twenty-five test cases are planned to be executed in the first week, 75 in the second week. A total of 100 test cases are planned to be executed by the end of first 2 weeks. The second row at the bottom of Figure 13.3, those actually executed, is updated. One should monitor the numbers of projected and actually executed test cases on a weekly basis during the execution cycles. If there is a large difference between the two, immediate action can be taken to understand its root cause and improve the execution rate before it is too late. The test execution rate was lower than the planned rate, as shown in the table, for the first 3 weeks. It took approximately 3 weeks for the test engineers to understand the whole system and come up to speed. The test project took 16 weeks, instead of the planned 14, to complete one execution cycle for all the test cases, including reexecution of all the failed ones. Initially, the test engineers did not run the test cases for the stress, load, and stability categories because some of the fixes concerning memory leak were checked in only at the end of week 13. Therefore, the test cycle was extended for 2 more weeks to ensure that the system could stay up without any memory leaks for an extended period of time.

Let us consider our previous example, the Bazooka test project, for which we had selected 1592 test cases. The EST metric that we monitor on a weekly basis for each test group concerns the number of test cases in the following states: passed, failed, invalid, blocked, and untested. As an example, the test execution status of week 4 is shown in Table 13.5.

TABLE 13.5 EST Metric in Week 4 of Bazooka Project

Test Groups	Total Number of Test Cases	Passed	Failed	Blocked	Invalid	Executed	Untested	Passed/ Executed (%)
Basic	156	64	9	0	0	73	83	87.67
Functionality	480	56	7	0	0	63	417	88.89
Robustness	230	22	1	0	0	23	207	95.65
Interoperability	149	15	2	0	0	17	132	88.24
Load and stability	43	1	0	0	0	1	42	100.00
Performance	54	0	0	0	0	0	54	0.00
Stress	36	0	0	0	0	0	36	0.00
Scalability	54	5	0	0	0	5	49	100.00
Regression	356	93	13	0	0	106	250	87.74
Documentation	34	8	1	0	0	9	25	88.89
Total	1592	264	33	0	0	297	1295	88.89

We explain how the Bazooka team controlled the progress of system testing. A revert criterion for a test cycle is a predicate on the quality level of a product in terms of percentage of executed test cases that have passed. If such a criterion holds, the test cycle is suspended. An example of a revert criteria is: *At any instant during the test cycle, if the cumulative failure rate reaches 20%, the cycle will restart all over after the reported defects are claimed to have been fixed*. We show the percentage of executed test cases that have passed in the rightmost column of Table 13.6. The pass rate was 71.11% at the end of the second week of Bazooka, which implies that the failed rate was 28.89%. Considering the example revert criteria given above, the test cycle should have been abandoned. However, as a special case, the Bazooka project management team made a decision not to abandon the test cycle for the following reasons:

- Only 45 out of 1592 test cases had been executed.
- Some of the test cases were considered to have failed due to procedural errors on the part of test engineers; these test cases were eventually considered to have passed.

A unanimous decision was made by the Bazooka team to wait for one more week and to monitor the test results on a daily basis. The pass rate went up to 89.26% by the end of the third week, as shown in Table 13.6. Since the failure rate was well below the 20% threshold for the revert criteria to be activated, the test cycle was continued until its completion.

The above example tells us that it is important to analyze the test metrics, rather than take decisions based on the raw data. It is interesting to note that the highest cumulative failure rate of 19.5% was observed in week 6, and 841 test cases—or 52% of the total number of test cases to be executed—were already executed. The system test cycle would have been abandoned with an increase of

TABLE 13.6 EST Metric in Bazooka Monitored on Weekly Basis

Week	Executed	Passed	Failed	Executed/ Total (%)	Passed/ Total (%)	Passed/ Executed (%)
1	5	5	0	0.31	0.31	100.00
2	45	32	13	2.83	2.01	71.11
3	121	108	13	7.60	6.78	89.26
4	297	264	33	18.66	16.58	88.89
5	542	451	91	34.05	28.33	83.21
6	841	677	164	52.83	42.53	80.50
7	991	835	156	62.25	52.45	84.26
8	1148	1009	139	72.11	63.38	87.89
9	1254	1096	158	78.77	68.84	87.40
10	1324	1214	110	83.17	76.26	91.69
11	1426	1342	84	89.57	84.30	94.11
12	1515	1450	65	95.16	91.08	95.71
13	1566	1519	47	98.37	95.41	97.00
14	1581	1567	14	99.31	98.43	99.11
15	1581	1570	11	99.31	98.62	99.30
16	1591	1580	11	99.94	99.25	99.31

Note: Total valid test cases 1592.

just 0.5% in the failure rate. Test cases concerning the most vulnerable areas of the system were executed on a priority basis in the beginning of the cycle. Therefore, a maximum number of 164 test cases failed within the first 6 weeks of testing. New images with fixes were released for system testing after the week 6. The failed test cases were reexecuted, and the system passed those tests, thus improving the pass rate, as shown in the rightmost column of Table 13.6. Only one test case from the stress group was still in the untested state by the end of week 16.

13.4.3 Metrics for Monitoring Defect Reports

Queries can be developed to obtain various kinds of information from the database after a defect tracking system is put in place. As the system test engineers submit defects, the defects are further analyzed. Useful data can be gathered from the analysis process to quantify the quality level of the product in the form of the following metrics:

- **Function as Designed (FAD) Count:** Often test engineers report defects which are not really true defects because of a misunderstanding of the system, called FADs. If the number of defects in the FAD state exceeds, say, 10% of the submitted defects, we infer that the test engineers have an inadequate understanding of the system. The lower the FAD count, the higher the level of system understanding of the test engineers.

- **Irreproducible Defects (IRD) Count:** One must be able to reproduce the corresponding failure after a defect is reported so that developers can understand the failure to be able to fix it. If a defect cannot be reproduced, then the developers may not be able to gain useful insight into the cause of the failure. Irreproducibility of a defect does not mean that the defect can be overlooked. It simply means that the defect is an intricate one and it is very difficult to fix it. The number of defects in the irreproducible state, or hidden-defect count, is a measure of the unreliability of the system.
- **Defects Arrival Rate (DAR) Count:** Defect reports arrive from different sources during system testing: system testing group (ST), software development group (SW), SIT group, and others with their own objectives in mind. The "others" group includes customer support, marketing, and documentation groups. Defects reported by all these groups are gathered and the percentage of defects reported by each group on a weekly basis computed. This is called the rate of defects reported by each group.
- **Defects Rejected Rate (DRR) Count:** The software development team makes an attempt to fix the reported defects by modifying the existing code and/or writing more code. A new software image is available for further system testing after these defects are fixed. The system testing group verifies that the defects have been actually fixed by rerunning the appropriate tests. At this stage the system testing team may observe that some of the supposedly fixed defects have not actually been fixed. Such defects are used to produce the DRR count. The DRR count represents the extent to which the development team has been successful in fixing defects. It also tells us about the productivity of the software development team in fixing defects. A high DRR count for a number of weeks should raise an alert because of the high possibility of the project slipping out of schedule.
- **Defects Closed Rate (DCR) Count:** The system testing group verifies the resolution by further testing after a defect is claimed to be resolved. The DCR metric represents the efficiency of verifying the claims of defect fixes.
- **Outstanding Defects (OD) Count:** A reported defect is said to be an outstanding defect if the defect continues to exist. This metric reflects the prevailing quality level of the system as system testing continues. A diminishing OD count over individual test cycles is an evidence of the increasingly higher level of quality achieved during the system testing of a product.
- **Crash Defects (CD) Count:** The defects causing a system to crash must be recognized as an important category of defects because a system crash is a serious event leading to complete unavailability of the system and possibly loss of data [4]. This metric, generally known as a stability metric, is very useful in determining whether to release the system with its current level of stability.

- **Arrival and Resolution of Defects (ARD) Count:** It is useful to compare the rate of defects found with the rate of their resolution [5]. Defect resolution involves rework cost which should be minimized by doing the first time development work in a better way. Hence, this metric is useful for the development team and helps them to estimate the time it may take to fix all the defects. Actions are taken to expedite the defect resolution process based on this information.

13.4.4 Defect Report Metric Examples

We give examples for most of the above metrics by using data from a second test project called Stinger. The weekly DAR from each group of the organization performing system-level testing in the second test cycle of execution for 16 weeks is shown in Table 13.7. The total remaining open defects from the first test cycle are given in the first row of the table as week 0. The average number of defects filed per week by ST, SW, SIT, and others are 57, 15, 10, and 18, respectively. Here, the term "others" includes members from customer support, marketing, and documentation groups. The arrival rates of the defects in the first 6 weeks are much higher than the rest of the test cycle. This is due to the fact that early in the test cycle test cases are prioritized to flush out defects in the more vulnerable portions of the system.

The DRR count for the Stinger project is shown in Table 13.8. The average rate of defect rejection was 5.24%. The rejected rate was almost 6% during the

TABLE 13.7 DAR Metric for Stinger Project

	ST		SW		SIT		Others		Total
Week	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects
0	663	69.21	172	17.95	88	9.19	35	3.65	958
1	120	79.47	28	18.54	2	1.32	1	0.66	151
2	99	55.00	44	24.44	31	17.22	6	3.33	180
3	108	53.73	54	26.87	31	15.42	8	3.98	201
4	101	65.16	16	10.32	22	14.19	16	10.32	155
5	107	57.22	21	11.23	26	13.90	33	17.65	187
6	108	56.25	16	8.33	35	18.23	33	17.19	192
7	59	57.84	0	0.00	1	0.98	42	41.18	102
8	15	31.91	9	19.15	3	6.38	20	42.55	47
9	28	59.57	0	0.00	1	2.13	18	38.30	47
10	14	50.00	4	14.29	0	0.00	10	35.71	28
11	12	48.00	1	4.00	0	0.00	12	48.00	25
12	59	53.64	5	4.55	9	8.18	37	33.64	110
13	28	54.90	5	9.80	0	0.00	18	35.29	51
14	23	51.11	0	0.00	0	0.00	22	48.89	45
15	12	31.58	21	55.26	3	7.89	2	5.26	38
16	24	38.71	23	37.10	0	0.00	15	24.19	62
Sum	917		247		164		293		1,621
Avearge	57	56.57	15	15.24	10	10.12	18	18.08	

TABLE 13.8 Weekly DRR Status for Stinger Test Project

Week	Number of Defects Verified	Number of Defects Closed	Number of Defects Rejected	Defects Rejected (%)
1	283	269	14	4.95
2	231	216	15	6.49
3	266	251	15	5.64
4	304	284	20	6.58
5	194	183	11	5.67
6	165	156	9	5.45
7	145	136	9	6.21
8	131	122	9	6.87
9	276	262	14	5.07
10	160	152	8	5.00
11	52	49	3	5.77
12	80	74	6	7.50
13	71	71	0	0.00
14	60	58	2	3.33
15	121	119	2	1.65
16	97	96	1	1.03
Sum	2636	2498	138	
Avearge	165	156	9	5.24

first 8 weeks, and for the next 4 weeks it was about 5%. For the remainder of the test cycle, the rejection rate dropped near 1.5%.

The OD counts are given in Table 13.9. Critical-, high-, medium-, and low-priority defects are denoted by P1, P2, P3, and P4, respectively. The *priority* level is a measure of how soon the defects are to be fixed. The outstanding defect counts carried forward from the first test cycle into the second cycle are given in the first row of the table as week 0 statistics. It is evident from this metric that the total number of outstanding defects gradually diminished from 958 to 81 within 16 weeks.

The number of crashes observed by different groups during the final 8 weeks of testing is shown in Table 13.10. It is evident from the table that the total number of crashes gradually diminished from 20 per week to 1 per week. One has to give these crashes a closer look to determine the root cause of each of the individual crashes. The development team must investigate these crashes individually.

Finally, we give an example of the ARD metric by considering a third test project called Bayonet. The projected number of defects submitted, resolved, and remaining open in the first four weeks of a test cycle are given in the upper half of Table 13.11. The projected numbers were derived from an earlier test project [6] that had very similar characteristics to Bayonet. At the start of the test cycle, the total number of open defects is 184. Out of these 184 defects, 50 defects are either of P1 or of P2 priority level, 63 defects are of level P3, and 71 defects are of

TABLE 13.9 Weekly OD on Priority Basis for Stinger Test Project

	P1		P1		P3		P4		Total
Week	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects
0	14	1.46	215	22.44	399	41.65	330	34.45	958
1	19	2.26	164	19.52	280	33.33	377	44.88	840
2	17	2.11	138	17.16	236	29.35	413	51.37	804
3	11	1.46	108	14.32	181	24.01	454	60.21	754
4	7	1.12	44	7.04	123	19.68	451	72.16	625
5	5	0.79	28	4.45	123	19.55	473	75.20	629
6	10	1.50	33	4.96	123	18.50	499	75.04	665
7	11	1.74	83	13.15	52	8.24	485	76.86	631
8	5	0.90	34	6.12	32	5.76	485	87.23	556
9	8	2.35	21	6.16	39	11.44	273	80.06	341
10	4	1.84	21	9.68	36	16.59	156	71.89	217
11	4	2.07	26	13.47	38	19.69	125	64.77	193
12	3	1.31	28	12.23	84	36.68	114	49.78	229
13	3	1.44	40	19.14	84	40.19	82	39.23	209
14	10	5.10	42	21.43	68	34.69	76	38.78	196
15	2	1.74	34	29.57	38	33.04	41	35.65	115
16	3	3.70	25	30.86	24	29.63	29	35.80	81

TABLE 13.10 Weekly CD Observed by Different Groups for Stinger Test Project

	S/W		ST		SIT		Others		Total
Week	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects	Percentage	Number of Defects	Percentage	Number of Crashes
9	6	30.00	8	40.00	6	30.00	0	0.00	20
10	5	31.25	6	37.50	3	18.75	2	12.50	16
11	4	25.00	9	56.25	0	0.00	3	18.75	16
12	3	33.33	4	44.44	0	0.00	2	22.22	9
13	2	22.22	6	66.67	0	0.00	1	11.11	9
14	1	14.29	2	28.57	1	14.29	3	42.86	7
15	0	0.00	2	66.67	1	33.33	0	0.00	3
16	0	0.00	1	100.00	0	0.00	0	0.00	1
Sum	21		38		11		11		81
Average	3	25.93	5	46.91	1	13.58	1	13.58	10

level P4. The actual numbers of submitted and resolved defects are shown in the lower half of the same table. The actual numbers of open defects are slightly lower than the projected numbers after the completion of the first week. However, the actual number of resolved defects is much lower than the projected number in the second week. This suggests that the software developers were slower in resolving the defects. The total number of open defects at the end of the second week is higher than the estimated number. At this stage the software development manager must take actions to improve the defect resolution rate by either increasing the working hours or moving software developers from another project to Bayonet. In

TABLE 13.11 ARD Metric for *Bayonet*

		Submitted				Resolved				Open			
Week	Build	Total	P1+P2	P3	P4	Total	P1+P2	P3	P4	Total	P1+P2	P3	P4
							Projected						
										184	50	63	71
1	build10	75	24	36	15	118	38	60	20	141	36	39	66
2	build11	75	24	36	15	118	38	60	20	98	22	15	61
3	build12	75	24	36	15	95	38	37	20	78	8	14	56
4	build13	14	5	7	2	20	10	5	5	72	3	16	53
							Actual						
1	build10	67	23	27	17	119	37	49	33	132	36	41	55
2	build11	77	26	36	15	89	37	37	15	120	25	40	55
3	build12	62	18	32	12	100	36	52	12	82	7	20	55
4	build13	27	11	12	4	33	15	8	10	76	3	24	49

this case, the manager moved an experienced software developer to this project to expedite the resolution of outstanding defects. It is advisable to have the upper portion of the table in place before the start of a test cycle. The second half of the table evolves as the test cycle progresses so that everyone in the development team becomes aware of the progress of defect resolution.

13.5 ORTHOGONAL DEFECT CLASSIFICATION

Orthogonal defect classification (ODC) [7] is a methodology for rapid capturing of the semantics of each software defect. The ODC methodology provides both a classification scheme for software defects and a set of concepts that provide guidance in the analysis of the classified aggregate defect data. Here, orthogonality refers to the nonredundant nature of the information captured by the defect attributes and their values that are used to classify defects. The classification of defects occurs at two different points in time during the life cycle of a defect. First, a defect is put in its initial new state when it is discovered, where the circumstances leading up to the exposure of the defect and the likely impact are typically known. Second, a defect is moved to the resolved state, where the exact nature of the defect and the scope of the fix are known. ODC categories capture the semantics of a defect from these two perspectives. In the new state, the submitter needs to fill out the following ODC attributes, or fields:

- **Activity:** This is the actual activity that was being performed at the time the defect was discovered. For example, the developer might decide to do a code review of a particular procedure during the system testing phase. In this case, the term would be "system testing," whereas the activity is "code review."

- **Trigger:** The environment or condition that had to exist for the defect to surface. Triggers describe the requirements to reproduce the defect.
- **Impact:** This refers to the effect the defect would have on the customer if the defect had escaped to the field.

The owner of the defect moves the defect to the resolved state when the defect has been fixed and needs to fill out the following ODC attributes, or fields:

- **Target:** The target represents the high-level identity, such as design, code, or documentation, of the entity that was fixed.
- **Defect Type:** The defect type represents the actual correction that was made.
- **Qualifier:** The qualifier specifies whether the fix was made due to missing, incorrect, or extraneous code.
- **Source:** The source indicates whether the defect was found in code developed in-house, reused from a library, ported from one platform to another, or provided by a vendor.
- **Age:** The history of the design or code that had the problem. The age specifies whether the defect was found in new, old (base), rewritten, or refixed code:
  1. **New:** The defect is in a new function which was created by and for the current project.
  2. **Base:** The defect is in a part of the product which has not been modified by the current project and is not part of a standard reuse library. The defect was not injected by the current project, and therefore it was a latent defect.
  3. **Rewritten:** The defect was introduced as a direct result of redesigning and/or rewriting an old function in an attempt to improve its design or quality.
  4. **Refixed:** The defect was introduced by the solution provided to fix a previous defect.

The ODC attributes can be collected by modifying an existing defect tracking tool that is being used for modeling defects. As the data are collected, those must be *validated* and *assessed* on a regular basis. The individually classified defects are reviewed to ensure the consistency and correctness of the classification. The data are ready for assessment after those have been validated. The assessment is not done against each individual defect; rather trends and patterns in the aggregate data are studied. Data assessment of ODC classified data is based on the relationships of the ODC attributes to one another and to non-ODC attributes, such as category, severity, and defect submit date [8]. For example, the relationships among the attributes of defect type, qualifier, category, submit date, and severity of defects need to be considered to evaluate the product stability.

The ODC assessment must be performed by an analyst who is familiar with the project and has an interest in data analysis. A good user-friendly tool for

visualizing data is needed. The ODC analyst must provide regular feedback, say, on a weekly basis, to the software development team so that appropriate actions can be taken. Once the feedback is given to the software development team, they can then identify and prioritize actions to be implemented to prevent defects from recurring.

The ODC along with application of the Pareto analysis technique [9, 10] gives a good indication of the parts of the system that are error prone and, therefore, require more testing. Juran [9] stated the Pareto principle very simply as "concentrate on the vital few and not the trivial many." An alternative expression of the principle is to state that 80% of the problems can be fixed with 20% of the effort, which is generally called the 80–20 rule. This principle guides us in efficiently utilizing the effort and resources. As an example, suppose that we have data on test *category* and the frequency of occurrence of defects for a hypothetical Chainsaw system test project as shown in Table 13.12. The data plotted on a Pareto diagram, shown in Figure 13.4, which is a bar graph showing the frequency of occurrence of defects with the most frequent ones appearing first. Note that the *functionality* and the *basic* groups have high concentration of defects. This information helps system test engineers to focus on the functionality and the basic category parts of the Chainsaw test project. The general guideline for applying Pareto analysis is as follows:

- Collect ODC and non-ODC data relevant to problem areas.
- Develop a Pareto diagram.
- Use the diagram to identify the vital few as issues that should be dealt with on an individual basis.
- Use the diagram to identify the trivial many as issues that should be dealt with as classes.

TABLE 13.12 Sample Test Data of Chainsaw Test Project

Category	Number of Defect Occurrences
1. Basic	25
2. Functionality	48
3. Robustness	16
4. Interoperability	4
5. Load and stability	6
6. Performance	12
7. Stress	6
8. Scalability	7
9. Regression	3
10. Documentation	2

Figure 13.4 Pareto diagram for defect distribution shown in Table 13.12.

Remark. Vilfredo Federico Damaso Pareto was a French–Italian sociologist, economist, and philosopher. He made several important contributions, especially in the study of income distribution and in the analysis of individuals' choices. In 1906 he made the famous observation that 20% of the population owned 80% of the property in Italy, later generalized by Joseph M. Juran and others into the so-called Pareto principle (also termed the 80–20 rule) and generalized further to the concept of a Pareto distribution.

13.6 DEFECT CAUSAL ANALYSIS

The idea of defect causal analysis (DCA) in software development is effectively used to raise the quality level of products at a lower cost. Causal analysis can be traced back to the quality control literature [11] as one of the quality circle activities in the manufacturing sector. The quality circle concept is discussed in Section 1.1. The causes of manufacturing defects are analyzed by using the idea of a quality circle, which uses *cause–effect* diagrams and Pareto diagrams. A cause–effect diagram is also called an *Ishikawa* diagram or a *fishbone* diagram. Philip Crosby [12] described a case study of an organization called "HPA Appliance" involving the use of causal analysis to prevent defects on manufacturing lines. Causal analysis of software defects is practiced in a number of Japanese companies, usually in the context of quality circle activities [13, 14].

The idea of DCA was developed at IBM [15]. Defects are analyzed to (i) determine the cause of an error, (ii) take actions to prevent similar errors from

occurring in the future, and (iii) remove similar defects that may exist in the system or detect them at the earliest possible point in the software development process. Therefore, DCA is sometimes referred to as defect prevention or defect RCA [16]. The importance of DCA has been succinctly expressed by Watts S. Humphrey [17] as follows: "While detecting and fixing defects is critically important, it is an inherently defensive strategy. To make significant quality improvements, you should identify the causes of these defects and take steps to eliminate them."

In his tutorial article [18], David N. Card explains a five-step process for conducting DCA. Card suggests three key principles to drive DCA:

- **Reduce the number of defects to improve quality:** One can define software quality in terms of *quality factors* and *attributes*, but it is needless to say that a product with many defects lacks quality. Software quality is improved by focusing on defect prevention and early detection of defects.
- **Apply local expertise where defects occur:** The responsibility of DCA should be given to the software developers who unwittingly contributed to the mistakes. They are best qualified to identify what went wrong and how to prevent it. DCA should take place at the actual location where the failure occurred, instead of in a remote conference room. In other words, DCA should be performed at the source.
- **Focus on systematic errors:** Intuitively, an error is said to be a systematic error if the same error or similar errors occur on many occasions. Systematic errors account for a significant portion of the defects found in software projects. Identifying and preventing systematic errors do have a huge impact on the quality of the product.

As the term suggests, DCA focuses on understanding the cause–effect relationship [19]. Three conditions are established to demonstrate a causal relationship between a cause and a symptom in the form of a failure:

- There must be a correlation between the hypothesized cause and the effect.
- The hypothesized cause must precede the effect in time.
- The mechanism linking the cause to the effect must be identified.

The first condition indicates that the effect is likely to be observed when a cause occurs. The second condition is obvious. The third condition requires investigation, which consists of five steps to select systematic errors associated with common failures and tracing them back to their original cause:

(i) *When was the failure detected and the corresponding fault injected?* Note the development phase, that is, code review, unit testing, integration testing, or system testing, where the failure is observed. Next, determine the phase of development in which the corresponding fault was injected. Developers identify the more problematic areas by considering the above information.

(ii) *What scheme is used to classify errors?* We identify the classes of important errors in this step. Grouping errors together helps us in identifying

clusters in which common problems are likely to be found. One can use Pareto diagrams to identify problem clusters. Unlike the ODC scheme, DCA does not use a predefined set of error categories. The objective of DCA is not to analyze percentages of errors in each categories, but rather to analyze each error to understand the reason the error occurred and take preventive measures. Categorization of errors is based on the nature of the work performed, the opportunities for making mistakes, and the current dynamics of the project. Suppose that during our analysis we notice that there are many errors related to the program interface. Then we have two levels of error classification: (i) a coarse-grain classification called interface errors and (ii) many fine-grained subcategories, such as construction, misuse of interface, and initialization, as discussed in Section 7.2.

(iii) *What is the common problem (systematic error)*? The set of problem reports comprising a cluster of errors, identified in the second step above, gives an indication of a systematic error. In general, systematic errors are associated with the specific functionality of a product. Ignore the effects of individual errors in seeking out systematic errors during the identification process.

(iv) *What is the principal cause of the errors*? We endeavor to trace the error back to its cause and understand the root cause associated with many errors in the cluster. Finding the causes of errors requires (i) thorough understanding of the process and the product and (ii) experience and judgment. In order to achieve this goal, we may need to draw a cause–effect diagram as shown in Figure 13.5. First, the problem being analyzed must be stated and then the main branch of the diagram connected to the problem statement is drawn. Second, the generic causes to be used in the diagram are added. The generic causes usually fall into four categories as Ishikawa has identified [11]:

- **Tools:** The tools used in developing the system may be clumsy, unreliable, or defective.
- **Input:** It may be incomplete, ambiguous, or defective.
- **Methods:** The methods may be incomplete, ambiguous, wrong, or unenforced.
- **People:** They may lack adequate training or understanding.

We need to brainstorm to collect specific causes and attach the specific causes to the appropriate generic causes. The significance of the causes identified is discussed after all the ideas have been explored, focusing on the principal causes contributing to the systematic errors. Often this is found in the denser cluster of causes in the diagram. The cause–effect diagram provides a focused way to address the problem.

(v) *How could the error be prevented in the future?* Actions, which are preventive, corrective, or mitigating, are taken after the principal cause of a systematic error has been identified:

Figure 13.5 Cause–effect diagram for DCA.

- **Preventive:** A preventive action reduces the chances of similar problems occurring in the future. The idea is to detect a problem earlier and fix it.
- **Corrective:** A corrective action means fixing the problems. Attacking the cause itself may not be cost-effecive in all situations.
- **Mitigating:** A mitigating action tries to counter the adverse consequences of problems. Mitigating actions are a part of a risk management activity [20].

A combination of preventive, corrective, and mitigating actions may be applied, depending on the cost of taking actions and the magnitude of the symptoms observed. A comprehensive action plan is created to promote either prevention or detection of systematic errors at the earliest possible time in the software development process. The following are typical questions to ask at this point to elicit recommendations:

- What is the earliest a defect could have been recognized?
- What symptoms suggest that the defect may occur again?
- What did we need to know to avoid the defect?
- How could we have done things differently?

The recommended actions need to be specific and precise for others to be able to follow. Some common examples of preventive and corrective actions are as follows:

- **Training:** This allows people to improve their knowledge about the product. Training may include concrete actions such as organizing a seminar series on the components of the product, preparing a technical write-up on a complex aspect of the product, and writing an article about repetitive errors.

- **Improvement in Communication:** These are actions that improve communication procedures within the product organizations. One can think of notifying programmers and testers via email when the interface specification changes. Holding weekly team meetings helps in clarifying and disseminating useful information.
- **Using Tools:** Actions are taken to develop new tools or enhance existing tools that support the processes. One can develop or buy a tool to check memory utilization or add a new coverage metric to the module coverage tool.
- **Process Improvements:** These are actions that improve existing processes and define new processes. Such actions might modify the design change process or add new items to a common error list.

13.7 BETA TESTING

Beta testing is conducted by the potential buyers prior to the official release of the product. The purpose of beta testing is not to find defects but to obtain feedback from the field about the usability of the product. There are three kinds of beta tests performed based on the relationships between the potential buyers and the sellers:

- **Marketing Beta:** The purpose here is to build early awareness and interest in the product among potential buyers.
- **Technical Beta:** The goal here is to obtain feedback about the usability of the product in a real environment with different configurations from a small number of friendly customers. The idea is to obtain feedback from a limited number of customers who commit a considerable amount of time and thought to their evaluation.
- **Acceptance Beta:** The idea of this test is to ensure that the product meets its acceptance criteria. It is the fulfillment of the contract between a buyer and a seller. The system is released to a specific buyer, who has a contractual original equipment manufacturer (OEM) agreement with the supplier. Acceptance beta includes the objective of technical beta as well. Acceptance testing is discussed in Chapter 14.

The defects and the progress of test execution provide useful information about the quality of the system. The quality of the system under development is a moving target during test execution. A decision about when to release the system to the beta customers is made by software project team members. The consequences of early and delayed releases are as follows:

- If it is released too early, too many defects in the system may have a negative impact on the prospective customers, and it may generate bad publicity for both the product and the company.
- If the release is delayed, other competitors may capture the market.

The beta release criteria are established by the project team, and the criteria are less stringent than the final system test cycle criteria given in Chapter 13. A framework for writing beta release criteria is given in Table 13.13. All the beta release criteria in Table 13.13 are self-explanatory, except the last item, namely, that all the identified beta blocker defects are in the closed state. The beta blocker defects are the set of defects that must be in the closed state before the software is considered for beta testing. Three weeks before the beta release, the beta blocker defects are identified at the cross-functional project meeting. These defects are tracked on a daily basis to ensure that the defects are resolved and closed before the beta release. In the meantime, as new defects are submitted, these defects are evaluated to determine whether or not these are beta blockers. If it is concluded that a new defect is a beta blocker, then the defect is included on the list of beta blockers and all the defects are tracked for closure.

The system test cycle continues concurrently with beta testing. Weekly meetings are conducted with beta customers by the beta support team, which consists of cross-functional team members of the project. The weekly test status reports from the beta customers are analyzed, and actions are taken to resolve the issues by either fixing the defects or providing them a workaround of the issues raised by the beta customers. If a beta customer decides to deploy the system, an agreement can be reached to provide patch releases by fixing any outstanding deficiency in the system by a certain time frame. A final test cycle with a scaled-down number of test cases is executed as a regression test suite after an agreement is reached, and the product is prepared for first customer shipment (FCS).

TABLE 13.13 Framework for Beta Release Criteria

1. Ninety-eight percent all the test cases have passed.
2. The system has not crashed in the last week of all testing.
3. All the *resolved* defects must be in the closed state.
4. A release note with all the defects that are still in the open state along with workaround must be available. If the workaround does not exist, then an explanation of the impact on the customer must be incorporated in the release note.
5. No defect with "critical" severity is in the open state.
6. No defect with "high" severity has a probability of hitting at a customer site of more than 0.1.
7. The product does not have more than a certain *number* of defects with severity "medium." The number may be determined by the software project team members.
8. All the test cases for performance testing must have been executed, and the results must be available to the beta test customer.
9. A beta test plan must be available from all potential beta customers. A beta test plan is nothing but a user acceptance test plan.
10. A draft of the user guide must be available.
11. The training materials on the system are available for field engineers.
12. The beta support team must be available for weekly meetings with each beta customer.
13. All the identified beta blocker defects are in the closed state.

13.8 FIRST CUSTOMER SHIPMENT

The exit criterion of the final test cycle must be satisfied before the FCS which is established in the test execution strategy section of Chapter 13. An FCS readiness review meeting is called to ensure that the product meets the shipment criteria. The shipment criteria are more than just the exit criteria of the final test cycle. This review should include representatives from the key function groups responsible for delivery and support of the product, such as engineering, operation, quality, customer support, and product marketing. A set of generic FCS readiness criteria is as follows:

- All the test cases from the test suite should have been executed. If any of the test cases is unexecuted, then an explanation for not executing the test case should have been provided in the test factory database.
- Test case results are updated in the test factory database with passed, failed, blocked, or invalid status. Usually, this is done during the system test cycles.
- The requirement database is updated by moving each requirement from the verification state to either the closed or the decline state, as discussed in Chapter 11, so that compliance statistics can be generated from the database. All the issues related to the EC must be resolved with the development group.
- The pass rate of test cases is very high, say, 98%.
- No crash in the past two weeks of testing has been observed.
- No known defect with *critical* or *high* severity exists in the product.
- Not more than a certain number of known defects with *medium* and *low* levels of severity exist in the product. The threshold number may be determined by the software project team members.
- All the identified FCS blocker defects are in the closed state.
- All the resolved defects must be in the closed state.
- All the outstanding defects that are still in either the open or assigned state are included in the release note along with the workaround, if there is one.
- The user guides are in place.
- A troubleshooting guide is available.
- The test report is completed and approved. Details of a test report are explained in the following section.

Once again, three weeks before the FCS, the FCS blocker defects are identified at the cross-functional project meeting. These defects are tracked on a daily basis to ensure that the defects are resolved and closed before the FCS. In the meantime, as new defects are submitted, these are evaluated to determine whether these are FCS blockers. If it is concluded that a defect is a blocker, then the defect is included in the blocker list and tracked along with the other defects in the list for its closure.

13.9 SYSTEM TEST REPORT

During the test execution cycles test reports are generated and distributed as discussed in Sections 13.4.2 and 13.4.4. A final summary report is created after completion of all the test cycles. The structure of the final report is outlined in Table 13.14.

The introduction to the test project section of the test report summarizes the purpose of the report according to the test plan. This section includes the following information:

- Name of the project
- Name of the software image
- Revision history of this document
- Terminology and definitions
- Testing staff
- Scope of the document
- References

The summary of test results section summarizes the test results for each test category identified in the test plan. In each test cycle, the test results are summarized in the form of the numbers of test cases passed, failed, invalid, blocked, and untested and the reasons for not executing some test cases. A test case may not be tested because of several reasons, such as unavailability of equipment and difficulty in creating the test scenario using the available test bed setup. The latter may only be possible at a real beta testing site. For each test cycle, the following data are given in this section:

- Start and completion dates
- Number of defects filed
- Number of defects in different states, such as irreproducible, FAD, closed, shelved, duplicate, postponed, assigned, and open

TABLE 13.14 Structure of Final System Test Report

1.	Introduction to the test project
2.	Summary of test results
3.	Performance characteristics
4.	Scaling limitations
5.	Stability observations
6.	Interoperability of the system
7.	Hardware/software compatible matrix
8.	Compliance requirement status

Problems still remaining in the system, including any shortcoming or likelihood of failure, are summarized in this section in terms of their levels of severity and their potential impact on the customers. For example, if the password is case insensitive, then it should be stated so. As another example, if an operating system puts a limit of 20 on the number of windows that can be simultaneously opened, then it should be stated that the user should not open more than 20 windows simultaneously; exceeding the limit may cause the system to crash.

In the performance characteristics section of the document, the system response time, throughput, resource utilization, and delay measurement results are described along with the test environments and tools used in the measurement process.

The limits of the system are stated in the scaling limitation section of the document based on the findings of scalability testing. Once again the facilities and tools that are used in scaling testing are described here.

The stability observations section states the following information:

- Uptime in number of hours of the system
- Number of crashes observed by different groups, such as software developer, ST, and SIT, in each week of the test cycles
- Descriptions of symptoms of the defects causing system crash that are in the irreproducible state

We state in the interoperability section of the document those third-party software and hardware systems against which the system interoperability tests were conducted. The list may include software and hardware against which the system will not interoperate.

A table shows what software versions are compatible with what hardware revisions in the hardware/software compatible matrix section. Generation of this table is based on the different kinds of hardware systems and different versions of the software used during system test cycles.

Finally, in the compliance requirement status section of the document compliance statistics are summarized by generating them from the requirements database. The following reports are generated by creating appropriate queries on the database:

- Requirements that are in compliance, partial compliance, or noncompliance with the software image
- Requirements that are verified by testing, analysis, demonstration, and inspection
- Requirements that are covered by each test case of the test suite along with the status of the test case

13.10 PRODUCT SUSTAINING

Once a product is shipped to one of the paying customers and if there is no major issue reported by the customer within a time frame of, say, three weeks,

then the product's general availability (GA) is declared. After that the software project is moved to a new phase called the sustaining phase. The goal of this phase is to maintain the software quality throughout the product's market life. Software maintenance activities occur because software testing cannot uncover all the defects in a large software system. The following three software maintenance activities were coined by Swanson [21]:

- **Corrective:** The process that includes isolation and correction of one or more defects in the software.
- **Adaptive:** The process that modifies the software to properly interface with a changing environment such as a new version of hardware or third-party software.
- **Perfective:** The process that improves the software by the addition of new functionalities, enhancements, and/or modifications to the existing functions.

The first major task is to determine the type of maintenance task to be conducted when a defect report comes in from a user through the customer support group. The sustaining team, which includes developers and testers, is assigned immediately to work on the defect if the defect reported by the customer is considered as *corrective* in nature. The status of the progress is updated to the customer within 24 hours. The group continues to work until a patch with the fix is released to the customer. If the defect reported is considered as either *adaptive* or *perfective* in nature, then it is entered in the requirement database, and it goes though the usual software development phases.

The sustaining test engineers are different from the system test engineers. Therefore, before a product transition occurs, the sustaining engineers are provided with enough training about the product to carry out sustaining test activities. In reality, the test engineers are rotated back and forth between the system test group and the sustaining group. The major tasks of sustaining test engineers are as follows:

- Interact with customers to understand their real environment
- Reproduce the issues observed by the customer in the laboratory environment
- Once the root cause of the problem is known, develop new test cases to verify it
- Develop upgrade/downgrade test cases from the old image to the new patch image
- Participate in the code review process
- Select a subset of regression tests from the existing pool of test cases to ensure that there is no side effect because of the fix
- Execute the selected test cases
- Review the release notes for correctness
- Conduct experiments to evaluate test effectiveness

13.11 MEASURING TEST EFFECTIVENESS

It is useful to evaluate the effectiveness of the testing effort in the development of a product. After a product is deployed in the customer environment, a common measure of test effectives is the number of defects found by the customers that were not found by the test engineers prior to the release of the product. These defects had escaped from our testing effort. A metric commonly used in the industry to measure test effectiveness is the defect removal efficiency (DRE) [22], defined as

$$\text{DRE} = \frac{\text{number of defects found in testing}}{\text{number of defects found in testing} + \text{number of defects not found}}$$

We obtain the number of defects found in testing from the defect tracking system. However, calculating the number of defects not found during testing is a difficult task. One way to approximate this number is to count the defects found by the customers within the first six months of its operation. There are several issues that must be understood to interpret the DRE measure in an useful way:

- Due to the inherent limitations of real test environments in a laboratory, certain defects are very difficult to find during system testing no matter how thorough we are in our approach. One should include these defects in the calculation if the goal is to measure the effectiveness of the testing effort including the limitations of the test environments. Otherwise, these kinds of defects may be eliminated from the calculation.
- The defects submitted by customers that need *corrective* maintenance are taken into consideration in this measurement. The defects submitted that need either *adaptive* or *perfective* maintenance are not real issues in the software; rather they are requests for new feature enhancements. Therefore, defects that need adaptive and perfective maintenance may be removed from the calculation.
- There must be a clear understanding of the duration for which the defects are counted, such as starting from unit testing or system integration testing to the end of system testing. One must be consistent for all test projects.
- This measurement is not for one project; instead it should be a part of the long-term trend in the test effectiveness of the organization.

Much work has been done on the *fault seeding* approach [23] to estimate the number of escaped defects. In this approach the effectiveness of testing is determined by estimating the number of actual defects using an extrapolation technique. The approach is to inject a small number of representative defects into the system and measure the percentage of defects that are uncovered by the sustaining test engineers. Since the sustaining test team remains unaware of the seeding, the extent to which the product reveals the known defects allows us to extrapolate the extent to which it found unknown defects. Suppose that the product contains N defects and K defects are seeded. At the end of the test experiments, the sustaining test team has found n unseeded and k seeded defects. The fault seeding theory

asserts the following:

$$\frac{k}{K} = \frac{n}{N} \Longrightarrow N = n\left(\frac{K}{k}\right)$$

For example, consider a situation where 25 known defects have been deliberately seeded in a system. Let us assume that the sustaining testers detect 20 (80%) of these seeded defects and uncovers 400 additional defects. Then, the total number of estimated defects in the system is 500. Therefore, the product still has $500 - 400 = 100$ defects waiting to be discovered plus 5 seed defects that still exist in the code. Using the results from the seeding experiment, it would be possible to obtain the total number of escaped defects from the system testing phase. This estimation is based on the assumption that the ratio of the number of defects found to the total number of defects is 0.80—the same ratio as for the seeded defects. In other words, we assume that the 400 defects constitute 80% of all the defects found by the sustaining test engineers in the system. The estimated remaining number of defects are still hidden in the system. However, the accuracy of the measure is dependent on the way the defects are injected. Usually, artificially injected defects are manually planted, but it has been proved difficult to implement defect seeding in practice. It is not easy to introduce artificial defects that can have the same impact as the actual defects in terms of difficulty of detection. Generally, artificial defects are much easier to find than actual defects.

Spoilage Metric Defects are injected and removed at different phases of a software development cycle [24]. Defects get introduced during requirements analysis, high-level design, detailed design, and coding phases, whereas these defects are removed during unit testing, integration testing, system testing, and acceptance testing phases. The cost of each defect injected in phase X and removed in phase Y is not uniformly distributed; instead the cost increases with the increase in the distance between X and Y. The delay in finding the dormant defects cause greater harms, and it costs more to fix because the dormant defects may trigger the injection of other related defects, which need to be fixed in addition to the original dormant defects. Therefore, an effective testing method would find defects earlier than a less effective testing method would. Hence an useful measure of test effectiveness is defect age, known as *PhAge*. As an example, let us consider Table 13.15, which shows a scale for measuring age. In this example, a requirement defect discovered during high-level design review would be assigned a PhAge of 1, whereas a requirement defect discovered at the acceptance testing phase would be assigned a PhAge of 7. One can modify the table to accommodate different phases of the software development life cycle followed within an organization, including the PhAge numbers.

If the information about a defect introduction phase can be determined, it can be used to create a matrix with rows corresponding to the defect injected in each phase and columns corresponding to defects discovered in each phase This is often called a *defect dynamic model*. Table 13.16 shows a defect injected–v discovered matrix from an imaginary test project called Boomerang. In this example, there were

TABLE 13.15 Scale for Defect Age

Phase Injected	Phase Discovered: Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing	Acceptance Testing
Requiremants	0	1	2	3	4	5	6	7
High-level design		0	1	2	3	4	5	6
Detailed design			0	1	2	3	4	5
Coding				0	1	2	3	4

TABLE 13.16 Defect Injection versus Discovery on Project Boomerang

Phase Injected	Phase Discovered: Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing	Acceptance Testing	Total Defects
Requiremants	0	7	3	1	0	0	2	4	17
High-level design		0	8	4	1	2	6	1	22
Detailed design			0	13	3	4	5	0	25
Coding				0	63	24	37	12	136
Summary	0	7	11	18	67	30	50	17	200

seven requirement defects found in high-level design, three in detailed design, one in coding, two in system testing, and four in acceptance testing phases.

Now a new metric called *spoilage* [25, 26] is defined to measure the defect removal activities by using the defect age and defect dynamic model. The spoilage metric is calculated as

$$\text{Spoilage} = \frac{\sum(\text{number of defects} \times \text{discovered PhAge})}{\text{total number of defects}}$$

Table 13.17 shows the spoilage values for the Boomerang test project based on the number of defects found (Table 13.15) weighted by defect age (Table 13.16). During acceptance testing, for example, 17 defects were discovered, out of which 4 were attributed to defects injected during the requirements phase of the Boomerang project. Since the defects that were found during acceptance testing could have been found in any of the seven previous phases, the requirement defects that were dormant until the acceptance testing were given a PhAge of 7. The weighted number of requirement defects revealed during acceptance testing is 28, that is, $7 \times 4 = 28$. The spoilage values for requirements, high-level design, detail design, and coding phases are 3.2, 2.8, 2.0, and 1.98, respectively. The spoilage value for the Boomerang test project is 2.2. A spoilage value close to 1 is an indication of

TABLE 13.17 Number of Defects Weighted by Defect Age on Project Boomerang

	Phase Discovered										
Phase Injected	Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing	Acceptance Testing	Weight	Total Defects	Spoilage as Weight/Total Defects
Requiremants	0	7	6	3	0	0	12	28	56	17	3.294117647
High-level design		0	8	8	3	8	30	6	63	22	2.863636364
Detailed design			0	13	6	12	20	0	51	25	2.04
Coding				0	63	48	111	48	270	136	1.985294118
Summary	0	7	14	24	72	68	173	82	440	200	2.2

a more effective defect discovery process. As an absolute value, the spoilage metric has little meaning. This metric is useful in measuring the long-term trend of test effectiveness in an organization.

13.12 SUMMARY

This chapter began with a state transition model which allows us to represent each phase in the life cycle of a defect by a state. At each state of the transition model, certain actions are taken by the owner; the defect is moved to a new state after the actions are completed. We presented a defect schema that can be used to monitor various defect metrics. Two key concepts involved in modeling defects are priority and severity. On the one hand, a priority level is a measure of how soon a defect needs to be fixed, that is, urgency. On the other hand, a severity level is a measure of the extent of the detrimental effect of the defect on the operation of the product.

Then we explored the idea of test preparedness to start system-level testing. We discussed the idea of a test execution working document that needs to be maintained before the start of the system test execution in order to ensure that system test engineers are ready to start the execution of system tests. We recommend two types of metrics to be monitored during the execution of system-level test cases:

- Metrics for monitoring test execution
- Metrics for monitoring defects

The first kind of metric concerns the process of executing test cases, whereas the second kind concerns the defects found as a result of test execution. We provided examples of test case execution and defect report metrics from different real-life test projects.

Next, we examined three kinds of defect analysis techniques: ODC, causal, and the Pareto methodology. Causal analysis is conducted to identify the root cause of the defects and to take actions to eliminate the sources of defects; this is done at the time of fixing defects. In the ODC method, assessment is not done against individual defects; rather, trends and patterns in the aggregate data are studied. The Pareto principle is based on the postulate that 80% of the problems can be fixed with 20% of the effort. We showed that ODC along with the application of Pareto analysis can give a good indication of which parts of the system are error prone and may require more testing.

We provided frameworks for beta releases and discussed the process of beta testing as is conducted at the customer's site. We classified three types of beta testing: marketing beta, technical beta, and acceptance beta. The purpose of marketing beta is to build early awareness and interest in the product among potential buyers. A technical beta test is performed to obtain feedback about the usability of the product in a real environment with different configurations. The idea is to obtain feedback from a limited number of users who commit a considerable amount of time and thought to their evaluation. An acceptance beta test is performed to

ensure that the product is acceptable, that is, the product meets its acceptance criteria. Acceptance testing is the fulfillment of the contract between a buyer and a seller.

Next, we provided a detailed structure of the system test report, which must be generated before the product's general availability is declared. Once the product is declared as generally available, the software project is moved to a new phase called the sustaining phase for maintenance. During this phase sustaining test engineers are responsible for carrying out sustaining test activities. We explained the tasks of sustaining test engineers in this chapter.

Finally, we provided two metrics used to measure test effectiveness: defect removal efficiency and spoilage. The objective of a test effectiveness metric is to evaluate the effectiveness of the system testing effort in the development of a product, that is, the number of defects that had escaped from our testing effort. We presented the fault seeding approach in order to estimate the number of escaped defects.

LITERATURE REVIEW

In reference [4] a set of in-process metrics for testing phases of the software development process are described. For each metric the authors describe its purpose, data, interpretation, and use and present graphical examples with real-life data. These metrics are applicable to most software projects and are an integral part of system-level testing.

In reference [8] three case studies are presented to measure test effectiveness using ODC. All three case studies highlight how technical teams can use ODC data for objective feedback on their development processes and the evolution of their product. The case studies include background information on the products, how their ODC process was implemented, and the details of the assessments, including actions that resulted.

Several excellent examples of defect RCA efforts in software engineering have been published. Interested readers are recommended to go through the following articles for more information:

M. Leszak, D. E. Perry, and D. Stoll, "Classification and Evaluation of Defects in a Retrospective," *Journal of Systems and Software*, Vol. 61, April 2002, pp. 173–187.

W. D. Yu, "A Software Prevention Approach in Coding and Root Cause Analysis," *Bell Labs Technical Journal*, Vol. 3, April 1998, pp. 3–21.

The article by Leszak et al. describes a retrospective process with defect RCA, process metric analysis, and software complexity analysis of a network element as a part of an optical transmission network at Lucent Technologies (now Alcatel-Lucent). The article by Yu discusses the guidelines developed by the team to conduct defect RCA at various Lucent switching development organizations.

Mills's seminal article on *fault seeding* is reprinted in *Software Productivity* (H. D. Mills, Little, Brown and Company, Boston, 1983). The book consists of articles on the subject of software process written by the author over many years.

REFERENCES

1. D. Lemont. *CEO Discussion—From Start-up to Market Leader—Breakthrough Milestones*. Ernst and Young Milestones, Boston, May 2004, pp. 9–11.
2. G. Stark, R. C. Durst, and C. W. Vowell. Using Metrics in Management Decision Making. *IEEE Computer*, September 1994, pp. 42–48.
3. S. H. Kan. *Metrics and Models in Software Quality Engineering*, 2nd ed. Addison-Wesley, Reading, MA, 2002.
4. S. H. Kan, J. Parrish, and D. Manlove. In-Process Metrics for Software Testing. *IBM Systems Journal*, January 2001, pp. 220–241.
5. R. Black. *Managing Testing Process*, 2nd ed. Wiley, New York, 2002.
6. E. F. Weller. Using Metrics to Manage Software Projects. *IEEE Computer*, September 1994, pp. 27–33.
7. R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Y. Wong. Orthogonal Defect Classification—A Concept for In-Process Measurement. *IEEE Transactions on Software Engineering*, November 1992, pp. 943–956.
8. M. Butcher, H. Munro, and T. Kratschmer. Improving Software Testing via ODC: Three Case Studies. *IBM Systems Journal*, January 2002, pp. 31–44.
9. J. Juran, M. Gryna, M. Frank, Jr., and R. Bingham, Jr. *Quality Control Handbook*, 3rd ed. McGraw-Hill, New York, 1979.
10. T. McCabe and G. Schulmeyer. The Pareto Principle Applied to Software Quality Assurance. In *Handbook of Software Quality Assurance*, 2nd ed., G. Schulmeyer and J. McManus, Eds. Van Nostrand Reinhold, New York, 1992.
11. K. Ishikawa. *What Is Total Quality Control*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
12. P. Crosby. *Quality Is Free*. New American Library, New York, 1979.
13. K. Hino. Analysis and Prevention of Software Errors as a Quality Circle Activity. *Engineers* (Japanese), January 1985, pp. 6–10.
14. H. Sugaya. Analysis of the Causes of Software Bugs. *Nikkei Computer* (Japanese), August 1985, pp. 167–176.
15. R. G. Mays, C. L. Jones, G. J. Holloway, and D. P. Studinski. Experiences with Defect Prevention. *IBM System Journal*, Vol. **29**, No. 1, 1990, pp. 4–32.
16. M. Pezzé and M. Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, Hoboken, NJ, 2007.
17. W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, Reading, MA, 1995.
18. D. N. Card. Learning from Our Mistakes with Defect Causal Analysis. *IEEE Software*, January-February 1998, pp. 56–63.
19. D. N. Card. Understanding Causal Systems. *Crosstalk, the Journal of Defense Software Engineering*, October 2004, pp. 15–18.
20. N. Crockford. *An Introduction to Risk Management*, 2nd ed. Woodhead-Faulkner, Cambridge, U.K. 1986.
21. E. B. Swanson. The Dimensions of Maintenance. In *Proceeding of the Second International Conference on Software Engineering*, October 1976, pp. 492–497.
22. C. Jones. *Applied Software Measurement*. McGraw-Hill, New York, 1991.
23. H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Survays*, December 1997, pp. 366–427.
24. A. A. Frost and M. J. Campo. Advancing Defect Containment to Quantitative Defect Management. *Crosstalk, the Journal of Defense Software Engineering*, December 2007, pp. 24–28.
25. R. D. Craig and S. P. Jaskiel. *Systematic Software Testing*. Artech House, Norwood, MA, 2002.

26. R. B. Grady and D. L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Prentice-Hall, Upper Saddle River, NJ, 1998.

Exercises

1. Why is it important to assign both severity and priority levels to a defect?
2. Implement the schema and the defect model discussed in this chapter using a commercially available defect tracking tool (viz., ClearQuest of IBM).
3. Why do in-process metrics need to be tracked on daily or weekly basis during system testing?
4. What is the difference between causal analysis, statistical analysis, and Pareto analysis? The ODC method belongs to which category?
5. Modify the schema shown in Table 13.2, including values of the fields and the state transition diagram in Figure 13.1, to model ODC.
6. The projected number of defects submitted, resolved, and remaining open in the first four weeks of a test project are given in the upper half of Table 13.18. The actual numbers of submitted and resolved defects are shown in the lower half of the table. Calculate the actual number of open defects.
7. For your current system testing project, select a set of in-process metrics discussed in this chapter and track it during the system test execution phase.
8. Develop a set of beta release criteria for your current test project.
9. Write a system test report after the completion of your current test project.
10. In a test project, the number of defects found in different phases of testing are as follows: unit testing, 163 defects; integration testing, 186 defects; system testing, 271 defects. The number of defects found by sustaining test engineers by conducting fault seeding experimentation is 57. What is the value of DRE? Calculate the value of the system test DRE?

TABLE 13.18 ARD Metric for Test Project

		Submitted				Resolved				Open			
Week	Build	Total	P1+P2	P3	P4	Total	P1+P2	P3	P4	Total	P1+P2	P3	P4
							Projected						
										184	50	63	71
1	build10	75	24	36	15	118	38	60	20	141	36	39	66
2	build11	75	24	36	15	118	38	60	20	98	22	15	61
3	build12	75	24	36	15	95	38	37	20	78	8	14	56
4	build13	14	5	7	2	20	10	5	5	72	3	16	53
							Actual						
1	build10	60	26	16	18	105	35	40	30				
2	build11	77	28	34	15	89	37	37	15				
3	build12	62	20	32	10	78	24	42	12				
4	build13	24	15	15	10	72	20	25	27				

TABLE 13.19 Scale for PhAge

	Phase Discovered							
Phase Injected	Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing	Acceptance Testing
Requiremants	0	1	2	4	6	8	10	14
High-level design		0	1	2	4	6	8	12
Detailed design			0	1	2	4	6	10
Coding				0	1	2	4	8

11. Modify the schema shown in Table 13.2, including values of the fields to capture in the defect dynamic model.
12. Use the defect age (PhAge) given in Table 13.19 to recalculate the spoilage metric for the Boomerang test project.

CHAPTER 14

Acceptance Testing

Acceptance of others, their looks, their behaviors, their beliefs, bring you an inner peace and tranquility—instead of anger and resentment.
— *Anonymous*

14.1 TYPES OF ACCEPTANCE TESTING

A product is ready to be delivered to the customer after the system test group is satisfied with the product by performing system-level tests. Customers execute acceptance tests based on their expectations from the product. The services offered by a software product may be used by millions of users. For example, the service provider of a cellular phone network is a customer of the software systems running the phone network, whereas the general public forms the user base by subscribing to the phone services. It is not uncommon for someone to have a dual role as a customer and a user. The service provider needs to ensure that the product meets certain criteria before the provider makes the services available to the general public. Acceptance testing is a formal testing conducted to determine whether a system satisfies its acceptance criteria—the criteria the system must satisfy to be accepted by the customer. It helps the customer to determine whether or not to accept the system [1]. The customer generally reserves the right to refuse to take delivery of the product if the acceptance test cases do not pass. There are two categories of acceptance testing:

- User acceptance testing.
- Business acceptance testing.

The UAT is conducted by the customer to ensure that system satisfies the contractual acceptance criteria before being signed off as meeting user needs. Actual planning and execution of the acceptance tests do not have to be undertaken directly by the customer. Often third-party consulting firms offer their services to do this task. However, the customer must specify the acceptance criteria for the third party to seek in the product. The BAT is undertaken within the development organization

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

of the supplier to ensure that the system will eventually pass the UAT. It is a rehearsal of UAT at the premises of the supplier. The development organization of the supplier derives and executes test cases from the client's contract, which include the acceptance criteria.

The acceptance criteria must be defined and agreed upon between the supplier and the customer to avoid any kind of protracted arguments. Either party or a third-party consulting firm may design the acceptance test plan. The acceptance criteria document is a part of the contract in the case of an outsourced development under the OEM agreement. If some hardware is an integral part of the system, then the hardware acceptance criteria are included in the contractual agreement. In general, the marketing organization of the buyer defines the acceptance criteria. However, it is important that the software quality assurance team of the buyer's organization initiate a dialogue with the seller and provide a set of "straw man" acceptance criteria for the marketing department to review and react to. The users, the system engineers, customer support engineers, and the software quality assurance group of the buyer's organization do the actual planning and execution of the acceptance tests after the criteria are agreed upon. The personnel developing an acceptance test plan must have a thorough understanding of the acceptance criteria that have been agreed upon. It is unlikely that the system passes all the acceptance criteria in one go for large, complex systems. It is useful to focus on the following three major objectives of acceptance testing for pragmatic reasons:

- Confirm that the system meets the agreed-upon criteria. The broad categories of criteria are explained in Section 14.2.
- Identify and resolve discrepancies, if there are any. The sources of discrepancies and mechanisms for resolving them have been explained in Section 14.5.
- Determine the readiness of the system for cut-over to live operations. The final acceptance of a system for deployment is conditioned upon the outcome of the acceptance testing. The acceptance test team produces an acceptance test report which outlines the acceptance conditions. The details of an acceptance test report have been explained in Section 14.6.

Acceptance testing is only one aspect of the contractual fulfillment of the agreement between a supplier and a buyer. A contractual agreement may require the seller to provide other materials, such as the design solution document that addresses the requirement document of the buyer. The acceptance test team may evaluate the acceptability of the system design in terms of graphical user interface, error handling, and access control.

14.2 ACCEPTANCE CRITERIA

At the core of any contractual agreement is a set of acceptance criteria. A key question is what criteria must the system meet in order to be acceptable? The acceptance criteria must be measurable and, preferably, quantifiable. The basic

principle of designing the acceptance criteria is to ensure that the quality of the system is acceptable. One must understand the meaning of the quality of a system, which is a complex concept. It means different things to different people, and it is highly context dependent [2].

Even though different persons may have a different view about quality, it is the customer's opinion that prevails. The concept of quality is, in fact, complex and multifaceted [3]. Five views of quality, namely, *transcendental view*, *user view*, *manufacturing view*, *product view*, and *value-based view*, have been explained in Chapter 17. The five views were originally presented by Garvin [3] in the context of production and manufacturing in general and subsequently explained by Kitchenham and Pfleeger [2] in a software development context. The five views are presented below in a concise form:

1. The transcendental view sees quality as something that can be recognized but is difficult to describe or define.
2. The user view sees quality as satisfying the purpose.
3. The manufacturing view sees quality as conforming to the specification.
4. The product view ties quality with the inherent characteristics of the product.
5. The value-based view puts a cost figure on quality—the amount a customer is willing to pay for it.

Acceptance criteria are defined on the basis of these multiple facets of quality attributes. These attributes determine the presence or absence of quality in a system. Buyers, or Customers, should think through the relevance and relative importance of these attributes in their unique situation at the time of formulating the acceptance criteria. The attributes of quality are discussed below and examples of acceptance criteria for each quality attribute are given.

Functional Correctness and Completeness One can ask the question: Does the system do what we want it to do? All the features which are described in the requirements specification must be present in the delivered system. It is important to show that the system works correctly under at least two to three conditions for each feature as a part of acceptance.

One can show the functional correctness of a system by using the requirements database, as discussed in Chapter 11. The database is used in generating a requirement traceability matrix during system-level testing. Basically a traceability matrix tells us the test cases that are used to verify a requirement and all the requirements that are partially verified by a test case. Such a traceability matrix is a powerful tool in showing the customer about the functional correctness of the system. It is important to obtain an early feedback from the customer on the requirements traceability matrix. The idea behind the feedback is to reach an agreement on the validation method to be employed for each requirement. The decision is especially significant because some validation methods are easier to implement

and less time intensive than other methods. For example, the demonstration method is less time intensive than the testing method.

In reality, rigorous functional correctness testing is conducted during the system testing phase, rather than during acceptance testing. However, the buyer may ask for the requirement traceability matrix before the start of acceptance testing to ensure that the system does function according to the requirement specification.

Accuracy The question is: Does the system provide correct results? Accuracy measures the extent to which a computed value stays close to the expected value. Accuracy is generally defined in terms of the magnitude of the error. A small gap—also called an error in numerical analysis, for example—between the actual value computed by a system and the expected value is generally tolerated in a continuous space. The accuracy problem is different in discrete space, leading to false-positive and false-negative results. False positives and false negatives are serious drawbacks in any diagnostic and monitoring software tools.

Data Integrity Data integrity refers to the preservation of the data while it is transmitted or stored such that the value of data remains unchanged when the corresponding receive or retrieve operations are executed at a later time. Thus, data must not be compromised by performing update, restore, retrieve, transmit, and receive operations. The requirement of data integrity is included in the acceptance test criteria to uncover design flaws that may result in data corruption. In communication systems, an intruder can change the data without the sender and receiver detecting the change. If integrity check mechanisms are in place, the data may be changed, but the mechanism will detect the tampering. Data integrity mechanisms detect changes in a data set. The concepts of message digest and digital signature are used in preserving data integrity [4].

Remark. A *message digest* algorithm takes in an input message of arbitrary length and produces a fixed-length code. The fixed-length code is called a *digest* of the original message. The commonly used message digest algorithms are the Message Digest 5 (MD5) and the Secure Hash Algorithm 1 and 2 (SHA-1 and SHA-2).

Remark. A *digital signature* is an encrypted message digest that is appended to a document to be stored or transmitted. A message digest is obtained by using, for example, the MD5, SHA-1, or SHA-2 algorithm. The message digest is encrypted with the private key of the party that stores or transmits the message.

Data Conversion Data conversion is the conversion of one form of computer data to another. For example, conversion of a file from one version of Microsoft Word to an earlier version for the sake of those who do not have the latest version of Word installed. Data conversion testing is testing of programs or procedures that are used to convert data from existing systems for use in replacement systems. Data may be converted into an invalid format that cannot be processed by the new system if this is not performed properly; thus the data will have no value.

In addition, data may be omitted from the conversion process resulting in gaps or system errors in the new system. Inability to process backup or archive files results in the inability to restore or interrogate old data.

An acceptance criterion for data conversion measures and reports the capability of the software to convert existing application data to new formats. The following questions must be answered in specifying the data conversion acceptance criteria:

- How can we undo a conversion and roll back to the earlier database version(s) if necessary?
- How much human involvement is needed to validate the conversion results?
- How are the current data being used and how will the converted data be used?
- Will the data conversion software conduct integrity checking as well?

Backup and Recovery Backup and recovery of data are default functionalities of large, complex systems. This is because, though systems are not expected to crash, in reality, a system crash is not uncommon. The backup and recovery acceptance criteria specify the durability and recoverability levels of the software in each hardware platform. The aim of the recovery acceptance test criteria is to outline the extent to which data can be recovered after a system crash. The following questions must be answered in specifying the recoverability acceptance criteria:

- How much data can be recovered after a crash and how?
- Is the recovered data expected to be consistent?

Generally, a system cannot recover from a crash unless the data have been previously backed up. The backup process includes taking periodic snapshots of a state of the system and saving them in stable storage to be retrieved later [5]. The following questions must be answered in specifying the backup acceptance criteria:

- How frequently is the backup process initiated?
- How long does the backup process take?
- Is the backup expected to work on-line or off-line with normal operation suspended during backup?
- Does the backup process check if sufficient storage space is available to accommodate all the data?
- Is the backup process fully automated?

Competitive Edge The system must provide a distinct advantage over existing methods and competing products through innovative features. An analysis of the competitiveness of the product is provided to the buyer. This document contains a comparative study of the system with products available in the market from other vendors. A competitive analysis is conducted by the systems engineering group

of the marketing organization. The following questions need to be answered in specifying the competitive analysis report acceptance criteria:

- What are the nearest direct competitors of the product?
- What are the indirect competitors of the product?
- Who are the potential competitors?
- Is the business in the product area steady, growing, or declining?
- What can be learned from product operations or from advertisements of competitors?
- What are the strengths and weaknesses of competitors?
- How do their products differ from the product being developed?

Usability The question is: How easy it is to use the system and how easy it is to learn? The goal of usability acceptance criteria is to ensure that the system is flexible, it is easy to configure and customize the system, on-line help is available, work-around is available, and userinterface is friendly. The following questions need to be addressed in specifying the usability acceptance criteria:

- How will the system help the user in the day-to-day job?
- Will the productivity, customer satisfaction, reliability, and quality of work life of the user improve by using the system?
- Are the menus, commands, screens, and on-line help clear to a typical user?
- Are the user procedures simple, logical, and clear to the typical user?
- Is the user guide clear, easy to access, and understandable for a typical user?
- Will the methods of error and exception handling utilized by the system increase reliability and productivity?
- Are the reports generated by the system in order, consistent, and clear?
- Is the system easy to install?

Performance The desired performance characteristics of the system must be defined for the measured data to be useful. The following questions relate to specification of the performance acceptance criteria:

- What types of performance characteristics of the system need to be measured?
- What is the acceptable value for each performance parameter?
- With what external data source or system does the application interact?
- What kind of workload should be used while running the performance tests? The workload should be a representative of the likely real-world operating condition in terms of low load, average load, and peak load.

- Is it required to perform a before-and-after comparison of the performance results with the prior version of the system?

Start-Up Time The system start-up time reflects the time taken to boot up to become operational. The following questions address the start-up acceptance criteria:

- How is the start-up time defined?
- Does the start-up time include the power-on self-test of all the system hardware?
- What is the longest acceptable start-up time?

Stress The system should be capable of handling extremely high or stressful load. It is necessary to identify the system limitations and then stress the system to find the results when the system is pushed to the border and beyond. The system limitation must be identified in the acceptance criteria. The following questions must be addressed in specifying the stress acceptance criteria:

- What are the design limits of the system?
- What is the expected and acceptable behavior of the recovery mechanism?
- What test environment, close to customer deployment architecture, is needed in order to force the system to be stressed?

Reliability and Availability Software reliability is defined as the probability that the software executes without failure for a specified amount of time in a specified environment. The longer a system runs without failure, the more reliable it is. A large number of reliability models are available to predict the reliability of software. A software reliability model provides a family of growth curves that describe the decline of failure rate as defects are submitted and closed during the system testing phase. The failure rate is often calculated in terms of MTBF. A growth model can answer the following questions, which can be part of the reliability acceptance criteria:

- What is the current failure rate of the software?
- What will be the failure rate if the customer continues acceptance testing for a long time?
- How many defects are likely to be in the software?
- How much testing has to be performed to reach a particular failure rate?

The failure rate goal that is acceptable must be set separately for each level of problem severity—from *critical* to *low*. A customer may be willing to tolerate tens of low-severity issues per day but not more than one critical problem in a year.

System availability consists of proactive methods for maximizing service uptime, for minimizing the downtime, and for minimizing the time needed to

recover from an outage. Downtime is measured in terms of MTTR. The creation of a customer environment is facilitated by gathering an *operational profile* from the customer. An operational profile describes the ways the system is to be used. One can uncover several deficiencies in the system while tuning the parameters of the system; parameter tuning will improve system availability level. Customers must be willing to share the operational profile of their computing environment to improve the target availability level, which may be proprietary information.

Maintainability and Serviceability The maintainability of a system is its ability to undergo repair and evolution. One way to characterize maintainability is to measure the MTTR, which reflects the time it takes to analyze a *corrective* defect, design a modification, implement the change, test it, and distribute it. The important factors, from a customer's perspective, is the responsiveness of the service rather than the internal technical maintainability of the system. The following are useful acceptance criteria from a customer's perspective:

- The customer is the final arbiter of setting the severity of a system problem. If the customer calls a problem critical, it must be fixed immediately.
- If a system problem is assessed as critical by the customer, then staff must be assigned to work on resolving the problem immediately with utmost priority.
- If the severity of a system problem is assessed as high by the customer, then staff must be assigned to work on resolving the problem during normal business hours until it is resolved or until a work-around has been delivered as an interim solution. The staff responsible for resolving the problem must ensure that there is significant effort made toward resolving the problem. However, they may spend time on other activities as priorities dictate.
- If a system problem is assessed as low by the customer, then staff must be assigned to work on resolving the problem during normal business hours as time permits. If the problem solution involves a software change, it will normally wait until the next software release has been implemented to provide the resolution.
- All the critical- and high-severity fixes must work 100% when installed.

Serviceability is closely related to maintainability of the system, which are designed to ensure the correctness of the tools that are used to diagnose and service the system. For example, the software may need to be serviced remotely via an Internet connection. Diagnostic utilities are used to monitor the operation and the cause of any malfunction. The following questions must be addressed in specifying the serviceability acceptance criteria:

- What kind of tools will be available for servicing the system?
- How should these tools be used?

Robustness The robustness of a system is defined as its ability to recover from errors, continue to operate under worst conditions, and operate reliably for an extended period of time. The following questions must be addressed in specifying the robustness acceptance criteria:

- What are the types of errors from which the system is expected to recover?
- What are the causes, or sources, of the errors so that these can be simulated in a test environment?
- How are the errors initiated, or triggered, in the real world?
- What types of corrective and recovery actions are required for each type of error?
- What kinds of disasters can strike? What are those scenarios?
- What is an acceptable response to each of these identified scenarios?
- What is the recovery mechanism for each of the scenarios? Is it workable, understood, and accepted?
- How can disaster be simulated in order to test recovery?

Timeliness Time to market is an important aspect of any contractual agreement. The supplier must be able to deliver the system to the buyer within the time frame agreed upon. Rewards and penalties are associated with the timeliness acceptance criteria as follows:

- If coding is completed on time, the buyer will reward 5% extra money on top of the contractual agreement.
- If system-level testing is completed on time, the buyer will reward 10% extra money on top of the contractual agreement.
- For every week of delay in completing the system tests, the supplier has to pay 2% penalty on top of the contractual agreement, with a maximum of 20% penalty.

Confidentiality and Availability The confidentiality acceptance criteria reter to the requirement that the data must be protected from unauthorized disclosure and the availability acceptance criteria to the requirement that the data must be protected from a denial of service (DoS) to authorized users. Different types of possible confidentiality and availability acceptance criteria are as follows:

- No unauthorized access to the system is permitted, that is, user authentication is performed.
- Files and other data are protected from unauthorized access.
- The system is protected against virus, worm, and bot attacks.
- Tools are available for detecting attacks.
- There is support against DoS attack.
- Privacy in communication is achieved by using encryption.

- All the customer data must be stored in a secure place in accordance with the policies of customer right, such as confidentiality.

Remark. A worm is defined as a software component that is capable of, under its own means, infecting a computer system in an automated fashion. On the other hand, a virus spreads rapidly to a large number of computers. However, it cannot do so with its own capability; it spreads using the assistance of another program.

Remark. A bot is a software agent. A bot interacts with other network services intended for people as if it were a person. One typical use of bots is to gather information. Another more malicious use for bots is the coordination and operation of an automated attack on networked computers, such as a distributed DoS attack.

Compatibility and Interoperability The compatibility of a system is defined as the ability to operate in the same way across different platforms and network configurations and in the *presence* of different mixes of other applications. On the other hand, the interoperability of a system is defined as the ability to *interface* with other network elements and work correctly as expected. The major challenge is in determining the platforms, configurations, and other applications with which the system is compatible. The following questions must be addressed in specifying the compatibility and interoperability acceptance criteria:

- What are the platforms, or configurations, on which the system must operate?
- Does the system have to work exactly the same way across different configurations? If not, what are the acceptable variations?
- What are the applications that must coexist with the system?
- With what network elements must the system interoperate?

Compliance The system should comply with the relevant technical standards, such as the IEEE standards, operating system interface standards, and the IP standards. In addition, the system should comply with regulatory requirements as established by external agencies. The following questions must be addressed in specifying the acceptance criteria for compliance:

- With what technical standards should the system comply? Are there any exceptions to these standards? If yes, specify the exceptions.
- Identify the regulatory bodies that must certify the system?

Installability and Upgradability The purpose of system installability and upgradability is to ensure that the system can be correctly installed and upgraded in the customer environment. If for some reason the customer wants to uninstall or downgrade the system software, it is required to be done smoothly. Installation and upgradation of a system is planned by identifying the major milestones and

contingency steps. The system installation and upgradation process document must be available with specific steps. The acceptance criteria of system installation and upgradation are as follows:

- The document must identify the person to install the system, for example, the end user or a trained technician from the supplier side.
- Over what range of platforms, configurations, and versions of support software is the installation or upgradation process expected to work? The hardware and software requirements must be clearly explained in the document.
- Can the installation or upgradation process change the user's existing environment? If yes, the risks of this change should be clearly documented.
- The installation or upgradation process should include diagnostic and corrective steps to be used in the event of the process not progressing as expected.
- The installation or upgradation process should contain a workable uninstall, downgrade, or backoff process in case a specific installation does not proceed as expected.
- The installation or upgradation process should correctly work from all of the various delivery media, such as download via File Transfer Protocol (FTP), CD-ROM, and DVD.
- If the system includes a licensing and registration process, it should work smoothly and should be documented.
- The installation or upgradation instructions should be complete, correct, and usable.
- The installation or upgradation process should be verified during system testing.
- There should be zero defects outstanding against a system installation or upgradation process.

Scalability The scalability of a system is defined as its ability to effectively provide acceptable performance as the following quantities increase: (i) geographic area of coverage of a system, (ii) system size in terms of the number of elements in the system, (iii) number of users, and (iv) volume of workload per unit time. A system may work as expected in limited-use scenarios but may not scale up very well. The following questions must be addressed in specifying the scalability acceptance criteria:

- How many concurrent users is the system expected to handle?
- How many transactions per unit time is the system expected to process?
- How many database records is the system expected to support?
- How many elements, or objects, must be managed in live operation?
- What is the largest geographic area the system can cover?

Documentation The quality of the system user's guide must be high. The documentation acceptance criteria are formulated as follows:

- All the user documents should be reviewed and approved by the software quality assurance group for correctness, accuracy, readability, and usefulness.
- The on-line help should be reviewed and signed off by the software quality assurance group.

14.3 SELECTION OF ACCEPTANCE CRITERIA

The acceptance criteria discussed above provide a broad idea about customer needs and expectations, but those are too many and very general. The customer needs to select a subset of the quality attributes and prioritize them to suit their specific situation. Next, the customer identifies the acceptance criteria for each of the selected quality attributes. When the customer and the software vendor reach an agreement on the acceptance criteria, both parties must keep in mind that satisfaction of the acceptance criteria is a trade-off between time, cost, and quality. As Ed Yourdon opined, sometimes less than perfect is good enough [6]. Only business goals and priority can determine the degree of "less than perfect" that is acceptable to both the parties. Ultimately, the acceptance criteria must be related to the business goals of the customer's organization.

Many organizations associated with different application domains have selected and customized existing quality attributes to define quality for themselves, taking into consideration their specific business and market situation. For example, IBM used the quality attribute list CUPRIMDS—capability, usability, performance, reliability, installation, maintenance, documentation, and service—for its products [7]. Similarly, for web-based applications [8], a set of quality attributes are identified in decreasing order of priority: reliability, usability, security, availability, scalability, maintainability, and time to market. Such a prioritization scheme is often used in specific application domains. For example, usability and maintainability take precedence over performance and reliability for a word processor software. On the other hand, it might be the other way around for a real-time operating system or telecommunication software.

14.4 ACCEPTANCE TEST PLAN

Planning for acceptance testing begins as soon as the acceptance criteria are known. Early development of an acceptance test plan (ATP) gives us a good picture of the final product. The purpose of an ATP is to develop a detailed outline of the process to test the system prior to making a transition to the actual business use of the system. Often, the ATP is delivered by the vendor as a contractual agreement, so that the business acceptance testing can be undertaken within the vendor's development organization to ensure that the system eventually passes the acceptance test.

In developing an ATP, emphasis is put on demonstrating that the system works according to the customer's expectation, rather than passing a set of comprehensive tests. In any case, the system is expected to have already passed a set of comprehensive tests during system-level testing. The ATP must be kept very simple because the audience of this plan may include people from diverse backgrounds, such as marketing and business managers. Some people argue that the ATP is redundant and unnecessary if a comprehensive system test plan is developed. We believe that even if a system test plan is adequate, acceptance tests usually uncover additional significant problems. Moreover, user's concerns are not addressed during system-level testing.

An ATP needs to be written and executed by the customer's special user group. The user group consists of people from different backgrounds, such as software quality assurance engineers, business associates, and customer support engineers. In addition, the acceptance test cases are executed at the user's operational environment, whereas the system-level test cases are executed in a laboratory environment. An overall test plan for acceptance testing and description of specific tests are documented in the ATP. The structure of a typical ATP is outlined in Table 14.1.

The introduction section of the ATP describes the structure of the test plan and what we intend to accomplish with this test plan. This section typically includes (i) test project name, (ii) revision history, (iii) terminology and definitions, (iv) names of the approvers and the date of approval, (v) an overview of the plan, and (vi) references.

For each quality category from the acceptance criteria signed-off document two subsections are created: operational environment and test case specification. The operational environment deals with discussion on site preparation for the execution of the acceptance test cases. Test cases are specified for each acceptance criteria within the quality category.

An outline of the timeline of execution of acceptance tests is provided in the schedule section of the ATP. Acceptance test execution is not intended to be exhaustive, and therefore it does not continue for long. The acceptance test

TABLE 14.1 Outline of ATP

1. Introduction

2. Acceptance test category. For each category of acceptance criteria:

- (a) Operational environment
- (b) Test case specification
  - (i) Test case ID number
  - (ii) Test title
  - (iii) Test objective
  - (iv) Test procedure

3. Schedule

4. Human resources

may take up to six weeks for a large system. The point here is that comprehensive acceptance testing, to the same extent and depth as targeted by system-level testing, is not required to demonstrate that the acceptance criteria are satisfied by the system.

The human resources section of the ATP deals with (i) the identification of the acceptance testers that form the client organization and (ii) their specific roles in the execution of acceptance test cases. The section includes acceptance test site preparation, overseeing installation of new hardware, upgrading the software, and setting up of the networks. These are the people who are knowledgeable in the operational environment and business operations. In addition, the human resources requirement from the supplier organization during the acceptance testing is included in this section. These engineers are usually from the supplier's system test group, who participated in testing the system.

The ATP is reviewed and approved by the relevant groups, such as the marketing, customer support, and software quality assurance groups. It can be shared with the system supplier organization.

14.5 ACCEPTANCE TEST EXECUTION

The acceptance test cases are divided into two subgroups. The first subgroup consists of basic test cases, and the second consists of test cases that are more complex to execute. The acceptance tests are executed in two phases. In the first phase, the test cases from the basic test group are executed. If the test results are satisfactory, then the second phase, in which the complex test cases are executed, is taken up. In addition to the basic test cases, a subset of the system-level test cases are executed by the acceptance test engineers to independently confirm the test results. Obviously, a key question is: Which subset of the system-level test cases are selected? It is recommended to randomly select 5–10 test cases from each test category. If a very large fraction, say more than 0.95, of the basic test cases pass, then the second phase can proceed. It may be counterproductive to carry out the execution of the more complex tests if a significant fraction of the basic tests fail.

Acceptance test execution is an important activity performed by the customer with much support from the developers. The activity includes the following detailed actions:

- The developers train the customer on the usage of the system.
- The developers and the customer coordinate the fixing of any problem discovered during acceptance testing.
- The developers and the customer resolve the issues arising out of any acceptance criteria discrepancy.

System-level test personnel from the development organization travel to the customer location where the acceptance tests are to be conducted. They assist the

TABLE 14.2 ACC Document Information

1. ACC number	A unique number
2. Acceptance criteria affected	Existing acceptance criteria
3. Problem/issue description	Brief description of issue
4. Description of change required	Description of changes needed to be done to original acceptance criterion
5. Secondary technical impacts	Description of impact it will have on system
6. Customer impacts	Impact it will have on end user
7. Change recommended by	Name of acceptance test engineer(s)
8. Change approved by	Name of approver(s) from both parties

customer in preparing a test site and train the acceptance test engineers on how to use the system. They provide the earlier system-level test results to the customer's test engineers in order to make informal decisions about the direction and focus of the acceptance testing effort. In addition, the on-site system test engineers answer the customer's questions about the system and assist the acceptance test engineers in executing the acceptance tests.

Any defect encountered during acceptance testing are reported to the software development organization through the on-site system test engineers. The defects are submitted through the defect tracking system. The software build is retested by the supplier and a satisfactory software image is made available to the customer for continuation of acceptance testing when the defects are fixed. The failed tests are repeated after the system is upgraded with a new software image. An agreement must be reached between the on-site system test engineers and the acceptance test engineers when to accept a new software image for acceptance testing. The number of times the system can be upgraded to a new software image during acceptance testing is negotiated between the customer and the supplier. Multiple failures of a system during acceptance testing are an indication of poor system quality.

It is possible that an acceptance test engineer may encounter issues related to acceptance criteria during the execution of acceptance test cases. The system may not provide services to the users as described in the acceptance criteria. Any deviation from the acceptance criteria discovered at this stage may not be fixed immediately. The acceptance test engineer may create an acceptance criteria change (ACC) document to communicate the deficiency in the acceptance criteria to the supplier. A representative format of an ACC document is shown in Table 14.2. An ACC report is generally given to the supplier's marketing department through the on-site system test engineers.

14.6 ACCEPTANCE TEST REPORT

Acceptance test activities are designed to reach one of three conclusions: Accept the system as delivered, accept the system after the requested modifications have

been made, or do not accept the system. Usually some useful intermediate decisions are made before making the final decision:

- A decision is made about the continuation of acceptance testing if the results of the first phase of acceptance testing are not promising. One may recall that the basic tests are executed in the first phase.
- If the test results are unsatisfactory, changes will be made to the system before acceptance testing can proceed to the next phase.

The intermediate decisions are made based on evaluation of the results of the first phase of testing. Moreover, during the execution of acceptance tests, the status of testing is reviewed at the end of every working day by the leader of the acceptance test team, on-site system test engineers, and project managers of the customer and the supplier. The acceptance team prepares a test report which forms the basis of discussion at the review meeting before they meet for a review. A template of the test report is given in Table 14.3.

The test report is reviewed on a daily basis to understand the status and progress of acceptance testing. If serious problems are encountered during acceptance testing, the project manager flags the issues to the senior management.

At the end of the first and the second phases of acceptance testing an acceptance test report is generated by the test team leader. A template for a test report is outlined in Table 14.4. Most of the information from the test status report can be used in the acceptance test summary report.

The report identifier uniquely identifies the report. It is used to keep track of the document under version control.

The summary section summarizes what acceptance testing activities took place, including the test phases, releases of the software used, and the test environment. This section normally includes references to the ATP, acceptance criteria, and requirements specification.

The variances section describes any difference between the testing that was planned and the actual testing carried out. It provides an insight into a process for improving acceptance test planning in the future.

TABLE 14.3 Structure of Acceptance Test Status Report

1. Date	Acceptance report date
2. Test case execution status	Number of test cases executed today
	Number of test cases passing
	Number of test cases failing
3. Defect identifier	Submitted defect number
	Brief description of issue
4. ACC number(s)	Acceptance criteria change document number(s), if any
5. Cumulative test execution status	Total number of test cases executed
	Total number of test cases passing
	Total number of test cases failing
	Total number of test cases not executed yet

TABLE 14.4 Structure of Acceptance Test Summary Report

1. Report identifier
2. Summary
3. Variances
4. Summary of results
5. Evaluation
6. Recommendations
7. Summary of activities
8. Approval

In the summary of results section of the document test results are summarized. The section gives the total number of test cases executed, the number of passing test cases, and the number of failing test cases; identifies all the defects; and summarizes the acceptance criteria to be changed.

The evaluation section provides an overall assessment of each category of the quality attributes identified in the acceptance criteria document, including their limitations. This evaluation is based on the test results from each category of the test plan. The deviations of the acceptance criteria that are captured in the ACC during the acceptance testing are discussed.

The recommendations section includes the acceptance test team's overall recommendation: (i) unconditionally accept the system, (ii) accept the system subject to certain conditions being met, or (iii) reject the system. However, the ultimate decision is made by the business experts of the supplier and the buyer organization.

The summary of activities section summarizes the testing activities and the major events. This section includes information about the resources consumed by the various activities. For example, the total manpower involved in and the time spent for each of the major testing activities are given. This section is useful to management for accurately estimating future acceptance testing efforts.

Finally, the names and titles of all the people that will approve this report are listed in the approvals section. Ideally, the approvers of this report should be the same people who approved the corresponding ATP because the summary report describes all the activities outlined in the ATP. If some of the reviewers have minor disagreements, they may note their views before signing off on the document.

14.7 ACCEPTANCE TESTING IN eXtreme PROGRAMMING

In the XP[9] framework user stories are used as acceptance criteria. The user stories are used to derive time estimates for each development iteration in release planning (time-to-market acceptance criteria) and acceptance tests. The user stories are written by the customer as things that the system needs to do for them. The

stories are usually about two to three sentences of text written using the customer's terminology. Several acceptance tests are created to verify that the user story has been correctly implemented. Acceptance tests are specified in a format that is clear enough that the customer can understand and specific enough that it can be executed.

The customer is responsible for verifying the correctness of the acceptance tests and reviewing the test results [10]. The acceptance test results are reviewed by the customer to decide what failed tests are of highest priority that must pass during the next iteration. A story is incomplete until it passes its associated acceptance tests.

Acceptance tests are executed by an acceptance test group, which is a part of the development team. Ideally, acceptance tests should be automated using either the unit testing framework or a separate acceptance testing framework before coding. Acceptance test engineers and customers can run the tests multiple times per day as a regression acceptance test suite after the acceptance tests are automated. An automated acceptance test suite does not lose its value even after the customer has approved the successful implementation of the user story in a development iteration. The acceptance tests take on the role of regression tests to ensure that subsequent changes to the system do not affect the unaltered functionality.

14.8 SUMMARY

This chapter began with an introduction to two types of acceptance testing: user acceptance testing and business acceptance testing. Next, the chapter described acceptance criteria in terms of quality attributes. Formulation of acceptances criteria is governed by the business goals of the customer's organization.

We presented an outline of an acceptance test plan and described in detail how to create such a plan. Emphasis must be put on the notion that the system works according to the customer's expectations in developing an acceptance test plan, rather than just passing comprehensive testing. Less emphasis is put on a system passing a comprehensive set of tests because rigorous testing is assumed to have already occurred during the system testing phase.

Next, we discussed the execution of acceptance tests, which is an important activity performed by the customer with much needed support from the developers. Three major activities were identified and discussed: (i) providing training to the customer's test engineers, (ii) fixing problems during acceptance testing, and (iii) resolving issues concerning any discrepancy related to acceptance criteria. After that, we described the generation of an acceptance test report, which must be completed at the end of acceptance testing.

Finally, we explained how user stories are used in XP as acceptance criteria and acceptance test cases are created. These tests are reviewed, automated, and executed multiple times per day as a regression acceptance test suite in the presence of on-site customers.

LITERATURE REVIEW

A thorough treatment of software quality standard can be found in ISO/IEC 9126-1:2001, "Software Engineering—Product Quality—Part-I: Quality Model." The standards group has recommended a hierarchical framework for software quality characteristics and subcharacteristics. Six top-level software quality characteristics are *functionality, reliability, usability, efficiency, maintainability*, and *portability*.

An excellent taxonomy in the field of the *dependability* quality attribute is presented in the article by Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 1, January–March 2004, pp. 11–33. In this article, the relationship between dependability and *security* quality attributes is discussed in great detail. Dependability is defined as an integrating concept that encompasses the following attributes: availability, reliability, safety, integrity, and maintainability. On the other hand, security brings in concerns for confidentiality in addition to availability and integrity. The article addresses the concept of threats to dependability and security (faults, errors, failures), their attributes, and the means for their achievement, such as fault prevention, fault tolerance, fault removal, and fault forecasting.

An interesting view of software quality characteristics is presented in C. K. Prahalad and M. S. Krishnan, "The New Meaning of Quality in the Information Age," *Harvard Business Review*, September 1999, pp. 109–118. The authors provide a framework for measuring the performance of software in a company's information technology portfolio based on three basic attributes: conformance, adaptability, and innovation. To judge software quality properly, the authors argue that managers must measure applications against all the three attributes.

The students are encouraged to read the article by P. Hsia, D. Kung, and C. Sell, "Software Requirements and Acceptance Testing," *Annals of Software Engineering*, Vol. 3, No. 0, January 1997, pp. 291–317. The authors present a systematic approach to scenario analysis and its application to acceptance testing. Based on the model, different acceptance test criteria are defined, and various types of acceptance testing are discussed.

The book entitled *Testing Extreme Programming* (L. Crispin and T. House, Addison-Wesley, Boston, MA, 2003) gives an excellent exposition of unit and acceptance testing in XP. The book is divided into three parts: the XP tester role, test drive through an XP project, and road hazard survival kit. The book lists several excellent books on the subject in its bibliography.

REFERENCES

1. Institute of Electrical and Electronics Engineers (IEEE). *IEEE Standard Glossary of Software Engineering Terminology*. IEEE standard 610.12-1990. IEEE, New York, 1990.
2. B. Kitchenham and S. L. Pfleeger. Software Quality: The Elusive Target. *IEEE Software*, January 1996, pp. 12–21.

3. D. A. Garvin. What Does Product Quality Really Mean? *Sloan Management Review*, Fall 1984, pp. 25–45.
4. K. Sankar, S. Sundaralingam, A. Balinsky, and D. Miller. *Cisco Wireless LAN Security*. Cisco Press, Indianapolis, IN, 2005.
5. U. Troppens, R. Erkens, and W. Müller. In *Storage Networks Explained (Chapter 7: Network Back-up)*. Wiley, Hoboken, NJ, 2004.
6. E. Yourdon. When Good Enough Software Is Best. *IEEE Software*, May 1995, pp. 79–81.
7. S. H. Kan. *Metrics and Models in Software Quality Engineering*, 2nd ed. Addison-Wesley, Reading, MA, 2002.
8. J. Offutt. Quality Attributes of Web Application. *IEEE Software*, February 2002, pp. 25–32.
9. K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley, Reading, MA, 2004.
10. R. Mugridge. Managing Agile Project Requirements with Storytest-Driven Development. *IEEE Software*, January 2008, pp. 68–75.

Exercises

1. What are the objectives of acceptance testing?
2. What are the differences between UAT and BAT?
3. Discuss the advantages and disadvantages of customer involvement in testing.
4. What is software quality?
5. Who should define the acceptance quality attribute criteria of a test project. Justify your answer?
6. What other quality attributes not mentioned in this book can you think of?
7. What is meant by DoS attack? Explain its importance in acceptance testing.
8. The security quality attribute is not explicitly discussed in this book. However, it can be a combination of subattributes, which are discussed in this book. What are those quality subattributes?
9. In the following series of application examples, provide four most important quality attributes discussed in this book that you think are critical to the applications. Why are the quality attributes chosen by you critical ones?
   (a) Telecommunication software
   (b) Video game software
   (c) Intrusion detection system (IDS) software
   (d) Medical diagnostic software for assisting doctors
   (e) Highly popular music download website
   (f) Operating system on deep space mission probe
   (g) Pension payment calculation system
   (h) E-mail system
   (i) Aircraft system software
   (j) Semiautomated telephone service to help answer questions about immigration and citizenship laws
10. Why are acceptance test cases executed in two phases?

11. Why are system test engineers required to be present at the ATP execution site?
12. For the current software project you are working on, answer the following questions:
    (a) List the quality attributes most important to your project. In order to focus on the business acceptance tests, select no more than six quality attributes from this list as the most critical acceptance criteria for your system.
    (b) Why are the selected acceptance criteria the most critical ones for your system?
    (c) Are there any other acceptance criteria which apply to your system but not listed in this book?
13. Based on the six selected acceptance criteria you identified for your software project in the previous questions:
    (a) Develop a business ATP.
    (b) Execute the business ATP against the system.
    (c) Generate an ACC document if you observed any deviation from the acceptance criteria during the execution of the ATP.
    (d) Create an acceptance test summary report.

CHAPTER 15

Software Reliability

Failure is only postponed success as long as courage ''coaches'' ambition. The habit of persistence is the habit of victory.
— *Herbert Kaufman*

15.1 WHAT IS RELIABILITY?

The concept of *reliability* is very broad, and it can be applied whenever someone expects something or someone else to "behave" in a certain way. For example, a lighting switch is expected to stay in one of two states—on and off—as set by its user. If a lighting switch causes a lamp to flicker even when the power supply is stable and the connecting cables are fault free, we say that the switch has turned unreliable. As a customer, we expect a new switch not to cause any flicker. We say that the switch is faulty if it causes a lamp to flicker. Moreover, a new switch may operate in a fault-free manner for several years before it starts causing the lamp to flicker. In that case we say that wear and tear have caused the switch to malfunction, and thus the switch has become unreliable. Therefore, the concept of fault is intertwined with the concept of reliability.

An initially fault-free hardware system behaves in a reliable manner simply because it operates as expected due to absence of faults in its design and manufacturing. Once the new hardware system starts functioning, wear and tear can cause it to develop faults, thereby causing the system to behave in an unexpected, or unreliable, manner. The sources of failures in software systems are design and implementation faults. Software failures usually occur when a program is executed in an environment that it was not developed for or tested for.

Reliability is one of the metrics used to measure the quality of a software system. It is arguably the most important quality factor sought in a product. Reliability is a user-oriented quality factor relating to system *operation*, and it takes into account the frequency of system failure. Intuitively, if the users of a system rarely experience system failure, then the system is considered to be more reliable than one that fails more often. The more and more failures of a system are observed by

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

users, the less and less the system is perceived to be reliable. Ideally, if there are no faults in a software system, users will never experience system failure, thereby considering the system to be highly reliable. Constructing a "correct" system, that is, a fault-free system, is a difficult task by itself given that real-life systems are inherently complex. The problem of constructing a fault-free system becomes more difficult when real-life factors are considered. For example, developing software systems is largely an economic activity, and companies may rarely have all the time and money needed to produce a highly reliable system even when they have a team of highly qualified and experienced personnel in the best case. The concept of *market window* may not allow a company to make an effort at producing a "correct" system. A market window is considered to be a time interval available for the introduction of a product before the product is surpassed in capability or cost by another from a competing vendor. Economic considerations may drive companies to trade off reliability in favor of cutting cost and meeting delivery schedule.

Given that there are an unknown number of faults in a delivered system and users can tolerate some failures, system reliability is best represented as a continuous variable rather than a Boolean variable. More efforts put into the development phase generally lead to a higher degree of reliability. Conversely, less effort produces systems with lower reliability. Thus, the concept of reliability can be used in examining the significance of trends, in setting goals, and in predicting when those goals will be achieved. For example, developers may be interested in knowing how a certain development process, how the length of system testing, or how a design review technique impacts software reliability. Developers may be interested in knowing the rate of system failure in a certain operational environment so as to be able to decide when to release the product.

Software maintenance involves making different kinds of changes to the system, namely, changes to requirements, changes to the design, changes to source code, and changes to test cases. While making those changes, the software goes through a phase of instability. The instability is in the form of reduced reliability of the system. The reliability level of a system decreases during system maintenance because additional defects may be introduced while making all those kinds of changes to the system. Intuitively, the smaller the amount of changes made to a system, the lesser the degradation in the current reliability level of the system. On the other hand, too many changes being simultaneously made to the system may significantly degrade the reliability level. Thus, the amount of changes that should be made to a product at a given time is dependent upon how much reliability one is ready to sacrifice for the moment.

15.1.1 Fault and Failure

The notions of *fault, failure*, and *time* are central to our understanding of reliability. In general, if a user never observes failures, the system is considered to be very reliable. On the other hand, a frequently failing system is considered to be highly unreliable. Therefore, we revisit the terms fault and failure in this section. A failure is said to occur if the *observable* outcome of a program execution is different from the expected outcome. The concept of observable outcome is very

broad, and it encompasses a variety of things, such as values produced, values communicated, performance demonstrated, and so on. The expected outcomes of program executions are specified as system requirements. System requirements are generally stated in explicit form in a requirements document. Sometimes, due to the complex nature of software systems, it is extremely difficult to explicitly state all the desired requirements. Where it is not possible to state all the requirements in an explicit manner, we still expect a system to behave in certain ways. Thus, while considering what is an expected outcome of program execution, one must take into account both the explicitly stated and the implicitly expected system requirements. Two characteristics of failures are as follows: (i) failures are associated with actual program executions and (ii) failures are observable concepts.

The adjudged cause of a failure is called a fault. While constructing a software system, a designer may mistakenly introduce a fault into the system. A fault can be introduced by having a defective block of code, a missing block of code for an unforeseen execution scenario, and so on. The mere presence of faults in a system does not cause failure. Rather, one or more defective portions of a system must execute for a fault to manifest itself as a system failure. One fault can cause more than one failure depending upon how the system executes the faulty code [1].

15.1.2 Time

Time plays a key role in modeling software reliability metrics. Let us go back to the concept of reliability of a lighting switch. Assume that a switch causes a lamp to flicker for a couple seconds, and let the average time gap between two successive flickers be six months. Here, flickering of the lamp can be considered as an observable failure of the switch. Though users may be irritated by such flickers, some may still consider the switch to be reliable because of the long time gap between two flickers. A time gap of six months can be considered to be long because a user may not remember when the lamp flickered last. However, if the lamp flickers a few times everyday, the switch appears less reliable and becomes a candidate for replacement. Thus, the notion of time is important in understanding the concept of reliability.

There are two commonly considered time models in the study of software reliability:

- Execution time (τ)
- Calendar time (t)

The *execution time* for a software system is the actual time spent by a processor in executing the instructions of the software system. Sometimes a processor may execute code from different software systems, and therefore, their individual execution times must be considered separately. On the one hand, there are many software systems which are expected to run continuously for months and years without developing failure. Examples of such systems are telephone switching software, air-traffic control software, and software for monitoring power plants. On the other hand, there are software systems which run for a while and terminate upon completion of their tasks. For example, a compiler terminates after compiling

a program. Though the concept of software reliability can be applied to all kinds of software systems, the ones running continuously draw more attention.

Calendar time is the time more commonly understood and followed in everyday life by all, including software engineers and project managers. Since observation of failures, fixing the corresponding faults, and replacing an old version of a software with a new one are intertwined in the life cycle of software systems, calendar time is useful for pragmatic reasons such as engineers leaving a company, engineers going on leave, and so on.

A third time model occasionally referred to is the *clock time* of a software system. Clock time refers to the elapsed time between the start and the end of program execution. Clock time includes the wait time of the software system and execution times of other software systems. Clock time does not include system shutdown.

In order to have a better understanding of the reliability of a software, one can ask the following questions in terms of failure and time:

- What is the time interval between two successive failures?
- How many failures have been observed within a certain time interval, for example, in the past one month or one year?
- What is the total number of failures observed so far?

The answers to the above questions give an indication of the quality level of a software. One can ask the above questions at any instant after code is developed. By asking the above questions during development, a test manager can monitor how the quality of the software is improving. On the other hand, by answering the questions during the operation of the system, one can know the delivered quality of the product.

15.1.3 Time Interval between Failures

One can infer several characteristics of a software system by monitoring the time interval between successive failures as follows:

- A small time interval between successive failures tells us that the software system is failing frequently, and hence the reliability level is too low. This can happen during system testing or even while the system is in operation. If the time interval between successive failures is long, the reliability is perceived to be high, in spite of the occasional system failure. The concepts of "short" and "long" time intervals between successive failures is a matter of user perception that is largely defined by the consequences of the failures. For example, if the operating system of a personal computer crashes about once in a year, the users can still consider the system to be reliable. On the other hand, if an air-traffic control software installed in a major international airport crashes once in a year, the system will be considered to be highly unreliable.

  In the hardware industry, a number of reliability metrics have been identified by considering the instants when failures occur and the instants

when the corresponding faults are repaired. The three commonly used metrics based on time intervals are the mean time to failure (MTTF), the mean time to repair (MTTR), and the mean time between failure (MTBF). When a failure occurs, it takes a certain amount of time to repair the system. The mean of all the repair times is the MTTR. We assume that a system is not in operation while it is being repaired. Thus, the mean of all the time intervals between the completion of a repair task and the occurrence of the next failure is the MTTF. The mean of the time intervals between successive failures is the (MTBF). The terms MTTR, MTTF, and MTBF are illustrated in Figure 15.1. A useful relationship between the three metrics can be stated as MTBF = MTTF + MTTR. It is easy to verify the relationship by considering the time intervals in Figure 15.1.

- At the beginning of system-level testing, usually a large number of failures are observed with small time intervals between successive failures. As system testing continues and faults are actually fixed, the time interval between successive failures increases, thereby giving an evidence that the reliability of the product is increasing. By monitoring the time interval between successive failures, one gets an idea about the reliability of the software system.

15.1.4 Counting Failures in Periodic Intervals

Useful information can be gathered by monitoring the cumulative failure count on a periodic basis. For example, we record the cumulative failure count every

Figure 15.1 Relationship between MTTR, MTTF, and MTBF.

month, plot it as a bar chart, and observe the pattern. Such monitoring can be done during system testing as well as while a system is in operation. During system testing, monitoring can be done at a fast pace, such as once per week, because of the large number of failures observed during testing. Development engineers and managers would like to observe the improvement in the reliability of the system under development. For a system already in operation, monitoring can be done once or twice per year. One can observe a number of system failures during its operation because of two reasons. First, there is a residual number of faults in most systems and systems might not have been tested in their real execution environments. Second, additional faults might be inadvertently introduced into a system during system maintenance.

15.1.5 Failure Intensity

By counting the total number of failures observed so far and plotting the data as a function of time, one can observe the change in the reliability of the system. This information is useful while a software is undergoing system-level testing as well as while a system is in operation. Ideally, the reliability level should reach a stable value after a while. A rising graph of the cumulative number of failures tells us that there are more faults in the system, and hence the system is perceived to be unreliable. The rate of rising of the graph is the rate at which failures are being observed. If the rate of rising is very small, we know that failures are infrequently being observed.

Two failure-related quantities that are generally used in defining software reliability are *cumulative failure*, denoted by the symbol μ, and *failure intensity*, denoted by the symbol λ. Failure intensity is expressed as the number of failures observed per unit of time. One can consider execution time τ or calendar time t. In this chapter, we consider the execution time τ in modeling software reliability. However, one can use calendar time as well. For example, if 10 failures are observed per hour of CPU time, then we express failure intensity as 10 failures per CPU hour. Similarly, if 30 failures are observed every two hours, then we express failure intensity as 15 (=30/2) failures per CPU hour.

Both μ and λ are functions of time. The cumulative failure count μ is a function of time because as time passes the total failure count monotonically increases. Moreover, λ is a function of time because as more and more failures are observed and the corresponding faults are actually fixed, we observe fewer and fewer number of failures per unit of execution time. Ideally, λ decreases with τ. But, in reality, λ can be a random function of τ as developers may introduce more faults while attempting to fix a known fault. The quantities $\mu(\tau)$ and $\lambda(\tau)$ are explained as follows:

- The quantity $\mu(\tau)$ denotes the total number of failures observed until execution time τ from the beginning of system execution.
- The quantity $\lambda(\tau)$ denotes the number of failures observed per unit time after τ time units of executing the system from the beginning. This quantity is also called the failure intensity observed at time τ.

Given these definitions, it is not difficult to establish the following relationship between $\mu(\tau)$ and $\lambda(\tau)$:

$$\lambda(\tau) = \frac{d\mu(\tau)}{d\tau}$$

Conversely, if we know the *failure intensity function*, we obtain the *mean value function* [2]:

$$\mu(\tau) = \int_0^{\tau} \lambda(x)\, dx$$

15.2 DEFINITIONS OF SOFTWARE RELIABILITY

There are two commonly used software reliability metrics, namely, the *probability of failure-free operation* and the *failure intensity*. The concept of failure is common to both definitions. One metric is probabilistic in nature, whereas the other is an absolute one. The two definitions are not contradictory. Rather, both can be simultaneously applied to the same software system without any inconsistency between them. User expectations from different systems may be different, and therefore it is more useful to apply one of the two to a given system. In the following, first the two metrics are defined, followed by examples of their applications to different systems.

15.2.1 First Definition of Software Reliability

Definition. Software reliability is defined as the probability of failure-free operation of a software system for a specified time in a specified environment.

The key elements of the above definition of reliability are as follows:

- Probability of failure-free operation
- Length of time of failure-free operation
- A given execution environment

Software reliability is expressed as a continuous random variable due to the fact that most large software systems do have some unknown number of faults, and they can fail anytime depending upon their execution pattern. Therefore, it is useful to represent software reliability as a probabilistic quantity. Though software systems do fail from time to time, the users are interested in completing their tasks most of the time. The need for failure-free operation for a certain length of time is important due to the fact that a user intends to complete a task that demands some length of execution time. For example, consider an inventory management software system used by a large store. Assume that the store opens from 8 AM to 8 PM, and they shut down the system after store hours. The store owner expects the inventory system to be up and running for at least 12 hours without developing any failure.

One can specify the reliability of a software system running on a personal computer (PC) as follows. Assume that an office secretary turns on his or her PC every morning at 8.30 AM and turns it off at 4:30 PM before leaving for home. The secretary expects that the PC will run for eight hours without developing any failure. If he or she comes to the office for 200 days in a year and observes that the PC crashes five times on different days in a year for a few years, we can say that the probability of failure-free operation of the PC for eight hours was 0.975 $[=(200-5)/200]$.

The third element in the definition of software reliability is an *execution environment*. An execution environment refers to how a user operates a software system. Not all users operate a software system in the same way. As an example, let us consider the case of a word processor. One group of users may be processing documents of small size, say 50 pages. A second group of users may be processing documents of large size, say, 1000 pages. Clearly, the two groups of users offer two different execution environments to the word processor. For example, to process documents of large size, the software is likely to invoke a part of its code for managing memory space which may not be needed to process a document of small size. The idea of how users operate a system gives rise to the concept of execution environment.

The concept of execution environment is an essential part of the definition of software reliability. Consider that a software system supports 10 different functions $f_1, \ldots, f_{10}$ and there are two groups of users. One group of users use only functions $f_1, \ldots, f_7$, and the second group uses all the functions. Let functions $f_1, \ldots, f_7$ be fault free, but there are faults in functions f_8, f_9, and f_{10}. Consequently, the first group of users will never observe any failure simply because their operations of the software system do not involve the faulty components of the software. From the viewpoint of the first group of users, the probability of failure-free operation of the software is 1.0. On the other hand, the second group of users will observe failure from time to time depending upon how frequently they use functions f_8, f_9, and f_{10}. Consequently, the level of reliability perceived by the second group of users is lower than that perceived by the first group of users. Therefore, the concept of execution environment is crucial to the definition of software reliability. In a later part of this chapter, the concept of operational profile will be discussed to describe execution environment.

15.2.2 Second Definition of Software Reliability

Definition. Failure intensity is a measure of the reliability of a software system operating in a given environment.

According to the second definition, the lower the failure intensity of a software system, the higher is its reliability. To represent the current reliability level of a software system, one simply states the failure intensity of the system. For example, let a software system be in its system testing phase where test engineers are observing failures at the rate of 2 failures per eight hours of system execution.

Then, one can state the current level of the reliability of the system as 0.25 failure per hour. The figure 0.25 failure per hour is obtained by dividing 2 failures by eight hours.

15.2.3 Comparing the Definitions of Software Reliability

The first definition of software reliability emphasizes the importance of a software system operating without failure for a certain minimum length of time to complete a transaction. Here, reliability is a measure of what fraction of the total number of transactions a system is able to complete successfully. Let us assume that an autonomous robot is sent out to explore the underwater sea bed, and it takes three days to complete each round of exploration. In this case, we expect that the on-board software system resident in the robot is expected to run for at least three days continuously so that a round of exploration is successfully completed. If a robot system has successfully completed 99% of all the exploration rounds, we say that the software system's reliability is 0.99.

The second definition of reliability simply requires that there be as few failures as possible. Such an expectation takes the view that the risk of failure at any instant of time is of significant importance. This is very much the case in the operation of an air-traffic control system in an airport. This case is also applicable to the operation of telephone switches. In this definition, it does not matter for how long the system has been operating without failure. Rather, the very occurrence of a failure is of much significance. For example, the failing of a traffic control system in an airport can lead to a major catastrophe.

15.3 FACTORS INFLUENCING SOFTWARE RELIABILITY

A user's perception of the reliability of a software system depends upon two categories of information, namely, (i) the number of faults present in the software and (ii) the ways users operate the system. The second category of information is known as the *operational profile* of the system. The number of faults introduced in a system and the developers' inability to detect many of those faults depend upon several factors as explained below:

- **Size and Complexity of Code:** The number of LOC in a software system is a measure of its size. Large software systems with hundreds of thousands of LOC tend to have more faults than smaller systems. The likelihood of faults in a large system, because of more module interfaces, is higher. The more number of conditional statements the code contains, the more complex the system is considered to be. Due to the economic considerations in software development, one may not have much time to completely understand a large system. Consequently, faults are introduced into the system in all its development phases. Similarly, while removing faults from a large system, new faults may be introduced.

- **Characteristics of Development Process:** Much progress has been made in the past few decades in the field of software engineering. New techniques and tools have been developed to capture system requirements, to design software systems, to implement designs, and to test systems. For example, formal methods, such as SDL (Specification and Description Language) and UML (Unified Modeling Language), are used to specify the requirements of complex, real-time systems. Code review techniques have been developed to detect design and implementation faults. Test tools are available to assist programmers in their unit-level and system-level testing. By developing a system under a larger quality control umbrella in the form of embracing the above software engineering techniques and tools, the number of remaining faults in software systems can be reduced.
- **Education, Experience, and Training of Personnel:** The information technology industry has seen tremendous growth in the past 20 years or so. However, education of software engineering as a separate discipline of study at the undergraduate level is only emerging now on a large scale. It is not unusual to find many personnel with little training to be writing, modifying, and testing code in large projects. Lack of desired skills in personnel can cause a system to have a larger number of faults.
- **Operational Environment:** Detection of faults remaining in a software system depends upon a test engineer's ability to execute the system in its actual operational environment. If a test engineer fails to operate a system in the same manner the users will do, it is very likely that faults will go undetected. Therefore, test engineers must understand the ways the users will operate a system. Because of a lack of sufficient time and lack of experience on the part of development and test engineers, faults can be introduced into a system, and those faults can go undetected during testing.

Therefore, software reliability is determined by a complex combination of a number of factors with wide-ranging characteristics. Ideally, from the viewpoint of modeling, it is useful to have a mathematical model that takes in the influencing parameters and gives us a concrete value of the level of reliability of a software system. However, such an ideal model has not been devised because of the sheer complex nature of the problem. In reality, the reliability models studied in the literature consider a small number of influencing parameters. Despite the limited scope of those models, we gain much insight into the complex concept of software reliability. For example, the more the test engineers execute a system during system-level testing, the more is the likelihood of observing failures. By identifying and fixing the faults causing those failures, development engineers can improve the reliability level of the system. Thus, the amount of testing time directly influences the level of software reliability. Consequently, a reliability model can be developed where *system testing time*, or simply *time*, is the independent variable and level of reliability is the dependent variable in the reliability model. By using such a model, one can predict the length of system testing time required to achieve a given level of software reliability [3].

15.4 APPLICATIONS OF SOFTWARE RELIABILITY

Reliability is a quantitative measure of the failure-related quality aspect of a software system. A number of factors, as explained in Section 15.3, influence the reliability level of software systems. It is natural to evaluate the effectiveness of the influencing factors.

15.4.1 Comparison of Software Engineering Technologies

To develop higher quality software systems in a cost-effective manner, a number of technologies have been introduced. For example, there are numerous software development processes, such as the *waterfall* model [4], the *spiral* model [5], the *prototyping* model [6], the *eXtreme Programming* model [7], and the *Scrum* model [8, 9]. In terms of introduction of testing tools, many techniques have been studied to generate test cases. Several techniques have been introduced to capture customer requirements, such as entity relationship diagrams, data flow diagrams, and the UML [10, 11]. When a new technology is developed, it is necessary to evaluate its effectiveness before fully embracing it. Three useful criteria for evaluating a technology from the management viewpoint are as follows:

- What is the cost of adopting the technology?
- How does the new technology affect development schedule?
- What is the return from the new technology in terms of software quality?

The concept of software reliability can be used to evaluate a new technology in terms of its usefulness in allowing software developers to produce higher quality software. For example, consider two technologies M_1 and M_2. If an application is developed using technology M_1 to generate a system S_1, and technology M_2 is used to generate another system S_2 to implement the same application, it is useful to observe the difference in the reliability levels of S_1 and S_2. By monitoring the reliability levels of the two systems S_1 and S_2 for the same application, managers can observe what technology is more effective in producing software systems of higher reliability.

15.4.2 Measuring the Progress of System Testing

Measuring the progress of software development is central to managing a project. An important question in project management is: *How much has been completed?* An answer to the question lets us know if progress is being made as planned. It is important to monitor the progress of system testing, because it consumes a significant portion of money and time. Two useful metrics to monitor the progress of system-level testing are as follows:

- Percentage of test cases executed,
- Percentage of successful execution of high-priority functional tests

The concept of software reliability can be used to measure the progress of system testing. For example, one can monitor the failure intensity of the SUT to know where on the reliability scale the system lies at this moment. If the current failure intensity is much larger than a tolerable failure intensity at the time of its release, we know that much work remains to be done. On the other hand, if the difference between the current failure intensity and the desired failure intensity is very small, we realize that we are close to reaching the goal. Thus, the concept of software reliability can be used in an objective manner to measure how much progress has been made in system-level testing [12, 13].

15.4.3 Controlling the System in Operation

The reliability of a system usually reduces as a result of maintenance works. The larger the amount of change made to a system, the larger is the reduction in the reliability level of the system. For example, assume that a system has k number of faults per 1000 lines of code at the beginning of system testing. The value of k can be obtained from statistical measures. By adding N lines of code to the system as a part of a maintenance activity, statistically, $Nk/1000$ faults are introduced into the system. Introduction of the new faults will decrease the reliability of the system, and it will require prolonged system testing to detect the faults and raise system reliability by fixing those faults. A project manager may put a limit on the amount of change in system reliability for each kind of maintenance activity. Therefore, the size of a maintenance work can be determined by the amount of system reliability that can be sacrificed for a while.

15.4.4 Better Insight into Software Development Process

The concept of reliability allows us to quantify the failure-related quality aspect of a software system. Quantification of the quality aspect of software systems gives developers and managers a better insight into the process of software development. For example, by observing the failure intensity at the beginning of system testing, a test manager may be able to make an informed decision regarding how long system testing may take to bring down the failure intensity to an acceptable level. In other words, managers will be capable of making informed decisions.

15.5 OPERATIONAL PROFILES

The notion of *operational profiles*, or *usage profiles*, was developed at AT&T Bell Laboratories [14] and IBM [15, 16] independently. As the name suggests, an operational profile describes how actual users operate a system. An operational profile is a quantitative characterization of how a system will be used. However, for accurate estimation of the reliability of a system one must test it by considering how it will actually be used in the field.

15.5.1 Operation

An *operation* is a major system-level logical task of short duration which returns control to the initiator when the operation is complete and whose processing is substantially different from other operations. In the following, the key characteristics of an operation are explained:

- *Major* means that an operation is related to a functional requirement or feature of a software system.
- An operation is a *logical* concept in the sense that it involves software, hardware, and user actions. Different actions may exist as different segments of processing. The different processing time segments can be contiguous, noncontiguous, sequential, or concurrent.
- *Short duration* means that a software system is handling hundreds of operations per hour.
- *Substantially different processing* means that an operation is considered to be an entity in the form of some lines of source code, and there is a high probability that such an entity contains a fault not found in another entity.

15.5.2 Representation of Operational Profile

The operational profile of a system is the set of operations supported by the system and their probabilities of occurrence. For example, there are three operations, namely, A, B, and C, supported by a system, and they occur 50, 30, and 2% of the time. Then the operational profile is represented by the set $\{(A, 0.5), (B, 0.3), (C, 0.2)\}$. Operational profiles are represented in two alternative ways:

- *Tabular* representation
- *Graphical* representation

Tabular Representation of Operational Profile An operational profile can be represented in a tabular form with three columns as shown in Table 15.1. The example is about a library information system. The first column lists the names of the operations, the second column gives the frequency of using the operations, and the final column shows the probability of using the operations. We list two kinds of book returned operations: books returned in time and books returned late. The two operations involve some separate processing because a book returned late may incur some penalty. The book renewed operation is a combination of one of the two book returned operations and a book checked out operation.

Graphical Representation of Operational Profile An operational profile can also be represented in graphical form as shown in Figure 15.2. In a graphical form, an operational profile is represented as a tree structure consisting of nodes and branches. Nodes represent attributes of operations, and branches represent values of attributes with the associated probability of occurrence. In Figure 15.2, four

TABLE 15.1 Example of Operational Profile of Library Information System

Operation	Operations per Hour	Probability
Book checked out	450	0.45
Book returned in time	324	0.324
Book renewed	81	0.081
Book returned late	36	0.036
Book reported lost	9	0.009
⋮	⋮	⋮
Total	1000	1.0

Figure 15.2 Graphical representation of operational profile of library information system.

nodes, namely 1, 2, 3, and 4 are shown. Node 1 represents the scope attribute of operations with two values, administration and user service. The scope of an operation refers to whether an operation is for the administration of the information system or for providing services to users. The probability of occurrence of the administration value of the scope attribute of an operation is 0.1, and the probability of occurrence of the user service value is 0.9. Node 2 represents an administration operation, whereas node 3 represents a user operation. There are two attributes of the administration operation, namely, account management and reporting, and the associated probabilities of occurrence are 0.4 and 0.6, respectively. There are four attributes of the user operations, namely, check out, renewal, loss, and return, with their associated probabilities of occurrence being 0.5, 0.09,

0.01, and 0.4, respectively. Node 4 represents a return operation with two attribute values, namely delayed and in-time, with their probabilities of occurrence being 0.1 and 0.9, respectively.

It is useful to note that a tabular form of an operational profile can be easily generated from a graphical form by considering all possible paths in the graphical form and multiplying the probabilities appearing on each path. For example, the probability of the book returned late operation (0.036) in Table 15.1 can be obtained from the graphical form shown in Figure 15.2 by multiplying the probabilities associated with user service (0.9), return (0.4), and delayed (0.1).

Choosing Tabular Form or Graphical Form The tabular form of an operational profile can be easily obtained from the graphical form. It is easy to specify the operational profile in a tabular form if a system involves only a small number of operations. However, the graphical form is preferred if a system involves a large number of operations. Operations that can be easily described as sequences of smaller processing steps are better suited for the graphical form. Moreover, it is easy to identify missing operations in a graphical form.

Using Operational Profile During System Testing The notion of operational profiles was created to guide test engineers in selecting test cases. Ideally, at least one test case should be selected to test an operation, and all the operations should be covered during testing. Since software reliability is very much tied with the concept of failure intensity, a software system with better reliability can be produced within a given amount of time by testing the more frequently used operations first. In reality, projects overrun their schedules, and there may be a tendency to deliver products without adequate testing. An operational profile can be used in making a decision concerning how much to test and what portions of a software system should receive more attention. The ways test engineers select test cases to operate a system may significantly differ from the ways actual users operate a system. However, for accurate estimation of the reliability of a system, test the system in the same way it will actually be used in the field.

Other Uses of Operational Profiles The operational profile of a system can be used in a number of ways throughout the life-cycle model of a software system as follows:

- As a guiding document in designing user interface. The more frequently used operations should be easy to learn and easy to use.
- In developing a version of a system for early release. The early-release version can contain the more frequently used operations.
- To determine where to put more resources for system development. For example, more resources can be allocated to the development of those operations which are used more frequently.
- As a guiding document for organizing user manuals. For example, the more frequently used operations are described earlier than the rest.

15.6 RELIABILITY MODELS

The reliability models are developed based on the following assumptions:

- Faults in the program are independent.
- Execution time between failures is large with respect to instruction execution time.
- Potential *test space* covers its *use space*.
- The set of inputs per test run is randomly selected.
- All failures are observed.
- The fault causing a failure is immediately fixed or else its reoccurrence is not counted again.

To understand the idea of fault independence in the first assumption, we can consider a mapping between the set of faults in a system and the set of failures caused by the faults. Faults are said to be independent if there is a one-to-one or one-to-many relationship between faults and failures. Thus, if faults are independent and a fault is fixed, then the corresponding single or multiple failures are no longer observed. In a many-to-one relationship, fixing one fault will not eliminate the corresponding failure; all the faults need to be fixed to eliminate the failure.

The second assumption concerning the length of execution time between failures tells us that the system does not fail too often. A reasonably stable system is a prerequisite for the reliability models to be valid. No meaningful prediction can be done if a system fails too often.

The third assumption, concerning test space and use space, implies that a system be tested by keeping in mind how it will be used. For example, consider a telephone system comprising three features: (i) call processing, (ii) billing, and (iii) administrative operation. Assume that there are 15 basic operations to support each feature. Even though there are a total of 45 operations, actual operation of the system can produce a large number of execution scenarios. Call establishment between two phones is a basic operation in the call processing group, and updating a customer profile is a basic operation in the administrative group. Those two operations can be tested separately. However, there may be a need to update a customer's profile while the customer is in the middle of a call. Testing such an operational situation is different from individually testing the two basic operations. The usefulness of a reliability model depends on whether or not a system is tested by considering the use space of the system that can be characterized by an operational profile.

The fourth assumption emphasizes the need to select test input randomly. For example, referring to the telephone system in the explanation of the third assumption, to test the call establishment function, we randomly select the destination number. Randomness in the selection process reduces any bias in favor of certain groups of test data.

The fifth assumption concerning failures simply tells us to consider only the final discrepancies between actual system behavior and expected system behavior.

Even if a system is in an erroneous state, there may not be a system failure if the system has been designed to tolerate faults. Therefore, only the observed failures are taken into consideration, rather than the possibility of failures.

The sixth assumption tells us how to count failures. When a failure is observed, it is assumed that the corresponding fault is detected and fixed. Because of the first assumption that faults are independent, the same failure is not observed due to another, unknown fault. Thus, we count a failure once whether or not the corresponding fault is immediately fixed.

The details of the development of two mathematical models of reliability are presented. In these two models failure intensity as a measure of reliability is expressed as a function of execution time. That is, an expression for $\lambda(\tau)$ is developed. The models are developed based on the following intuitive idea: *As we observe another system failure and the corresponding fault is fixed, there will be a fewer number of faults remaining in the system and the failure intensity of the system will be smaller with each fault fixed. In other words, as the cumulative failure count increases, the failure intensity decreases*.

In the above intuitive idea, an important concept is the characterization of the *decrement* in failure intensity as a function of cumulative failure count. In this chapter, two reliability models are developed by considering two decrement processes as follows:

Decrement Process 1: The decrease in failure intensity after observing a failure and fixing the corresponding fault is constant. The reliability model developed using this model of failure intensity decrement is called the basic model.

Decrement Process 2: The decrease in failure intensity after observing a failure and fixing the corresponding fault is smaller than the previous decrease. In other words, fixing a fault leading to an earlier failure causes the failure intensity to be reduced by a larger amount than fixing a fault causing a later failure. Therefore, failure intensity is exponential in the number of failures observed. The reliability model developed using this model of failure intensity decrement is called the logarithmic model.

In the development of the two reliability models, the following notation is used:

- μ denotes the mean number of failures observed.
- λ denotes the mean failure intensity.
- λ_0 denotes the *initial* failure intensity observed at the beginning of system-level testing.
- ν_0 denotes the total number of system failures that we expect to observe over infinite time starting from the beginning of system-level testing.
- θ denotes the decrease in failure intensity in the logarithmic model. This term will be further explained in the discussion of logarithmic model below.

Basic Model The constant decrement in failure intensity per failure observed is illustrated in Figure 15.3, where the straight line represents the failure decrement

Figure 15.3 Failure intensity λ as function of cumulative failure μ ($\lambda_0 = 9$ failures per unit time, $\nu_0 = 500$ failures, $\theta = 0.0075$).

process in the basic model. Initially, the observed failure intensity is 9 failures per unit time. The total number of failures to be observed over infinite time is assumed to be 500. When all the 500 failures have been observed and the corresponding faults have been fixed, no more system failure will be observed. Therefore, the failure intensity becomes zero at the point we have observed the final failure. The rate of decrement in failure intensity is represented by the slope of the straight line and is equal to $-\lambda_0/\nu_0 = -9/500$ per unit time. The straight line in Figure 15.3 can be expressed as follows:

$$\lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{\nu_0}\right)$$

Since both λ and μ are functions of τ and $\lambda(\tau)$ is the derivative of $\mu(\tau)$, we have

$$\frac{d\mu(\tau)}{d\tau} = \lambda_0 \left(1 - \frac{\mu(\tau)}{\nu_0}\right)$$

By solving the above differential equation, we have

$$\mu(\tau) = \nu_0(1 - e^{-\lambda_0 \tau/\nu_0})$$

and

$$\lambda(\tau) = \lambda_0 e^{-\lambda_0 \tau/\nu_0}$$

Logarithmic Model The curved line in Figure 15.3 illustrates the process of decrement in failure intensity per failure observed in the logarithmic model. Whichever reliability model chosen, the observed failure intensity remains the same. Therefore, in Figure 15.3, the initially observed failure intensity is shown to be the same, that is, 9 failures per unit time, as in the basic model. The total number of failures to be observed over infinite time is infinite. Therefore, the failure intensity never reaches zero. The relationship between failure intensity and cumulative failure count shows that faults fixed in the beginning cause a larger decrement in failure intensity than faults fixed at a later stage. This view of decrement in failure intensity is consistent with real-life systems. For example, those faults which cause a system to fail in many ways are likely to manifest in earlier failures than those faults which manifest themselves in fewer failures. Therefore, by fixing a fault early, one observes a larger drop in failure intensity than caused by fixing a fault at a later stage.

The nonlinear drop in failure intensity in the logarithmic model is captured by a decay parameter θ associated with a negative exponential function as shown in the following relationship between λ and μ:

$$\lambda(\mu) = \lambda_0 e^{-\theta\mu}$$

Since both λ and μ are functions of τ and $\lambda(\tau)$ is the derivative of $\mu(\tau)$, we can write

$$\frac{d\mu(\tau)}{d\tau} = \lambda_0 e^{-\theta\mu(\tau)}$$

By solving the above differential equation, we obtain

$$\mu(\tau) = \frac{\ln(\lambda_0\theta\tau + 1)}{\theta} \qquad \lambda(\tau) = \frac{\lambda_0}{(\lambda_0\theta\tau + 1)}$$

Comparison of Reliability Models In Figure 15.4, we have plotted the expressions for $\lambda(\tau)$ in both the models and in Figure 15.5 the expressions for $\mu(\tau)$. We have assumed that the initial failure intensities are $\lambda_0 = 9$ failures per unit time in both models, the total number of failures to be observed over infinite time in the basic model is $\nu_0 = 500$ failures, and the failure intensity decay parameter $\theta = 0.0075$ in the logarithmic model.

Therefore, if we choose one of the two models to apply to a real-life software system, the next task is to estimate the model's parameters. This can be done by noting the time instants $\tau_1, \ldots, \tau_k$ at which the first k failures $\mu(\tau_1), \ldots, \mu(\tau_k)$, respectively, occur. This gives us k data points $(\tau_1, \mu(\tau_1)), \ldots, (\tau_k, \mu(\tau_k))$. We may then determine the chosen model's parameters so that the resulting plot fits the set of actually observed points. We can use a least squares technique, for example, for this purpose.

Example Assume that a software system is undergoing system-level testing. The initial failure intensity of the system was 25 failures per CPU hour, and the current

Figure 15.4 Failure intensity λ as function of execution time τ ($\lambda_0 = 9$ failures per unit time, $\nu_0 = 500$ failures, $\theta = 0.0075$).

Figure 15.5 Cumulative failure μ as function of execution time τ ($\lambda_0 = 9$ failures per unit time, $\nu_0 = 500$ failures, $\theta = 0.0075$).

failure intensity is 5 failures per CPU hour. It has been decided by the project manager that the system will be released only after the system reaches a reliability level of at most 0.001 failure per CPU hour. From experience the management team estimates that the system will experience a total of 1200 failures over infinite time. Calculate the additional length of system testing required before the system can be released.

First, an appropriate reliability model for the software system is chosen. Because of the assumption that the system will experience a total of 1200 failures over infinite time, we use the basic model.

Let us denote the current failure intensity and the desired failure intensity at the time of release as λ_c and λ_r, respectively. Assume that the current failure intensity has been achieved after executing the system for τ_c hours. Let the release time failure intensity λ_r be achieved after testing the system for a total of τ_r hours. We can write λ_c and λ_r as

$$\lambda_c = \lambda_0 e^{-\lambda_0 \tau_c / \nu_0} \qquad \lambda_r = \lambda_0 e^{-\lambda_0 \tau_r / \nu_0}$$

The quantity $\lambda_r - \lambda_c$ denotes the additional amount of system testing time needed to achieve reliability λ_r at the time of release. The quantity $\lambda_r - \lambda_c$ can be represented as follows:

$$\frac{\lambda_c}{\lambda_r} = \begin{cases} \dfrac{\lambda_0 e^{-\lambda_0 \tau_c / \nu_0}}{\lambda_0 e^{-\lambda_0 \tau_r / \nu_0}} \\ e^{(-\lambda_0 \tau_c / \nu_0) + (\lambda_0 \tau_r / \nu_0)} \\ e^{(\tau_r - \tau_c)\lambda_0 / \nu_0} \end{cases}$$

or

$$\ln\left(\frac{\lambda_c}{\lambda_r}\right) = \frac{(\tau_r - \tau_c)\lambda_0}{\nu_0}$$

or

$$\tau_r - \tau_c = \begin{cases} \left(\dfrac{\nu_0}{\lambda_0}\right) \ln\left(\dfrac{\lambda_c}{\lambda_r}\right) \\ \left(\dfrac{1200}{25}\right) \ln\left(\dfrac{5}{0.001}\right) \\ 408.825 \text{ hours} \end{cases}$$

Therefore, it is required to test the system for more time so that the CPU runs for another 408.825 hours to achieve the reliability level of 0.001 failure per hour.

15.7 SUMMARY

This chapter began with an introduction to the following concepts: (i) fault and failure, (ii) execution and calendar time, (iii) time interval between failures, (iv) failures in periodic intervals, and (v) failure intensity. With these concepts

in place, software reliability was defined in two ways: (i) the probability of failure-free operation of a software system for a specified time in a specified environment and (ii) failure intensity as a measure of software reliability.

Next, we discussed user's perception of software reliability, which depends upon two factors: (i) the number of faults present in the software and (ii) how the user operates the system, that is, operational profile of the system. The number of faults introduced into the system and the developer's inability to detect many of those faults are discussed, which include (i) size and complexity of code, (ii) characteristics of development process, (iii) education, experience, and training of personnel, and (iii) operational environment. In summary, software reliability is determined by a complex combination of a number of factors with wide-ranging characteristics. After that, we explained various applications of software reliability.

Then we presented the notion of operational profiles developed by Musa at Bell Laboratories. An operational profile is a quantitative characterization of how actual users operate a system. Operational profiles are represented in two alternative ways: tabular form and graphical form. Operational profiles can be used to guide the design of a user interface, the determination of where to put more resources for system development, the selection of test cases, and the development of user manuals.

Finally, two mathematical models of reliability were explained: the basic model and the logarithmic model. In these two models, failure intensity as a measure of reliability is expressed as a function of executed time.

LITERATURE REVIEW

Measuring the reliability of large software systems presents both technical and management challenges. Large software systems involve a large number of simultaneous users, say, several thousands, and most of them may be located in different geographic locations. Some examples of large software systems are software controlling the operation of cellular phone networks, on-line purchase systems, and banking systems, to name just a few. The books by Musa, Iannino, and Okumoto [7] and Musa [8] address a number of issues concerning software reliability: The technical issues addressed by Musa, Iannino, and Okumoto are as follows:

- **Parameter Determination:** Five methods of parameter determination, namely *prediction*, *estimation*, *identification*, *formula and/or experience*, and *data*, have been discussed.
- **Project-Specific Techniques:** Techniques for dealing with different special problems have been discussed. For example, they explain ways to estimate the execution time of occurrence of a failure. Another project-specific technique concerns measuring failure times in multiple installations.

The concept of operational profile has been presented in great detail by Musa (*Software Reliability Engineering*, McGraw-Hill, New York, 1999). Specifically,

the main activities related to preparing an operational profile have been discussed. Moreover, the handling of the evolution of the definition of operation during system development has been discussed.

The management aspects of software reliability have been discussed in the books by both Musa, Iannino, and Okumoto [7] and Musa [8]. How to plan for implementing a reliability program has been discussed by Musa, Iannino, and Okumoto [7]. Specifically, the following activities have been explained to run a reliability program:

- **Data Collection:** Data collection for reliability analysis involves activities such as what data to gather, how to motivate data takers, how to make data collection easy, collecting data in real time, feeding results back, and recordkeeping.
- **Use of Consultants:** Consultants play an important role in technology transfer concerning reliability engineering. A consultant can be a professional external consultant or an individual inside the organization. By virtue of their expertise, consultants have a valuable impact throughout an organization.

Musa, Iannino, and Okumoto [17] present a few more models, such as Poisson-type models and binomial-type models. Lyu [19] presents several other reliability models and actual failure data from field operations of large software systems.

Another prominent method similar to reliability testing is known as *statistical testing*, which uses a formal experimental paradigm for random testing according to a usage model of the software. In statistical testing, a model is developed to characterize the population of uses of the software, and the model is used to generate a statistical correct sample of all possible uses of the software. Performance on the sample is used as a basis for conclusions about general operational reliability. Interested readers are referred to the following book and articles for discussion on this topic:

N. Baskiotis, M. Sebag, M.-C. Gaudel, and S. Gouraud, "A Machine Learning Approach for Statistical Software Testing," in *Proceedings, International Conference on Artificial Intelligence*, Hyderabad, India, Morgan Kaufman, San Francisco, January 6–12, 2007, pp. 2274–2278.

S. Prowell, C. Trammell, R. Linger, and J. Poore, *Cleanroom Software Engineering*, Addison-Wesley, Reading, MA, 1999.

G. H. Walton, J. H. Poore, and C. J. Trammell, "Statistical Testing of Software Based on a Usage Model," *Software Practice and Experience*, Vol. 25, No. 1, January 1995, pp. 97–108.

J. A. Whittaker and M. G. Thomason, "A Markov Chain Model for Statistical Software Testing," *IEEE Transactions on Software Engineering*, Vol. 20, No. 10, October 1994, pp. 812–824.

REFERENCES

1. W. K. Ehrlich, A. Iannino, B. S. Prasanna, J. P. Stampfel, and J. R. Wu. How Faults Cause Software Failures: Implications for Software Reliability Engineering. In *Proceedings of the International Symposium on Software Reliability Engineering*, Austin, TX, May 1991, IEEE Computer Society Press, Piscataway, New Jersey, 1991, pp. 233–241.
2. K. Trivedi. *Probability and Statistics with Relaibility, Queuing, and Computer Science Applications*. Wiley, Hoboken, NJ, 2002.
3. H. Pham. *System Software Reliability*. Springer, 2006.
4. W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of WESCON*, August 1970, pp. 1-9. Republished in ICSE, Monterey, CA, 1987, pp. 328-338.
5. B. W. Boehm. A Spiral Model for Software Development and Enhancement. *IEEE Computer*, May 1988, pp. 61–72.
6. I. Sommerville. *Software Engineering*. Addison-Wesley, Reading, MA, 2006.
7. K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley, Reading, MA, 2004.
8. K. Schwaber. *Agile Project Management with Scrum*. Microsoft Press, Redmond, WA, 2004.
9. H. Takeuchi and I. Nonaka. The New Product Development Game. *Harvard Business Review*, Harvard Business School Publishing, Boston, MA, January-February 1986, pp. 1-11.
10. M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Reading, MA, 2003.
11. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Longman, Reading, MA, 1998.
12. S. Dalal and C. Mallows. When Should One Stop Testing Software. *Journal of the American Statistical Associations*, Vol. 81, 1988, pp. 872–879.
13. M. C. K. Yang and A. Chao. Reliability-Estimation and Stopping-Rules for Software Testing, Based on Repeated Appearances of Bugs. *IEEE Transactions on Reliability*, June 1995, pp. 315–321.
14. J. D. Musa. Software Reliability Engineering. *IEEE Software*, March 1993, pp. 14–32.
15. R. Cobb and H. D. Mills. Engineering Software under Statistical Quality Control. *IEEE Software*, November 1990, pp. 44–54.
16. H. D. Mills, M. Dyer, and R. C. Linger. Cleanroom Software Engineering. *IEEE Software*, September 1987, pp. 19–24.
17. J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability*. McGraw-Hill, New York, 1987.
18. J. D. Musa. *Software Reliability Engineering*. McGraw-Hill, New York, 1999.
19. M. R. Lyu. *Handbook of Software Reliability Engineering*. McGraw-Hill, New York, 1995.

Exercises

1. What definition of software reliability is best suited for each of the following kinds of software systems? Justify your answers.
   (a) Operating system of computer for personal use
   (b) Operating system of computer used to control electric power plant
   (c) Spell checking system
   (d) Embedded software controlling washing machine
   (e) Video game software
   (f) Communication software system running on cellular phone
   (g) Communication software system controlling base stations of cellular phone system

2. Assume that a software system will experience 150 failures in infinite time. The system has now experienced 60 failures so far. The initial failure intensity at the beginning of system testing was 15 failures per CPU hour. What is the current failure intensity?
3. Assume that a software system is undergoing system-level tests and the initial failure intensity is 15 failures per CPU hour. The failure intensity decay parameter has been found to be 0.025 per failure. So far the test engineers have observed 60 failures. What is the current failure intensity?
4. Explain the consequence of not satisfying the assumption that potential *test space* covers its *use space* in developing the two reliability models.
5. Explain how the parameters of the basic model and the logarithmic model can be determined.
6. Explain how the operational profile concept can be applied to regression tests.
7. Explain how the idea of software reliability can uncover missing requirements.
8. Explain the relationship between the *correctness* and the *reliability* quality attributes of a software system.
9. Can an incorrect, that is, faulty, software system be considered to be reliable? Justify your answer.
10. Explain why it may not be possible to satisfy some of the assumptions used in the two models of software reliability.

CHAPTER 16

Test Team Organization

In the end, all business operations can be reduced to three words: people, product, and profits. Unless you've got a good team, you can't do much with the other two.
— *Lee Iacocca*

16.1 TEST GROUPS

There are many ways in which test teams can be organized. There are no right or wrong ways to organize test teams. The structure one chooses will affect productivity, quality, customer satisfaction, employee morale, and budget. As we know, testing is a distributed activity, distributed in time and space, conducted in different phases of a software development life cycle. The different, broad kinds of testing are unit testing, integration testing, system testing, and acceptance testing. It is logical to have different testing groups in an organization for each testing phase of a software development life cycle. However, it is recommended that the unit tests be developed and executed by the software developers themselves, rather than an independent group of unit test engineers. The programmer who develops a software unit should take the ownership of producing good-quality software to his or her satisfaction. Therefore, instead of forming a unit test group, software developers should be given the responsibilities of testing their own code. The acceptance test group is formed on a demand basis consisting of people from different backgrounds, such as integration test engineers, system test engineers, customer support engineers, and marketing engineers. The test group is dismantled after the project is completed. In any organization, it is recommended to have at least two test groups: integration test group and system test group. In the following sections, their details are discussed.

16.1.1 Integration Test Group

System integration testing, or simply, integration testing, is performed by an integration test group. Integration testing requires a thorough knowledge of the modules

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

and their interfaces. Since large systems are integrated in an incremental manner, possibly by using the concept of incremental builds, the integration test personnel must be very conversant with the idea of builds. Thus, it natural that software developers, who together built the modules, must be involved in performing integration testing. In practice, the developers themselves may integrate the system. The system architects are also involved in integration testing for complex systems because of their thorough understanding of the larger picture of the system. The mandate of this group is to ensure that unit-tested modules operate correctly when they are combined. The modules are combined by following the software structure identified during system design. System integration is performed as a part of the larger concept of system development. The leader of the integration test group reports to the software development manager. In addition to integrating the system, the test group may perform other duties, such as code inspection, configuration management, release management, and management of development laboratory. The activities pertaining to this group are described in Chapter 7.

16.1.2 System Test Group

A system test group is a team whose primary job is to perform testing without any bias and fear. The system test group is truly an independent group, and it usually has a separate head count and budget. Separate head count means that people are specifically hired and retained to perform testing. The reader may recall that some development engineers take up the role of an integration test engineer. However, to eliminate the possibility of any bias in system-level testing, the task is performed by a separate group, and not by some members of the development group. The manager of this group does not report to hardware or software development managers. Rather, they are all peers at the organizational level. The mandate of this group is to ensure that the system requirements have been satisfied and that the system is acceptable. The system test group conducts different categories of tests as discussed in Chapter 8. To facilitate the system to be eventually accepted by the customer, the group executes *business acceptance tests* identified in the user acceptance test plan.

The system test group can be further divided into several focused subgroups as the size of an organization becomes large: *development test group*, *performance test group*, *scalability test group*, *automation test group*, and *sustaining test group*, as shown in Figure 16.1. Each of the groups may be headed by a line manager, who in turn reports to the senior-level manager in charge of the system test group. The system test group structure must be regularly reviewed and adjusted to meet shifting internal and external needs. Note that the structure of the system test group is organized on a functional basis, and not on a project basis. A long-lasting organization of a test group will lead to better development of its member skills.

In order to form a test group, a crucial thing is the *size* of the system test group. This essentially tells us the number of people should be performing system-level testing. The size of the system test group impacts the delivered quality of the product, the development cost of the product, and the time to deliver the product. The *tester–developer* ratio is a matter of considerable debate. In practice, $\frac{1}{2}$, $\frac{2}{3}$, and $\frac{1}{4}$ are common values of the tester–developer ratio [1]. An appropriate

Figure 16.1 Structure of test groups.

value of the ratio depends upon the nature of the software under development. A value of the ratio is estimated during the development of a test planning document, and it has been discussed in Chapter 12.

Development Test Group The focus of this group is on the testing of new features in a particular release. This includes *basic tests, functionality tests, robustness tests, interoperability tests, stress tests, load and stability tests, regression tests, documentation tests*, and *business acceptance tests*.

Performance Test Group This group puts emphasis on system performance. Tests are conducted to identify system bottlenecks and recommendations are made to the developers for improving system performance. The group uses test, measurement, and analysis tools to carry out its tasks. This group may take up additional responsibilities such as reliability testing.

Scalability Test Group The focus of this group is on determining whether or not the system can scale up to its engineering limits. For example, a cellular phone network might have been designed with certain engineering limits in mind, such as the maximum number of base stations it can support, the maximum number of simultaneous calls it can handle, and so on. The group tests whether the designed system can reach those limits. This group may take up additional responsibilities such as load and stability testing.

Automation Test Group The responsibility of this group is to develop test automation infrastructure, test libraries, and tests tools. This group assists other groups in the development of automated test suites.

Sustaining Test Group This group maintains the software quality throughout the product's market life. This team is responsible for maintaining the *corrective* aspect of software maintenance. The group works very closely with customers and conducts regression testing of patch software.

16.2 SOFTWARE QUALITY ASSURANCE GROUP

Software testing is a part of the larger concept of software quality assurance. Testing encompasses the processes of finding defects and fixing them. Defects can be found through a combination of code inspection and actual execution of test cases. On the other hand, software quality assurance deals not only with the location of the defects but also with mechanisms to prevent defects. In the IEEE standard 610-12-1990 [2] quality assurance is defined as follows:

1. A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements
2. A set of activities designed to evaluate the process by which products are developed or manufactured

The first item above deals with testing which contributes to the quality of a product. The second item is procedural in nature; it includes actively ensuring the continuous improvement of product quality. This implies that a software quality assurance group has a larger role in ensuring *conformance to the best development practices* throughout the organization. Such a role is well beyond the scope of a test group. The software quality assurance group should have sufficient authority, power, and standing to work with the entire organization in defining processes and dictating peers to follow the processes [3]. The software quality assurance team should have sufficient members to do a thorough and professional job at system testing as well as quality management work. It is recommended to have a separate group for quality management work, as shown in Figure 16.2, rather than assign quality management task to system test engineers.

Quality Management Group This group works on customizing software development processes and ensuring that processes are adhered to. The group is responsible for creating and implementing a quality management program plan for the entire organization by following a standard framework, such as the ISO9000:2000 quality model. The group proactively works to drive process improvement initiatives across the organization. The group makes an effort to understand the best practices followed around the world through systematic benchmarking and to amalgamate those practices with the one existing within the organization. The team can take up additional responsibilities to implement

Figure 16.2 Structure of software quality assurance group.

automated collection, tracking, and reporting of metrics relevant to product development, testing, and project management.

Quality control is another term that is often used in the literature. Quality control is defined in the IEEE standard 610 [2] as *a set of activities designed to evaluate the quality of developed or manufactured products*. The term is used in a production or hardware manufacturing environment, where a large number of physical items are produced and shipped. Each of the items has to go through a testing process to ensure that the quality of the product is good enough for shipment; otherwise, the item is rejected. The quality check is conducted by the quality control group within the manufacturing organization, and the person who conducts the testing is called a quality controller.

16.3 SYSTEM TEST TEAM HIERARCHY

Five kinds of subgroups within the system test group of an organization have been identified in Section 16.1.2. Each subgroup is organized in a hierarchical fashion headed by a test manager. The size of a test subgroup can grow to a maximum of 12 test engineers. A subgroup with more than 12 members may be divided into two subgroups for better management. Members of a subgroup include *technical leaders, principal engineers, senior engineers*, and *junior engineers* as depicted in Figure 16.3.

The duties of the team members are different from subgroup to subgroup. The role and responsibilities of each team member must be clearly defined. In the following, we give a brief description of the role and responsibilities for each kind of team member that are common to most organizations:

- **Test Managers:** The manager of a test subgroup is a key person in charge of all aspects of the assigned testing task. The manager leads the group by working with other test subgroups to perform testing functions to ensure that high-quality products are released. The administrative responsibilities

Figure 16.3 System test team hierarchy.

of a manager include managing the budget, hiring personnel, assigning tasks, training personnel, developing their career, and reviewing their performance. The technical responsibilities of a test manager include (i) developing a test plan, executing a test plan, and handling crises; (ii) developing, following, and improving test processes; and (iii) defining, collecting, and analyzing metrics. The manager is proactive in improving people, productivity, and processes. The manager is expected to be able to work with other test groups within the company as well as the customer test organization. A particular test group may be located in different places and even different countries. The test manager acts as a nodal point between upper management, project management, quality management, and the marketing people.

- **Junior Engineers:** New, inexperienced test engineers are put at a junior level. They gain experience by assisting the senior and principal engineers in charge of test execution, setting up of test beds, and developing scripts for test automation. They may also be asked to test the user manual, the on-line help feature, and the graphical user interface.
- **Senior Engineers:** The senior engineers design, develop, and execute test cases. They design and set up test laboratories and environments. Senior engineers assist software developers in reproducing known defects. They participate in test plan review meetings and support maintenance of test laboratories, automated test suites, and test tools. Test tools include a defect tracking system and a test factory.
- **Principal Engineers:** The principal engineers are test specialists responsible for test planning, test automation, test environment planning, test tool development, procurement of test equipment, performance modeling, reliability engineering, and business acceptance testing. In addition, the principal engineers execute test cases and mentor junior engineers within the group.
- **Technical Leaders:** A technical leader often needs to coordinate the tasks of many engineers working on complex projects. At this level the engineer should have technical testing skills of a principal engineer as well as negotiation skills, process and project management skills, and communication skills. The technical leader provides the strategic direction for software testing, assists test managers in collecting test metrics, coordinates test plan review meetings, attends entry/exit criteria review meetings, and participates in cross-functional review meetings such as for requirements review, functional specification review, and entry/exit criteria selection.

16.4 EFFECTIVE STAFFING OF TEST ENGINEERS

Designing test cases to reveal defects in a complex system is a challenging task. One may not get as much satisfaction from designing test cases as from designing a system. However, designing test cases to reveal defects in a system is definitely

an ingenuous task. Top-notch test engineering talent is not only desirable but also essential to making high-quality products. A poor-quality test engineer can do much damage in terms of reporting false positives and false negatives. A hiring process should focus on never letting in a bad fit, even if that means accidentally rejecting some good fits. One must remember the saying, *you can only be as good as your people*. A leader must set a personal standard to staff the team with the best test engineers, developing them fully, and letting them take advantage of opportunities.

A team leader must have a clear picture of the experience and skills required for a test position. It is useful to keep in mind the following five-C characteristics identified by Bill Hetzel [4] that a good test engineer must possess.

1. *Controlled*: A test engineer must be organized, disciplined, and methodical in his or her work.
2. *Comprehensive*: A test engineer must be very attentive to the details [5].
3. *Considerate*: A test engineer must have good interpersonal skills such as the ability to handle aggressive behavior, not to be easily offended, to be malleable, and, finally, to bring situations to a win–win closure.
4. *Critical*: A test engineer must be very good in analysis and assertiveness. Assertiveness typically consists of being persistent, using multiple methods of reflective listening to determine exactly what is being communicated.
5. *Competent*: A test engineer must be aware of test techniques, tools, technology, and domain knowledge that can be used in order to carry out the job efficiently.

In addition to the five-C characteristics, test engineers are expected to have the following skills [6] as a part of their competence to become successful test specialists:

- Have credibility with software developers
- Understand developers' terminologies
- Know when the code is ready for testing
- Be able to evaluate the impact of a problem on the customers
- Assist software developers in expediting defect resolution
- Reduce false-positive and false-negative results in testing
- Develop expertise in test automation
- Mentor the junior test engineers

A successful test team is made up of members whose strengths are complementary. An effort must be made to avoid hiring test engineers who are mirror images of existing staff or yourself. Otherwise, the team will be constrained with limited skills and perspectives. The team should be well balanced with people who have a variety of skills. There should be a good cross section of experience and technical abilities. Therefore, it is advisable to have people on the test team with

diverse background and experience, such as developers, integration testers, information technology administrators, technical support personnel, technical writers, quality management personnel, experienced test engineers, and recent graduates. The roles of people with such diverse backgrounds are explained in the following.

- **Developers:** A developer's expertise in system design, coding, and unit testing is very useful in becoming an effective test engineers. To effectively communicate with software developers, test engineers must use their vocabulary, such as object, class, inheritance, polymorphism, variables, pointers, structures, and so on, to have credibility with the software development group. Code development experience is useful when a test engineer participates in code review. Another reason for test engineers to have development experience is to be able to automate the tests without much problem. A key criterion in hiring is to develop competency and effectiveness in the long term, anticipating the staffing needs and challenges one will face in the future. In the long term, automation of tests is a key process activity for a test organization to be successful. Therefore, it is better to staff a test team with personnel having much coding experience.
- **Integration Testers:** Experienced integration test engineers are an asset in the system test group. When an integration test engineer experiences a defect, he or she is in a better position to identify the area where the problem may exist and assist developers in expediting defect resolution. It saves the developers much time because they do not have to attempt to reproduce the problem(s). Test engineers with integration testing experience are in a better position to know when the code is ready for system-level testing.
- **Information Technology Administrators:** Information technology (IT) administrators are very knowledgeable in configuring; setting up routers, servers, and firewalls; and troubleshooting. In addition, they know about different kinds of tools available in the market. Test engineers with IT experience can easily set up the test environments and maintain test laboratories.
- **Technical Support Personnel:** Customer support engineers are often good testers, especially in performing high level-tests such as acceptance tests. They are in a position to evaluate the impact of a defect on the customers. Their knowledge of business functions and interaction with many different customers make them invaluable to a test group.
- **Technical Writers:** One of the tasks of a system test engineer is to go through all the manuals, user guides, troubleshooting guides, and release notes. Therefore, it is useful to have technical writers in a test group to execute documentation tests. The idea in performing documentation tests is to ensure that all the user documentations are accurate and usable. Moreover, they can be useful in performing usability and graphical user interface testing.

- **System Test Engineers:** It is effective and efficient to hire test specialists who are already trained, say, by another company, in test techniques, tools, technology, and domain knowledge. An experienced test engineer knows the various testing strategies and techniques used to find more defects with less effort, including thorough understanding of system architectures and their functionalities, programming languages, code structures, and implementations. This is also a good way to acquire expertise in specific testing tools and domain knowledge.
- **Recent Graduates:** One can hire recent engineering graduates as junior test engineers and train them the way one wants them to be. They are much eager, enthusiastic, and willing to learn more about testing. They are good at developing test tools and automating test cases. New graduates need mentoring from a senior test engineer to be productive.

16.5 RECRUITING TEST ENGINEERS

Interviewing is the primary way of evaluating applicants in the recruiting process. One must possess good skills at identifying the strengths and weaknesses of an applicant. However, interviewing is a skill that improves with practice. The following characteristics make an excellent interviewer:

- The interviewer should be prepared in advance.
- The interviewer must be a good listener.
- The interviewer must be focused in questioning and probing.
- The interviewer needs to accurately document his or her assessment of the candidate during the interview.
- It is effective to give the candidate an opportunity to ask questions.
- It is effective to be focused on specific requirements of the job to be filled and not on the resume.
- It is fruitful to evaluate an applicant based on facts and data, rather than one's "gut feeling."

A recruiting process must be in place in order to be effective in hiring excellent test engineers. Six simple phases are recommended to be followed in staffing the system test team, as illustrated in Figure 16.4.

16.5.1 Job Requisition

The first step in hiring is to get an approval from executive management, which is sometimes referred to as having a "requisition number." A requisition number means that the position has been approved by the chief financial officer (CFO) of the organization. The requisition application form, which is usually available from the human resources department, is filled out and submitted to executive

Figure 16.4 Six phases of effective recruiting process.

management for approval. The following information is provided on the application form: *title of the position*, *salary range*, and *justification*. The hiring manager needs to justify why and for which test project this position is being created and explain the impact if this position is not filled. Salary may vary depending on the test position, experience, and education sought in an applicant. Therefore, it is recommended to put a range, rather than an exact figure. Of course, the salary range should be based on the recommended guidelines provided by the human resources department. It is recommended to have a requisition number for the test position from human resources before proceeding to the next step in the hiring process.

16.5.2 Job Profiling

During or after the requisition phase one determines the specific requirements, also called a *job profile*, of the job to be filled. At this point it is useful to keep in mind the principle: *if you don't know what you are looking for, you're not likely to find it*. The job profile serves two purposes: (i) for job posting by recruiters at various places such as career portals, job fairs, newspapers, and state department of employment and (ii) as a formal definition of the role and responsibilities of the new hiree that can be used when the applicant is interviewed. A job profile contains the following information:

- Position
- Essential duties and responsibilities
- Skills and competence
- Experience
- Education, training, certificate, security clearances (if required)

16.5.3 Screening Resumes

Resumes come from different sources, including employee referrals, advertisements, job fairs, career portals, college job boards, and agencies. Evaluate the

resumes against the job profile without delay. Identify the qualified candidates by addressing the following three critical questions:

- Does the candidate possess the necessary experience and skills to do the job?
- What is the candidate's motivation to do the job and will the candidate fit within the company culture?
- How will the candidate perform on the job?

The resume of an applicant is likely to answer the first question. For the other two questions, one may schedule an interview with the candidate. To cut cost and save time, the idea of a phone interview is gaining much ground. The telephone interview should continue uninterrupted for 30 minutes and be documented. One can ask open-ended questions about candidates' past actions to explore their behavior and the motivation behind them. An example of an open-ended question is: Tell me about one or two defects you reported and how you verified the fixes. Another open-ended question is: What is your motivation for doing this kind of work?

At this point it is useful to recall the five-C characteristics. An example question that explores the *controlled* characteristic is: Can you tell me about a specific project where you had a million details? A question that explores the probing ability of an applicant is: What method do you use to keep track of all the defects you need to verify? At the end of the telephone screening, the candidate should be allowed to ask questions. Hiring is a two-way street, meaning the hiring test manager and the candidate need to develop mutual respect. By the end of the telephone screening, one should have answers to all three questions. If an answer to any of the three questions is not positive, the resume should be put aside and the interview terminated. Otherwise, an on-site interview with the candidate should be scheduled.

16.5.4 Coordinating an Interview Team

The method of using multiple interviewers helps reduce the individual bias, makes efficient use of individual interviewer time, and permits the collection of in-depth information about the candidate. The following guidelines can help make the team interview process effective:

1. One person coordinates the team interview process.
2. The team size may vary from three to six, and at least one member must be from another group, such as software development or customer support. Other managers may be included in the team to interview principal engineers, technical leads, or management candidates.
3. Give each interviewer a copy of the job profile and the candidate's resume. This will ensure that the interviewers know about the candidate's background before the meeting and what to look for when they interview.
4. Assign each interviewer a different dimension of the job so that each can focus on a specific area. For example, one person could obtain information

on technical skills, another on professional skills, and a third about the candidate's behavior. Some overlap in their assigned areas will give more than one perspective on each skill area.

5. Have each interviewer use a standard outline containing questions that focus on the candidate's past behavior and accomplishments in the area being evaluated. Decide which interviewer will tell the applicant about various aspects of the job and the test organization so the interviewers do not repeat or contradict one another. Schedule that interviewer as the first to meet the candidate.
6. After the interview, have each interviewer rate the candidate on the dimension he or she was assigned.

16.5.5 Interviewing

One must remember that interviewing is a process for gathering information about the candidate. The gathered information is used to evaluate the candidate immediately after the interview. Questioning and probing are two important aspects of an interview. After asking the right question, the interviewer should remain silent to allow the candidate to formulate a response and should listen carefully. A follow-up question will help to understand and assess the depth of the candidate's experience. Providing examples and time to think will help the candidate come up with an answer. It is the interviewer's responsibility to provide a memorable environment. In order to improve the effectiveness of an interview, the interviewes may follow these steps:

1. Introduce yourself and your position to give the interview a gentle start.
2. If you are the first interviewer, then explain the various aspects of the job and the test organization.
3. Ask the first question based on the specific assigned area that needs to be explored. Take notes by paraphrasing what was said. Probe until you get an accurate understanding.
4. Ask the next question and probe until you get a good understanding. Similarly, ask more questions.
5. Invite the candidate to ask a few questions.
6. Close the interview and introduce the applicant to the next interviewer.
7. Complete the evaluation immediately after the interview.

During the interview aim for an 80/20 ratio of talk of the applicant to the interviewer. Display energy and show enthusiasm for the job for which the candidate is being interviewed. Be precise with the language, and interrupt the candidate with "I understand," "I agree," or "I appreciate" phrases. Formulating questions is a critical aspect of a successful interview. Therefore, be prepared with a questionnaire before the interview. In the following, we provide some sample questions in each dimension of the five-C characteristics.

Meticulous and thorough:

- Can you tell me about a specific test project where you had a large number of details? (Probe) How did you organize them?

Systematized and tidy:

- What do you do to keep yourself on track for your commitments and priorities?
- What method do you have for keeping track of what you have agreed to with clients or colleagues?

Dependable and trustworthy:

- Tell me about a team project you worked on. (Probe) What was the project and how were responsibilities assigned and managed? (Probe) How did you manage your responsibilities?

Precise and scrupulous:

- How do you approach completing testing tasks?
- Give me an example of a test project that demanded precision and thoroughness.
- Walk me through your solution to one of the most difficult testing problems you have faced.

Written communication:

- What rules, or standards, do you personally follow when writing a test plan document? Describe a time when you had to compose an acceptance test plan document to be viewed by people with diverse backgrounds.
- How do you write a defect report?

Accomplish and Carrythrough:

- Describe a time when you worked hard on a test project only to end without completing it? (Probe) Why did it not get finished?
- Tell me about a test project that you led? (Probe) What did the project accomplish and what did you do?

Adjustable and malleable:

- Have you ever had to shift gears in the middle of a test project? (Probe) What happened and what did you do?

Synergetic and team player:

- Do you prefer working on your own or in a team?
- People who work on test projects often find that priorities vary or last-minute things occur that cause you to have to change what you are doing or pinch-hit for someone else. Has that ever happened to you? (Probe) What did you do?
- Tell me about a recent test project that was a success. (Probe) What made the project so successful?

Verbal communication:

- In your last job who did you have to communicate with on a regular basis? (Probe) How did you communicate with them? How would you describe yourself as a communicator?
- Tell me about the manner and the tone you used to discuss a defect report with other members in a project team meeting.

Tactful and relationship building:

- Tell me about your best business relationships.
- Describe a situation where you had to build a relationship with someone (developers) you did not like.
- What makes a good working relationship?
- With what type of people do you like to work?

Determined and tenacious:

- Tell me about a time when you really had to persist to get the outcome you wanted.
- Tell me about a controversial defect for which you advocated.

Results driven and goal oriented:

- Do you consider yourself goal driven or task driven? (Probe) Can you give me an example?
- Can you explain the strategy you followed in your last testing project? (Probe) What was your tactics to execute the strategy?
- What personal goals have you set for the next year or so?
- Describe the career you see for yourself in testing?

Fast learner and quick study:

- Describe a situation where you had to learn new testing skills quickly before you could start? (Probe) How did you handle the fast-paced learning?

Innovative and visionary:

- Give me some specific examples of creative solutions of which are proud? (Probe) What did others think about them?

Apart from evaluating the candidate's technical and interpersonnel skills, other kinds of assessments are conducted to ensure that the candidate can and will do the job. These assessments are explained in the following.

- Assess the candidate's listening skills by observing whether or not the candidate frequently interrupts the interviewer.
- Assess how well the candidate answers the questions. Observe whether or not the answers are well articulated and brief.
- Assess the kinds of questions the candidate asks and whether or not the candidate takes the initiative to know more about the position and the company.

- Assess the response to know whether the candidate will do the job whatever it takes.
- Take note of the attitude of the candidate toward the testing profession.
- Take note of the candidate's attitude toward supervisors and colleagues in past workplaces.
- If the resume indicates excessive job changes, observe how the candidate explains it. Frequent job changes may indicate that the candidate may not be a stable person, may easily get bored, or cannot get along with colleagues.
- Do not hire an engineer who did not get a job as a software developer. A bad software developer is not likely to make a good software tester.

It is important to select good, incisive questions. At the same time, it is counterproductive to ask ineffective questions. Some examples of types of ineffective questions are as follows:

- A question that produces a "yes" or "no" answer is ineffective unless the candidate is asked to justify the answer.
- It is not useful to provide a lead in a question. For example, "You must have had to work over the weekend to get everything done on time?" can be adapted to "What did you do to handle the situation?" to elicit a more meaningful response.
- It is not conducive to ask a threatening question. The question, "Why didn't you work over the weekend?" may be perceived as threatening. Such a question may put candidates on the defensive and may inhibit their responses during the rest of the interview.
- Avoid multiple-choice questions. These questions may create confusion in the mind of the candidate.
- Completely avoid questions about philosophy, beliefs, and opinion. An example is "In your opinion, what qualities are essential to be an effective test engineer?" Candidates tend to answer what the interviewer wants to hear.
- Avoid illegal questions. A question can be considered illegal if it can be used to discriminate against the candidate on a basis that is not critical to the job. Avoid questions that may probe for a possible *disability*, *personal characteristics*, or *personal situation*. Examples of such questions are as follows:

  What is your date of birth?

  Is your wife expecting a child?

  How will your spouse feel about you traveling?

  What does your spouse do?

  Do you have any handicaps that might prevent you from performing this job?

  By the way, where did you say you are from?

Show me your green card?

I have a little kid, how many do you have?

Are your kids in daycare?

It is more effective to perform an evaluation of the candidate right after the interview. This is because the general impression about the candidate is still fresh in the mind of the interviewer. The evaluation should contain the following information:

- Basic information: State the position, the candidate's name, the interviewer's name, and the date.
- Professional skill: Rank on a scale of 1 (poor) to 10 (excellent).
- Behaviors: Rank on a scale of 1 (poor) to 10 (excellent).
- Organizational fit: Rank on a scale of 1 (poor) to 10 (excellent).
- Strengths: Summarize the candidate's strengths for this position.
- Weakness: Summarize the candidate's weaknesses for this position.
- General comments: State any other information that may be useful.

16.5.6 Making a Decision

It is recommended to have a debriefing meeting of all the interviewers at the end of the interview to compare their individual ratings. If there is a discrepancy among the ratings, a consensus is sought. A reference check needs to be done for a candidate about to be hired. Such checks provide additional and valuable information about the candidate which may not be possible to obtain during an interview process. Attention must be given to the words the references use during the conversation, hence the underlying message. The hiring manager should make a decision as quickly as possible because good candidates do not stay in the market for long.

16.6 RETAINING TEST ENGINEERS

Retaining testers is a big challenge, especially in today's market. Test engineers must know they have support from various levels of management. Managers should be encouraged to recognize testing efforts at par with development efforts. Management should treat its test engineers as professionals and as a part of the overall team that delivers a quality product. The following are key factors that positively impact the ability of an organization to retain good system test engineers.

16.6.1 Career Path

It is desirable to have a clear career path within the system testing group to be able to attract and retain qualified staff test engineers. Each organization needs to establish its own career path and responsibility levels for test engineers. This can be similar to the test team hierarchy described in Section 16.3. An organization

can establish performance- and experience-based criteria to move from one role to another. To deliver high-quality products, testing must not be a dead-end career path. There must be ample opportunities within an organization for test engineers to move into other areas, such as development, system engineering, customer support, and project management. A test manager must work with team members for their career advancement—this is true for any division of an organization, and it is no different for system test engineers.

16.6.2 Training

Testing technology is changing almost as fast as development. Training is necessary to continually improve testing knowledge, business knowledge, interpersonal skill, leadership capability, communication skills, and process issues related to testing. For example, testing tools are changing with the technology. It is no longer cost-effective to build in-house automation tools when there are corporations focusing on just that effort. Testing tools should be continuously evaluated to know what they can and cannot do and test engineers should be trained in new tools. If test engineers have not done any test automation, they should be encouraged to attend some introductory lectures on test automation. They are likely to become very good in the development of automated test cases with some training. There are several options available to train the staff [7]:

- **On-Site Commercial Training:** An individual contractor from another company is invited to train a group of system test engineers. Training materials are customized to meet the unique needs of the group. The trainer provides hands-on training on test tools for automation, code review process, and design of test cases using examples from the domain of interest.
- **Public Forum Training:** Individual test engineers are sent to a public training class. This is useful if the group size is small. Here, the instructor cannot customize the material for a couple of students because the students may be from different companies with diverse backgrounds.
- **In-House Training:** Some companies run in-house training programs for new employees in development and quality assurance processes, such as the ISO certification process. The drawbacks of this approach are the unavailability of good trainers and lack of good training materials.
- **Specialty Training:** There are many special training programs available, such as web-enabled training and distance learning. There are product-based training programs, such as Microsoft's certified systems engineer (MCSE) and Cisco's certified internetwork expert (CCIE). It is important for a network system test engineer to have a thorough understanding of how a network system works, rather than just if it works.
- **Mentoring:** This is a process where a new employee works with a senior, experienced system test engineer called a mentor. The mentor's job is to train and promote the career of a newly hired test engineer. A mentor's

responsibility begins the day the new person arrives for work. A mentor owns the following minimal responsibilities:

- Showing how to fill out corporate paperwork to get things done
- Handling questions the person may have regarding normal everyday operations
- Training the person in the methodology being used
- Having discussions on how to test, what are good tests, when, how, and why tools should be used, and so on

16.6.3 Reward System

Rewards are a great way to encourage people to continue improving and stay within the test organization. Salary increment based on merit, pay range adjustment, stock option, and office environment are all the things that can make a difference. Pay is not everything, but it is definitely an issue that must be addressed. It is recommended to have test engineers' pay scales above developers'. The key idea is to attract developers in to the testing group and then train them to be system testers. Another way to appreciate the good work of test engineers is to institute a formal awards program with certificates, plaques, and other awards, such as gift certificates from a very nice restaurant, to honor teams for their special effort.

16.7 TEAM BUILDING

It is important that the test engineers in a group work together with solidarity to maximize the ultimate success of the group. Building a team attitude means managing the engineers in a way that fosters teamwork instead of individual gains. It involves ongoing effort and practice among all the team members, including the team manager. The essential ingredients of a good work environment for test engineers are explained in the following.

16.7.1 Expectations

All test engineers should know what their assignments are and the expectations that come with the assignments. They should also know the larger consequences of not being able to meet the expectations. The role and responsibilities of a test engineer need to be clearly defined. Team members need to have opportunities to clarify and, if possible, negotiate their roles and relationships with one another. Team members may be empowered to work out responsibilities and roles among themselves and report their recommendations to the test manager. A test manager should schedule a weekly one-on-one meeting with each test engineer to talk about how things are going and to know the engineers and their interests beyond the official work. A manager needs to listen to their concerns and show empathy, understanding, and encouragement. Informal discussions and sharing of thoughts can go a long way in understanding their needs, and a manager can help them to improve their self-esteem.

16.7.2 Consistency

Consistency is typically something system test engineers look for in their testing. Having it in the testing process is just as beneficial. The process should not be a fluid one—it should be changing all the time. For example, when filing a defect, the severity of the defect must be assigned based on the defined criteria, and it should not be changed from time to time. It should be used consistently throughout the organization. Stability is essential for successful testing of software, and the same holds for the processes people are expected to follow. It is important to be consistent with the entry and exit criteria for testing. If and when inconsistency is accepted, it is necessary to tell the reasons to everyone and indicate the need to correct the direction.

16.7.3 Information Sharing

Sharing information is key to being successful. An atmosphere for timely, smooth flow of high-value information between self and system test engineers should be created and open expression of ideas and opinions encouraged. Methods for expressing ideas and opinions include e-mail, white boards, discussions, presentations, centralized documentation repositories, and short and regular meetings. Test engineers should be kept up-to-date with useful information, such as company policy change, upcoming test projects, and organizational events. It is important to ensure that there are no *surprises* for the team members. Team members should be encouraged to share beneficial information by talking. Direct and open communication with others fosters trust, enhances information flow, and builds stronger relationship. Brainstorming is also an efficient method for information sharing. A manger needs to take notes during the brainstorming sessions to distribute the minutes. At these sessions, all ideas are welcome, but no analysis is performed during the meetings. Brainstorming sessions are generally fun and educational. Sometimes the most ridiculous sounding idea becomes the *concept* in the future.

16.7.4 Standardization

It is imperative to have a standard vocabulary of terms so everyone understands what is being said. For instance, in a defect review meeting of eight people, if a test manager asks the question "What is the difference between priority and severity of a defect?" one will hear many different answers. Standardization of terminology will not only help the team, but if it is shared with others in the organization, the terms will begin showing up in requirements and other documents. Eventually, everyone will understand what is being said.

16.7.5 Test Environments

It would be reasonable to ask "What do test environments have to do with team building?" Having test environments that can be shared will greatly enhance testing efforts. However, if one has a single test environment, teamwork is critical to

being responsive. Having good test environments leads to better team cohesiveness. A test environment should provide test engineers with access to the defect tracking database, test factory database, system configuration database, and other information resources to execute the test cases and report results efficiently. It is recommended to have a separate test laboratory, where the test environments can be set up by test engineers to execute test cases. To help the test organization to maximize productivity and efficiency through their test laboratory, the following ideas should be considered: (i) creating more than one test environment, (ii) purchasing new tools and equipment, (iii) updating existing equipment, (iv) replacing old equipment, (v) keeping the test environments clean, and (vi) keeping an inventory of all the equipment in the laboratory.

16.7.6 Recognitions

It is productive and cost-effective to value the work every test engineer performs. Verbal recognitions for engineer's contributions should be given. Acknowledging and celebrating the group accomplishment are a powerful way to recognize the team efforts and to keep the motivation and momentum afloat. The test team should be told that its efforts make a difference to the quality of the product through a personal memo. Special get-togethers, such as team lunches and breakfasts, should be organized upon successful completion of a test project.

16.8 SUMMARY

This chapter dealt with the organization of test groups in terms of functionality, management, and staffing. The recommendation is to have at least two test groups: integration test group and system test group. Unit testing must be performed by the programmers during coding so that there is no need for a separate test group for unit-level testing. A integration test group largely consists of developers. A system test group can be further divided into a number of subgroups, each handling a specialized task, as an organization grows. The need for differentiating between the quality assurance group and the system testing group was explained. The mandate of a quality assurance group is much broader than a system testing group. A quality assurance group has a larger role in ensuring conformance to best development practices throughout the organization, which is beyond the scope of a system test group.

Next, details of effectively staffing junior, senior, and principal test engineers were presented. Six simple phases of recruiting test engineers were discussed: job requisition, job profiling, screening resume, coordinating an interview team, interviewing, and decision making. The chapter explained some details of how to retain test engineers by means of training, rewarding, and establishing a clear career path for them within the test organization.

Finally, the chapter explained the ways to build and manage a test team with solidarity in a way that fosters team spirit instead of individual gain. Six

ingredients were discussed to maximize the effectiveness of a test team: expectations, consistency, information sharing, standardization, testing environment, and recognition.

LITERATURE REVIEW

A good discussion of test team organization, staffing test engineers, and management of test teams can be found in Chapters 8 and 9 of the books by Black [3] and Craig and Jaskiel [7], respectively. Each chapter presents useful examples and opinions concerning hiring, test organizations, motivating testers, and managing test team. In addition, the book by Craig and Jaskiel presents an excellent chapter, namely Chapter 10, on what makes a good test manager. A good tutorial on test team management can be found in Chapter 5 of the book by Dustin, Rashka, and Paul [1]. The chapter contains sections on the organizational structure of a test team, test effort sizing, test engineer recruiting, and roles and responsibilities of a test manager and a test engineer.

A description of career paths for test engineers can be found in E. Weyuker, T. Ostrand, J. Brophy, and R. Prasad, "Clearing a Career Path for Software Tester," *IEEE Software*, Vol. 17, No. 2, March/April 2000, pp. 76–82. The authors describe five phases, namely apprenticeship, mastery, specialization, leadership, and top-level tester, in a test engineer career path at AT&T. The researchers have identified a group of general engineering skills that a test engineer needs to have. The general skill set includes knowledge about computers, system architecture, software development methods, information systems, and operating and database systems.

Successful Manager's Handbook (B. Davis, C. Skube, L. Hellervik, S. Gebelein, and J. Sheard, Personnel Decisions International, Minneapolis, 1996) is a comprehensive book on improving managerial skills. The book contains a thorough discussion of how to become an effective manager by practicing nine dimensions/factors: administrative factor, leadership factor, interpersonal factor, communication factor, motivation factor, self-management factor, organizational knowledge factor, organizational strategy factor, and thinking factor. This is a highly recommended book to become a successful test manager.

REFERENCES

1. E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley, Reading, MA, 1999.
2. Institute of Electrical and Electronics Engineers (IEEE). *IEEE Standard Glossary of Software Engineering Terminology*, IEEE standard 610. 12–1990. IEEE, New York, 1990.
3. R. Black. *Managing Testing Process*, 2nd ed. Wiley, New York, 2002.
4. B. Hetzel. *The Complete Guide to Software Testing*. Wiley, New York, 1988.
5. J. J. Cappel, V. R. Prybutok, and B. Varghese. A Closer Look At: Attention to Detail. *Communications of the ACM*, July 2005, pp. 87-92.
6. J. Cook. Building a Responsive Testing Team. Paper presented at the International Conference on Software Testing, Analysis and Review, Orlando, FL, May 1999.

7. R. D. Craig and S. P. Jaskiel. *Systematic Software Testing*. Artech House, Norwood, MA, 2002.

Exercises

1. Why does the integration test group function within the development group?
2. Unit-level tests are designed and executed by individual developers. Why don't we form a unit test group within a development group similar to an integration test group?
3. Discuss the advantages and disadvantages of having an independent system test group that is separated from the software development group with its own reporting structure.
4. What are the roles and responsibilities of a test manager and a (software) quality assurance manager?
5. Assume that you are a system test manager for a startup software development organization. Your company develops software in the network security domain. You want to hire a new test engineer for your team. Prepare a job profile for the test engineer that you could use for job posting and for screening the applicants.
6. For the organization described in exercise 5, prepare a job profile for hiring a quality assurance manager to lead the quality management group.
7. Do you think you can ask the following questions during an interview. Justify your answers.
   (a) Do you smoke? We maintain a smoke-free work environment. Can you work without smoking on the job?
   (b) Have you ever been convicted of a felony?
   (c) Have you ever been arrested?
   (d) Have you ever tested positive for the AIDS virus?
   (e) Are you gay?
   (f) What kind of name is Sagar?
   (g) Is it an Indian name?
8. Write some of your interview questions to assess test engineers in the following areas:
   (a) Time management
   (b) Problem solving
   (c) Decision making
   (d) Multitasking
   (e) Leadership
   (f) Motivation
   (g) Sincerity

Figure 16.5 System test organization as part of development.

(h) Work environment

(i) Quickness and promptness

9. In small startup organizations, the system test group is a part of the development group, as shown in Figure 16.5. Do you think it is a good idea to have this structure? Justify your answer.
10. What role does a test manager play in building a system test team?

CHAPTER 17

Software Quality

Quality is never an accident; it is always the result of intelligent effort.
— *John Ruskin*

17.1 FIVE VIEWS OF SOFTWARE QUALITY

In the early days of computers, software developers mainly focused on product functionalities, and most of the end users were highly qualified professionals, such as mathematicians, scientists, and engineers. Development of personal computers and advances in computer networks, the World Wide Web, and graphical user interface made computer software highly accessible to all kinds of users. These days there is widespread computerization of many processes that used to be done by hand. For example, until the late 1990s taxpayers used to file returns on paper, but these days there are numerous web-based tax filing systems. There has been increasing customer expectations in terms of better quality in software products, and developers are under tremendous pressure to deliver high-quality products at a lower cost. Even though competing products deliver the same functionalities, it is the lower cost products with better quality attributes that survive in the competitive market. Therefore, all stakeholders—users, customers, developers, testers, and managers—in a product must have a broad understanding of the overall concept of software quality.

A number of factors influence the making and buying of software products. These factors are user's needs and expectations, the manufacturer's considerations, the inherent characteristics of a product, and the perceived value of a product. To be able to capture the quality concept, it is important to study quality from a broader perspective. This is because the concept of quality predates software development. In a much cited paper published in the *Sloan Management Review* [1], Garvin has analyzed how quality is perceived in different manners in different domains, namely, philosophy, economics, marketing, and management:

Transcendental View: In the transcendental view quality is something that can be recognized through experience but is not defined in some tractable

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

form. Quality is viewed to be something ideal, which is too complex to lend itself to be precisely defined. However, a good-quality object stands out, and it is easily recognized. Because of the philosophical nature of the transcendental view, no effort is made to express it using concrete measures.

User View: The user view concerns the extent to which a product meets user needs and expectations. Quality is not just viewed in terms of what a product can deliver, but it is also influenced by the service provisions in the sales contract. In this view, a user is concerned with whether or not a product is fit for use. This view is highly personalized in nature. The idea of *operational profile*, discussed in Chapter 15, plays an important role in this view. Because of the personalized nature of the product view, a product is considered to be of good quality if it satisfies the needs of a large number of customers. It is useful to identify what product attributes users consider to be important. The reader may note that the user view can encompass many subjective elements apart from the expected functionalities central to user satisfaction. Examples of subjective elements are *usability*, *reliability*, *testability*, and *efficiency*.

Manufacturing View: The manufacturing view has its genesis in the manufacturing sectors, such as the automobile and electronics sectors. In this view, quality is seen as conforming to requirements. Any deviation from the stated requirements is seen as reducing the quality of the product. The concept of *process* plays a key role in the manufacturing view. Products are to be manufactured "right the first time" so that development cost and maintenance cost are reduced. However, there is no guarantee that conforming to process standards will lead to good products. Some criticize this view with an argument that conformance to a process can only lead to *uniformity* in the products, and, therefore, it is possible to manufacture bad-quality products in a consistent manner. However, product quality can be incrementally enhanced by continuously improving the process. Development of the *capability maturity model* (CMM) [2] and ISO 9001 [3] are based on the manufacturing view.

Product View: The central hypothesis in the product view is this: *If a product is manufactured with good internal properties, then it will have good external qualities*. The product view is attractive because it gives rise to an opportunity to explore causal relationships between *internal properties* and *external qualities* of a product. In this view, the current quality level of a product indicates the presence or absence of measurable product properties. The product view of quality can be assessed in an objective manner. An example of the product view of software quality is that high degree of modularity, which is an internal property, makes a software testable and maintainable.

Value-Based View: The value-based view represents a merger of two independent concepts: *excellence* and *worth*. Quality is a measure of excellence, and value is a measure of worth. The central idea in the value-based

view is how much a customer is willing to pay for a certain level of quality. The reality is that quality is meaningless if a product does not make economic sense. Essentially, the value-based view represents a trade-off between cost and quality.

Measuring Quality The five viewpoints help us in understanding different aspects of the quality concept. On the other hand, measurement allows us to have a quantitative view of the quality concept. In the following, we explain the reasons for developing a quantitative view of a software system [4]:

- Measurement allows us to establish baselines for qualities. Developers must know the minimum level of quality they must deliver for a product to be acceptable.
- Organizations make continuous improvements in their process models—and an improvement has a cost associated with it. Organizations need to know how much improvement in quality is achieved at a certain cost incurred due to process improvement. This causal relationship is useful in making management decisions concerning process improvement. Sometimes it may be worth investing more in process improvement, whereas some other time the return may not be significant.
- The present level of quality of a product needs to be evaluated so the need for improvements can be investigated.

Measurement of User's View The user's view encompasses a number of quality factors, such as functionality, reliability, and usability. It is easy to measure how much of the functionalities a software product delivers by designing at least one test case for each functionality. A product may require multiple test cases for the same functionality if the functionality is to be performed in different execution environments. Then, the ratio of the number of passed test cases to the total number of test cases designed to verify the functionalities is a measure of the functionalities delivered by the product. Among the qualities that reflect the user's view, the concept of *reliability* has drawn the most attention of researchers.

In the ISO 9126 quality model, *usability* has been broken down into three subcharacteristics, namely, *learnability*, *understandability*, and *operability*. Learnability can be specified as the average elapsed time for a typical user to gain a certain level of competence in using the product. Similarly, understandability can be quantified as the average time needed by a typical user to gain a certain level of understanding of the product. One can quantify operability in a similar manner. The basic idea of breaking down usability into learnability, understandability, and operability can be seen in light of Gilb's technique [5]: *The quality concept is broken down into component parts until each can be stated in terms of directly measurable attributes*. Gilb's technique is a general one to be applicable to a wide variety of user-level qualities.

Measurement of Manufacturer's View Manufacturers are interested in obtaining measures of the following two different quantities:

- **Defect Count:** How many defects have been detected?
- **Rework Cost:** How much does it cost to fix the known defects?

Defect count represents the number of all the defects that have been detected so far. If a product is in operation, this count includes the defects detected during development and operation. A defect count reflects the quality of work produced. Merely counting the defects is of not much use unless something can be done to improve the development process to reduce the defect count in subsequent projects. One can analyze the defects as follows:

- For each defect identify the development phase in which it was introduced and the phase in which it was discovered. Let us assume that a large fraction of the defects are introduced in the requirements gathering phase, and those are discovered during system testing. Then, we can conclude that requirement analysis was not adequately performed. We can also conclude that work done subsequently, such as design verification and unit testing, were not of high standard. If a large number of defects are found during system operation, one can say that system testing was not rigorously performed.
- Categorize the defects based on modules. Assuming that a module is a cohesive entity performing a well-defined task, by identifying the modules containing most of the defects we can identify where things are going wrong. This information can be used in managing resources. For example, if a large number of defects are found in a communication module in a distributed application, more resource could be allocated to train developers in the details of the communication system.
- To compare defects across modules and products in a meaningful way, normalize the defect count by product size. By normalizing defect count by product size in terms of the number of LOC, we can obtain a measure, called *defect density*. Intuitively, defect density is expressed as the *number of defects found per thousand lines of code*.
- Separate the defects found during operation from the ones found during development. The ratio of the number of defects found during operation to the total number of defects is a measure of the effectiveness of the entire gamut of test activities. If the ratio is close to zero, we can say that testing was highly effective. On the other hand, if the ratio is farther from the ideal value of zero, say, 0.2, it is apparent that all the testing activities detected only 80% of the defects.

After defects are detected, the developers make an effort to fix them. Ultimately, it costs some money to fix defects—this is apart from the "reputation" cost to an organization from defects discovered during operation. The rework cost includes all the additional cost associated with defect-related activities, such as fixing documents. Rework is an additional cost that is incurred due to work being done in a less than perfect manner the first time it was done. It is obvious that

organizations strive to reduce the total cost of software development, including the rework cost. The rework cost can be split into two parts as follows:

- **Development Rework Cost:** This is the rework cost incurred *before* a product is released to the customers.
- **Operation Rework Cost:** This is the rework cost incurred *when* a product is in operation.

On the one hand, the development rework cost is a measure of development efficiency. In other words, if the development rework cost is zero, then the development efficiency is very high. On the other hand, the operation rework cost is a measure of the delivered quality of the product in operation. If the development rework cost is zero, then the delivered quality of the product in operation is very high. This is because the customers have not encountered any defect and, consequently, the development team is not spending any resource on defect fixing.

17.2 MCCALL'S QUALITY FACTORS AND CRITERIA

The concept of software quality and the efforts to understand it in terms of measurable quantities date back to the mid-1970s. McCall, Richards, and Walters [6] were the first to study the concept of software quality in terms of quality factors and quality criteria.

17.2.1 Quality Factors

A *quality factor* represents a behavioral characteristic of a system. Some examples of high-level quality factors are *correctness*, *reliability*, *efficiency*, *testability*, *portability*, and *reusability*. A full list of the quality factors will be given in a later part of this section. As the examples show, quality factors are external attributes of a software system. Customers, software developers, and quality assurance engineers are interested in different quality factors to a different extent. For example, customers may want an efficient and reliable software with less concern for portability. The developers strive to meet customer needs by making their system efficient and reliable, at the same time making the product portable and reusable to reduce the cost of software development. The software quality assurance team is more interested in the testability of a system so that some other factors, such as correctness, reliability, and efficiency, can be easily verified through testing. The testability factor is important to developers and customers as well: (i) Developers want to test their product before delivering it to the software quality assurance team and (ii) customers want to perform acceptance tests before taking delivery of a product. In Table 17.1, we list the quality factors as defined by McCall et al. [6]. Now we explain the 11 quality factors in more detail:

Correctness: A software system is expected to meet the explicitly specified functional requirements and the implicitly expected nonfunctional requirements. If a software system satisfies all the functional requirements, the

TABLE 17.1 McCall's Quality Factors

Quality Factors	Definition
Correctness	Extent to which a program satisfies its specifications and fulfills the user's mission objectives
Reliability	Extent to which a program can be expected to perform its intended function with required precision
Efficiency	Amount of computing resources and code required by a program to perform a function
Integrity	Extent to which access to software or data by unauthorized persons can be controlled
Usability	Effort required to learn, operate, prepare input, and interpret output of a program
Maintainability	Effort required to locate and fix a defect in an operational program
Testability	Effort required to test a program to ensure that it performs its intended functions
Flexibility	Effort required to modify an operational program
Portability	Effort required to transfer a program from one hardware and/or software environment to another
Reusability	Extent to which parts of a software system can be reused in other applications
Interoperability	Effort required to couple one system with another

Source: From ref. 6.

system is said to be correct. However, a correct software system may still be unacceptable to customers if the system fails to meet unstated requirements, such as stability, performance, and scalability. On the other hand, even an incorrect system may be accepted by users.

Reliability: It is difficult to construct large software systems which are correct. A few functions may not work in all execution scenarios, and, therefore, the software is considered to be incorrect. However, the software may still be acceptable to customers because the execution scenarios causing the system to fail may not frequently occur when the system is deployed. Moreover, customers may accept software failures once in a while. Customers may still consider an incorrect system to be reliable if the failure rate is very small and it does not adversely affect their mission objectives. Reliability is a customer perception, and an incorrect software can still be considered to be reliable.

Efficiency: Efficiency concerns to what extent a software system utilizes resources, such as computing power, memory, disk space, communication bandwidth, and energy. A software system must utilize as little resources as possible to perform its functionalities. For example, by utilizing less communication bandwidth a base station in a cellular telephone network can support more users.

Integrity: A system's integrity refers to its ability to withstand attacks to its security. In other words, integrity refers to the extent to which access to software or data by unauthorized persons or programs can be controlled. Integrity has assumed a prominent role in today's network-based applications. Integrity is also an issue in multiuser systems.

Usability: A software system is considered to be usable if human users find it easy to use. Users put much emphasis on the user interface of software systems. Without a good user interface a software system may fizzle out even if it possesses many desired qualities. However, it must be remembered that a good user interface alone cannot make a product successful—the product must also be reliable, for example. If a software fails too often, no good user interface can keep it in the market.

Maintainability: In general, maintenance refers to the upkeep of products in response to deterioration of their components due to continued use of the products. Maintainability refers to how easily and inexpensively the maintenance tasks can be performed. For software products, there are three categories of maintenance activities: corrective, adaptive, and perfective. Corrective maintenance is a postrelease activity, and it refers to the removal of defects existing in an in-service software. The existing defects might have been known at the time of release of the product or might have been introduced during maintenance. Adaptive maintenance concerns adjusting software systems to changes in the execution environment. Perfective maintenance concerns modifying a software system to improve some of its qualities.

Testability: It is important to be able to verify every requirement, both explicitly stated and simply expected. Testability means the ability to verify requirements. At every stage of software development, it is necessary to consider the testability aspect of a product. Specifically, for each requirement we try to answer the question: What procedure should one use to test the requirement, and how easily can one verify it? To make a product testable, designers may have to instrument a design with functionalities not available to the customer.

Flexibility: Flexibility is reflected in the cost of modifying an operational system. As more and more changes are effected in a system throughout its operational phase, subsequent changes may cost more and more. If the initial design is not flexible, it is highly likely that subsequent changes are very expensive. In order to measure the flexibility of a system, one has to find an answer to the question: How easily can one add a new feature to a system?

Portability: Portability of a software system refers to how easily it can be adapted to run in a different execution environment. An execution environment is a broad term encompassing hardware platform, operating system, distributedness, and heterogeneity of the hardware system, to

name a few. Portability is important for developers because a minor adaptation of a system can increase its market potential. Moreover, portability gives customers an option to easily move from one execution environment to another to best utilize emerging technologies in furthering their business. Good design principles such as modularity facilitate portability. For example, all environment-related computations can be localized in a few modules so that those can be easily identified and modified to port the system to another environment.

Reusability: Reusability means if a significant portion of one product can be reused, maybe with minor modification, in another product. It may not be economically viable to reuse small components. Reusability saves the cost and time to develop and test the component being reused. In the field of scientific computing, mathematical libraries are commonly reused. Reusability is not just limited to product parts, rather it can be applied to processes as well. For example, we are very much interested in developing good processes that are largely repeatable.

Interoperability: In this age of computer networking, isolated software systems are turning into a rarity. Today's software systems are coupled at the input–output level with other software systems. Intuitively, interoperability means whether or not the output of one system is acceptable as input to another system; it is likely that the two systems run on different computers interconnected by a network. When we consider Internet-based applications and wireless applications, the need for interoperability is simply overriding. For example, users of document processing packages, such as LaTex and Microsoft Word, want to import a variety of images produced by different graphics packages. Therefore, the graphics packages and the document processing packages must be interoperable. Another example of interoperability is the ability to roam from one cellular phone network in one country to another cellular network in another country.

The 11 quality factors defined in Table 17.1 have been grouped into three broad categories as follows:

- Product operation
- Product revision
- Product transition

The elements of each of the three broad categories are identified and further explained in Table 17.2. It may be noted that the above three categories relate more to postdevelopment activities expectations and less to in-development activities. In other words, McCall's quality factors emphasize more on the quality levels of a product delivered by an organization and the quality levels of a delivered product relevant to product maintenance. Quality factors in the product operation category refer to delivered quality. Testability is an important quality factor that is of much significance to developers during both product development and maintenance. Maintainability, flexibility, and portability are desired quality factor sought

TABLE 17.2 Categorization of McCall's Quality Factors

Quality Categories	Quality Factors	Broad Objectives
Product operation	Correctness	Does it do what the customer wants?
	Reliability	Does it do it accurately all of the time?
	Efficiency	Does it quickly solve the intended problem?
	Integrity	Is it secure?
	Usability	Can I run it?
Product revision	Maintainability	Can it be fixed?
	Testability	Can it be tested?
	Flexibility	Can it be changed?
Product transition	Portability	Can it be used on another machine?
	Reusability	Can parts of it be reused?
	Interoperability	Can it interface with another system?

Source: From ref. 6.

in a product so that the task of supporting the product after delivery is less expensive. Reusability is a quality factor that has the potential to reduce the development cost of a project by allowing developers to reuse some of the components from an existing product. Interoperability allows a product to coexist with other products, systems, and features.

17.2.2 Quality Criteria

A *quality criterion* is an attribute of a quality factor that is related to software development. For example, modularity is an attribute of the architecture of a software system. A highly modular software allows designers to put cohesive components in one module, thereby increasing the maintainability of the system. Similarly, traceability of a user requirement allows developers to accurately map the requirement to a subset of the modules, thereby increasing the correctness of the system. Some quality criteria relate to products and some to personnel. For example, modularity is a product-related quality criterion, whereas training concerns development and software quality assurance personnel. In Table 17.3, we list the 23 quality criteria defined by McCall et al. [6].

17.2.3 Relationship between Quality Factors and Criteria

The relationship between quality factors and quality criteria is shown in Figure 17.1. An arrow from a quality criterion to a quality factor means that the quality criterion has a positive impact on the quality factor. For example, traceability has a positive impact on correctness. Similarly, the quality criterion simplicity positively impacts reliability, usability, and testability.

Though it is desirable to improve all the quality factors, doing so may not be possible. This is because, in general, quality factors are not completely independent. Thus, we note two characteristics of the relationship as follows:

Figure 17.1 Relation between quality factors and quality criteria [6].

- If an effort is made to improve one quality factor, another quality factor may be degraded. For example, if an effort is made to make a software product testable, the efficiency of the software is likely to go down. To make code testable, programmers may not be able to write compact code. Moreover, if we are interested in making a product portable, the code must

TABLE 17.3 McCall's Quality Criteria

Quality Criteria	Definition
Access audit	Ease with which software and data can be checked for compliance with standards or other requirements
Access control	Provisions for control and protection of the software and data
Accuracy	Precision of computations and output
Communication commonality	Degree to which standard protocols and interfaces are used
Completeness	Degree to which a full implementation of the required functionalities has been achieved
Communicativeness	Ease with which inputs and outputs can be assimilated
Conciseness	Compactness of the source code, in terms of lines of code
Consistency	Use of uniform design and implementation techniques and notation throughout a project
Data commonality	Use of standard data representations
Error tolerance	Degree to which continuity of operation is ensured under adverse conditions
Execution efficiency	Run time efficiency of the software
Expandability	Degree to which storage requirements or software functions can be expanded
Generality	Breadth of the potential application of software components
Hardware independence	Degree to which the software is dependent on the underlying hardware
Instrumentation	Degree to which the software provides for measurement of its use or identification of errors
Modularity	Provision of highly independent modules
Operability	Ease of operation of the software
Self-documentation	Provision of in-line documentation that explains implementation of components
Simplicity	Ease with which the software can be understood.
Software system independence	Degree to which the software is independent of its software environment—nonstandard language constructs, operating system, libraries, database management system, etc.
Software efficiency	Run time storage requirements of the software
Traceability	Ability to link software components to requirements
Training	Ease with which new users can use the system

Source: From ref. 6.

be written in such a manner that it is easily understandable, and, hence, code need not be in a compact form. An effort to make code portable is likely to reduce its efficiency. In fact, attempts to improve integrity, usability, maintainability, testability, flexibility, portability, reusability, and interoperability will reduce the efficiency of a software system.

- Some quality factors positively impact others. For example, an effort to enhance the correctness of a system will increase its reliability. As another

example, an effort to enhance the testability of a system will improve its maintainability.

17.2.4 Quality Metrics

The high-level quality factors cannot be measured directly. For example, we cannot directly measure the testability of a software system. Neither can testability be expressed in "yes" or "no" terms. Instead, the degree of testability can be assessed by associating with testability a few quality metrics, namely, *simplicity*, *instrumentation*, *self-descriptiveness*, and *modularity*. A *quality metric* is a measure that captures some aspect of a quality criterion. One or more quality metrics should be associated with each criterion. The metrics can be derived as follows:

- Formulate a set of relevant questions concerning the quality criteria and seek a "yes" or "no" answer for each question.
- Divide the number of "yes" answers by the number of questions to obtain a value in the range of 0 to 1. The resulting number represents the intended quality metric.

For example, we can ask the following question concerning the *self-descriptiveness* of a product: *Is all documentation written clearly and simply such that procedures, functions, and algorithms can be easily understood?* Another question concerning self-descriptiveness is: *Is the design rationale behind a module clearly understood?* Different questions can have different degrees of importance in the computation of a metric, and, therefore, individual "yes" answers can be differently weighted in the above computation.

The above way of computing the value of a metric is highly subjective. The degree of subjectivity varies significantly from question to question in spite of the fact that all the responses are treated equally. It is difficult to combine different metrics to get a measure of a higher level quality factor. In addition, for some questions it is more meaningful to consider a response on a richer measurement scale. For example, the question "Is the design of a software system simple?" needs to be answered on a multiple ordinal scale to reflect a variety of possible answers, rather than a yes-or-no answer.

Similarly, one cannot directly measure the reliability of a system. However, the number of distinct failures observed so far is a measure of the initial reliability of the system. Moreover, the time gap between observed failures is treated as a measure of the reliability of a system.

17.3 ISO 9126 QUALITY CHARACTERISTICS

There has been international collaboration among experts to define a general framework for software quality. An expert group, under the aegis of the ISO, standardized a software quality document, namely, ISO 9126, which defines six broad, independent categories of quality characteristics as follows:

Functionality: A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.

Reliability: A set of attributes that bear on the capability of software to maintain its performance level under stated conditions for a stated period of time.

Usability: A set of attributes that bear on the effort needed for use and on the individual assessment of such use by a stated or implied set of users.

Efficiency: A set of attributes that bear on the relationship between the software's performance and the amount of resource used under stated conditions.

Maintainability: A set of attributes that bear on the effort needed to make specified modifications (which may include corrections, improvements, or adaptations of software to environmental changes and changes in the requirements and functional specifications).

Portability: A set of attributes that bear on the ability of software to be transferred from one environment to another (this includes the organizational, hardware or, software environment).

The ISO 9126 standard includes an example quality model, as shown in Figure 17.2, that further decomposes the quality characteristics into more concrete subcharacteristics. For example, the maintainability characteristic has been decomposed into four subcharacteristics, namely, *analyzability*, *changeability*, *stability*, and *testability*. The decomposition shown in Figure 17.2 is just a sample model—and not a universal one. The 20 subcharacteristics of Figure 17.2 are defined as follows:

Suitability: The capability of the software to provide an adequate set of functions for specified tasks and user objectives.

Accuracy: The capability of the software to provide the right or agreed-upon results or effects.

Interoperability: The capability of the software to interact with one or more specified systems.

Security: The capability of the software to prevent unintended access and resist deliberate attacks intended to gain unauthorized access to confidential information or to make unauthorized modifications to information or to the program so as to provide the attacker with some advantage or so as to deny service to legitimate users.

Maturity: The capability of the software to avoid failure as a result of faults in the software.

Fault Tolerance: The capability of the software to maintain a specified level of performance in case of software faults or of infringement of its specified interface.

Recoverability: The capability of the software to reestablish its level of performance and recover the data directly affected in the case of a failure.

Figure 17.2 ISO 9126 sample quality model refines standard's features into subcharacteristics. (From ref. 4. © 1996 IEEE.)

Understandability: The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.

Learnability: The capability of the software product to enable the user to learn its applications.

Operability: The capability of the software product to enable the user to operate and control it.

Attractiveness: The capability of the software product to be liked by the user.

Time Behavior: The capability of the software to provide appropriate response and processing times and throughput rates when performing its function under stated conditions.

Resource Utilization: The capability of the software to use appropriate resources in an appropriate time when the software performs its function under stated condition.

Analyzability: The capability of the software product to be diagnosed for deficiencies or causes of failures in the software or for the parts to be modified to be identified.

Changeability: The capability of the software product to enable a specified modification to be implemented.

Stability: The capability of the software to minimize unexpected effects from modifications of the software.

Testability: The capability of the software product to enable modified software to be validated.

Adaptability: The capability of the software to be modified for different specified environments without applying actions or means other than those provided for this purpose for the software considered.

Installability: The capability of the software to be installed in a specified environment.

Coexistence: The capability of the software to coexist with other independent software in a common environment sharing common resources.

Replaceability: The capability of the software to be used in place of other specified software in the environment of that software.

Organizations must define their own quality characteristics and subcharacteristics after a fuller understanding of their needs. In other words, organizations must identify the level of the different quality characteristics they need to satisfy within their context of software development. Reaching an ideally best quality level from the present one is a gradual process. Therefore, it is important to understand the need for moving on to the next achievable step toward the highest level—the ideally best level.

At this point it is useful to compare McCall's quality model with the ISO 9126 model. Since the two models focus on the same abstract entity, namely, *software quality*, it is natural that there are many similarities between the two models. What

is called *quality factor* in McCall's model is called *quality characteristic* in the ISO 9126 model. The following high-level quality factors/characteristics are found in both models: reliability, usability, efficiency, maintainability, and portability. However, there are several differences between the two models as explained in the following:

- The ISO 9126 model emphasizes characteristics *visible* to the users, whereas the McCall model considers *internal* qualities as well. For example, reusability is an internal characteristic of a product. Product developers strive to produce reusable components, whereas its impact is not perceived by customers.
- In McCall's model, one quality criterion can impact several quality factors, whereas in the ISO 9126 model, one subcharacteristic impacts exactly one quality characteristic.
- A high-level quality factor, such as testability, in the McCall model is a low-level subcharacteristic of maintainability in the ISO 9126 model.

Following are a few concerns with the quality models [4]:

- There is no consensus about what high-level quality factors are most important at the top level. McCall et al. suggest 11 high-level quality factors, whereas the ISO 9126 standard defines only 6 quality characteristics. Some of the quality factors in the McCall model are more important to developers. For example, reusability and interoperability are important to developers. However, the ISO 9126 model just considers the product.
- There is no consensus regarding what is a top-level quality factor/characteristic and what is a more concrete quality criterion/subcharacteristic. These days many applications run on computer and communications networks. However, interoperability is not an independent, top-level quality characteristic in the ISO 9126 model. It is not clear why interoperability is a part of functionality. The absence of a rationale makes it difficult to follow a prescribed quality model.

17.4 ISO 9000:2000 SOFTWARE QUALITY STANDARD

There are ongoing efforts at the international level for standardizing different aspects of computer communications and software development. Standardization has been particularly successful in the field of computer networking and wireless communications. For example, the collaborative work of the Internet Engineering Task Force (IETF) has been the key to the proliferation of the Internet. Similarly, standardization efforts from the IEEE have led to the successful development of the local area network (LAN) standard, namely the IEEE 802.3 standard, and the wireless local area network (WLAN) standards, namely IEEE 802.11a/b/g.

In spite of the positive consequence of standardization in the field of communications, standardization in software development is met with mixed reactions.

On the one hand, the main argument against standardization is that it curtails individual drive to be innovative. On the other hand, standards reduce the activity of reinventing the same, or similar, processes for development and quality assurance. Repeatability of processes is a key benefit emanating from standardization—and repeatability reduces the cost of software development and produces a base quality level of software products.

The ISO has developed a series of standards, collectively known as the ISO 9000. The ISO was founded in 1946, and it is based in Geneva, Switzerland. It develops and promotes international standards in the field of quality assurance and quality management. The ISO 9000 standards are generally applicable to all tangible products manufactured with human endeavor, say, from spices to software—Even some brands of spice and rice used in everyday cooking are claimed to be ISO 9000 certified. The ISO 9000 standards are reviewed and updated from time to time, once every 5–8 years. The latest ISO 9000 standards released in the year 2000 are referred to as ISO 9000:2000. There are three components of the ISO 9000:2000 standard as follows:

ISO 9000: Fundamentals and vocabulary [7]

ISO 9001: Requirements [8]

ISO 9004: Guidelines for performance improvements [9]

At this point we remind the reader that ISO 9002 and ISO 9003 were parts of ISO 9000:1994, but these are no longer parts of ISO 9000:2000. ISO 9002 dealt with the quality system model for quality assurance in production and installation, whereas ISO 9003 dealt with the quality system model for quality assurance in final inspection and testing.

17.4.1 ISO 9000:2000 Fundamentals

The ISO 9000:2000 standard is based on the following eight principles:

- **Principle 1. Customer Focus:** Success of an organization is highly dependent on satisfying the customers. An organization must understand its customers and their needs on a continued basis. Understanding the customers helps in understanding and meeting their requirements. It is not enough to just meet customer requirements. Rather, organizations must make an effort to exceed customer expectations. By understanding the customers, one can have a better understanding of their real needs and their unstated expectations. People in different departments of an organization, such as marketing, software development, testing, and customer support, must capture the same view of the customers and their requirements. An example of customer focus is to understand how they are going to use a system. By accurately understating how customers are going to use a system, one can produce a better *user profile*.
- **Principle 2. Leadership:** Leaders set the direction their organization should take, and they must effectively communicate this to all the people involved in the process. All the people in an organization must

have a coherent view of the organizational direction. Without a good understanding of the organizational direction, employees will find it difficult to know where they are heading. Leaders must set challenging but realistic goals and objectives. Employee contribution should be recognized by the leaders. Leaders create a positive environment and provide support for the employees to collectively realize the organizational goal. They reevaluate their goals on a continual basis and communicate the findings to the staff.

- **Principle 3. Involvement of People:** In general, organizations rely on people. People are informed of the organizational direction, and they are involved at all levels of decision making. People are given an opportunity to develop their strength and use their abilities. People are encouraged to be creative in performing their tasks.
- **Principle 4. Process Approach:** There are several advantages to performing major tasks by using the concept of *process*. A process is a sequence of activities that transform inputs to outputs. Organizations can prepare a plan in the form of allocating resources and scheduling the activities by making the process defined, repeatable, and measurable. Consequently, the organization becomes efficient and effective. Continuous improvement in processes leads to improvement in efficiency and effectiveness.
- **Principle 5. System Approach to Management:** A system is an interacting set of processes. A whole organization can be viewed as a *system* of interacting processes. In the context of software development, we can identify a number of processes. For example, gathering customer requirements for a project is a distinct process involving specialized skills. Similarly, designing a functional specification by taking the requirements as input is another distinct process. There are simultaneous and sequential processes being executed in an organization. At any time, people are involved in one or more processes. A process is affected by the outcome of some other processes, and, in turn, it affects some other processes in the organization. It is important to understand the overall goal of the organization and the individual subgoals associated with each process. For an organization as a whole to succeed in terms of effectiveness and efficiency, the interactions among processes must be identified and analyzed.
- **Principle 6. Continual Improvement:** Continual improvement means that the processes involved in developing, say, software products are reviewed on a periodic basis to identify where and how further improvements in the processes can be effected. Since no process can be a perfect one to begin with, continual improvement plays an important role in the success of organizations. Since there are independent changes in many areas, such as customer views and technologies, it is natural to review the processes and seek improvements. Continual process improvements result in lower cost of production and maintenance. Moreover, continual improvements lead to less differences between the expected behavior and actual behavior

of products. Organizations need to develop their own policies regarding when to start a process review and identify the goals of the review.

- **Principle 7. Factual Approach to Decision Making:** Decisions may be made based on facts, experience, and intuition. Facts can be gathered by using a sound measurement process. Identification and quantification of parameters are central to measurement. Once elements are quantified, it becomes easier to establish methods to measure those elements. There is a need for methods to validate the measured data and make the data available to those who need it. The measured data should be accurate and reliable. A quantitative measurement program helps organizations know how much improvement has been achieved due to a process improvement.
- **Principle 8. Mutually Beneficial Supplier Relationships:** Organizations rarely make all the components they use in their products. It is a common practice for organizations to procure components and subsystems from third parties. An organization must carefully choose the suppliers and make them aware of the organization's needs and expectations. The performance of the products procured from outside should be evaluated, and the need to improve their products and processes should be communicated to the suppliers. A mutually beneficial, cooperative relationship should be maintained with the suppliers.

17.4.2 ISO 9001:2000 Requirements

In this section, we will briefly describe five major parts of the ISO 9001:2000. For further details, we refer the reader to reference 8. The five major parts of the ISO 9001:2000, found in parts 4–8, are presented next.

Part 4: Systemic Requirements The concept of a *quality management system* (QMS) is the core of part 4 of the ISO 2001:2000 document. A quality management system is defined in terms of quality policy and quality objectives. In the software development context, an example of a quality policy is to review all work products by at least two skilled persons. Another quality policy is to execute all the test cases for at least two test cycles during system testing. Similarly, an example of a quality objective is to fix all defects causing a system to crash before release. Mechanisms are required to be defined in the form of processes to execute the quality policies and achieve the quality objectives. Moreover, mechanisms are required to be defined to improve the quality management system. Activities to realize quality policies and achieve quality objectives are defined in the form of *interacting quality processes*. For example, requirement review can be treated as a distinct process. Similarly, system-level testing is another process in the quality system. Interaction between the said processes occur because of the need to make all requirements testable and the need to verify that all requirements have indeed been adequately tested. Similarly, measurement and analysis are important processes in modern-day software development. Improvements in an existing QMS is achieved by defining a measurement and analysis process and identifying areas for improvements.

Documentation is an important part of a QMS. There is no QMS without proper documentation. A QMS must be properly documented by publishing a quality manual. The quality manual describes the quality policies and quality objectives. Procedures for executing the QMS are also documented. As a QMS evolves by incorporating improved policies and objectives, the documents must accordingly be controlled. A QMS document must facilitate effective and efficient planning, execution, and management of organizational processes. Records generated as a result of executing organizational processes are documented and published to show evidence that various ISO 9001:2000 requirements have been met. All process details and organizational process interactions are documented. Clear documentation is key to understanding how one process is influenced by another. The documentation part can be summarized as follows:

- Document the organizational policies and goals. Publish a vision of the organization.
- Document all quality processes and their interrelationship.
- Implement a mechanism to approve documents before they are distributed.
- Review and approve updated documents.
- Monitor documents coming from suppliers.
- Document the records showing that requirements have been met.
- Document a procedure to control the records.

Part 5: Management Requirements The concept of quality cannot be dealt with in bits and pieces by individual developers and test engineers. Rather, upper management must accept the fact that quality is an all-pervasive concept. Upper management must make an effort to see that the entire organization is aware of the quality policies and quality goals. This is achieved by defining and publishing a QMS and putting in place a mechanism for its continual improvement. The QMS of the organization must be supported by upper management with the right kind and quantity of resources. The following are some important activities for upper management to perform in this regard:

- Generate an awareness for quality to meet a variety of requirements, such as customer, regulatory, and statutory.
- Develop a QMS by identifying organizational policies and goals concerning quality, developing mechanisms to realize those policies and goals, and allocating resources for their implementations.
- Develop a mechanism for continual improvement of the QMS.
- Focus on customers by identifying and meeting their requirements in order to satisfy them.
- Develop a quality policy to meet the customers' needs, serve the organization itself, and make it evolvable with changes in the marketplace and new developments in technologies.

- Deal with the quality concept in a planned manner by ensuring that quality objectives are set at the organizational level, quality objectives support quality policy, and quality objectives are measurable.
- Clearly define individual responsibilities and authorities concerning the implementation of quality policies.
- Appoint a manager with the responsibility and authority to oversee the implementation of the organizational QMS. Such a position gives clear visibility of the organizational QMS to the outside world, namely, to the customers.
- Communicate the effectiveness of the QMS to the staff so that the staff is in a better position to conceive improvements in the existing QMS model.
- Periodically review the QMS to ensure that it is an effective one and it adequately meets the organizational policy and objectives to satisfy the customers. Based on the review results and changes in the marketplace and technologies, actions need to be taken to improve the model by setting better policies and higher goals.

Part 6: Resource Requirements Resources are key to achieving organizational policies and objectives. Statements of policies and objectives must be backed up with allocation of the right kind and quantity of resources. There are different kinds of resources, namely, staff, equipment, tool, financial, and building, to name the major ones. Typically, different resources are controlled by different divisions of an organization. In general, resources are allocated to projects on a need basis. Since every activity in an organization needs some kind of resources, the resource management processes interact with other kinds of processes. The important activities concerning resource management are as follows:

- Identify and provide resources required to support the organizational quality policy in order to realize the quality objectives. Here the key factor is to identify resources to be able to meet—and even exceed–customer expectations.
- Allocate quality personnel resources to projects. Here, the quality of personnel is defined in terms of education, training, experience, and skills.
- Put in place a mechanism to enhance the quality level of personnel. This can be achieved by defining an acceptable, lower level of competence. For personnel to be able to move up to the minimum acceptable level of competence, it is important to identify and support an effective training program. The effectiveness of the training program must be evaluated on a continual basis.
- Provide and maintain the means, such as office space, computing needs, equipment needs, and support services, for successful realization of the organizational QMS.
- Manage a work environment, including physical, social, psychological, and environmental factors, that is conducive to producing efficiency and effectiveness in "people" resources.

Part 7: Realization Requirements This part deals with processes that transform customer requirements into products. The reader may note that not much has changed from ISO 9001:1994 to ISO 9001:2000 in the realization part. The key elements of the realization part are as follows:

- Develop a plan to realize a product from its requirements. The important elements of such a plan are identification of the processes needed to develop a product, sequencing the processes, and controlling the processes. Product quality objectives and methods to control quality during development are identified during planning.
- To realize a product for a customer, much interaction with the customer is necessary to understand and capture the requirements. Capturing requirements for a product involves identifying different categories of requirements, such as requirements generated by the customers, requirements necessitated by the product's use, requirements imposed by external agencies, and requirements deemed to be useful to the organization itself.
- Review the customers' requirements before committing to the project. Requirements that are not likely to be met should be rejected in this phase. Moreover, develop a process for communicating with the customers. It is important to involve the customers in all phases of product development.
- Once requirements are reviewed and accepted, product *design* and *development* take place:

  Product design and development start with planning: Identify the stages of design and development, assign various responsibilities and authorities, manage interactions between different groups, and update the plan as changes occur.

  Specify and review the *inputs* for product design and development.

  Create and approve the *outputs* of product design and development. Use the outputs to control product quality.

  Periodically review the outputs of design and development to ensure that progress is being made.

  Perform design and development verifications on their outputs.

  Perform design and development validations.

  Manage the changes effected to design and development: Identify the changes, record the changes, review the changes, verify the changes, validate the changes, and approve the changes.

- Follow a defined purchasing process by evaluating potential suppliers based on a number of factors, such as ability to meet requirements and price, and verify that a purchased product meets its requirements.
- Put in place a mechanism and infrastructure for controlling production. This includes procedures for validating production processes, procedures for identifying and tracking both concrete and abstract items, procedures

for protecting properties supplied by outside parties, and procedures for preserving organizational components and products.

- Identify the monitoring and measuring needs and select appropriate devices to perform those tasks. It is important to calibrate and maintain those devices. Finally, use those devices to gather useful data to know that the products meet the requirements.

Part 8: Remedial Requirements This part is concerned with measurement, analysis of measured data, and continual improvement. Measurement of performance indicators of processes allows one to determine how well a process is performing. If it is observed that a process is performing below the desired level, then corrective action can be taken to improve the performance of the process. Consider the following example. We find out the sources of defects during system-level testing and count, for example, those introduced in the design phase. If too many defects are found to be introduced in the design phase, actions are required to be taken to reduce the defect count. For instance, an alternative design review technique can be introduced to catch the defects in the design phase. In the absence of measurement it is difficult to make an objective decision concerning process improvement. Thus, measurement is an important activity in an engineering discipline. Part 8 of the ISO 9001:2000 addresses a wide range of performance measurement needs as explained in the following:

- The success of an organization is largely determined by the satisfaction of its customers. Thus, the standard requires organizations to develop methods and procedures for measuring and tracking the customer's satisfaction level on an ongoing basis. For example, the number of calls to the help line of an organization can be considered as a measure of customer satisfaction—too many calls is a measure of less customer satisfaction.
- An organization needs to plan and perform internal audits on a regular basis to track the status of the organizational QMS. An example of an internal audit is to find out whether or not personnel with adequate education, experience, and skill have been assigned to a project. An internal audit needs to be conducted by independent auditors using a documented procedure. Corrective measures are expected to be taken to address any deficiency discovered by the auditors.
- The standard requires that both processes, including QMS processes, and products be monitored using a set of key performance indicators. An example of measuring product characteristics is to verify whether or not a product meets its requirements. Similarly, an example of measuring process characteristics is to determine the level of modularity of a software system.
- As a result of measuring product characteristics, it may be discovered that a product does not meet its requirements. Organizations need to ensure that such products are not released to the customers. The causes of the differences between an expected product and the real one need to be identified.

- The standard requires that the data collected in the measurement processes are analyzed for making objective decisions. Data analysis is performed to determine the effectiveness of the QMS, impact of changes made to the QMS, level of customer satisfaction, conformance of products to their requirements, and performance of products and suppliers.
- We expect that products have defects, since manufacturing processes may not be perfect. However, once it is known that there are defects in products caused by deficiencies in the processes used, efforts must be made to improve the processes. Process improvement includes both corrective actions and preventive actions to improve the quality of products.

17.5 SUMMARY

The concept of software quality is very broad, and, therefore, it is useful to look at it from different viewpoints. In 1984, Garvin [1] analyzed the concept of quality—of products in general, not specifically software products—from five viewpoints, namely, *transcendental view*, *user view*, *manufacturing view*, *product view*, and *value-based view*. In the transcendental view, quality is something that can be perceived only through experience. The user's view concerns to what extent the product meets user needs and expectations. According to the manufacturing view, quality is perceived through conformance to a manufacturing process. According to the product view, good internal qualities of a product translates into good external qualities of products. The value-based view concerns how much a user is willing to pay for a certain level of quality.

It is useful to measure quality for three reasons. First, measurement allows us to develop baselines for quality. Second, since quality improvement has an associated cost, it is important to know how much quality improvement is achieved for a certain cost. Finally, it is useful to know the present level of quality so that further improvement can be planned.

Gilb's technique [5] for quality measurement, stated in the following, is useful in measuring quality factors which are not amenable to direct measurement: *The quality concept is successively broken down into its component parts until each component can be expressed in terms of some directly measurable attributes*.

Measuring the manufacturing view is easier. The two widely used metrics are *defect count* and *rework cost*. The first metric refers to how many defects have been detected, whereas the second metric refers to how much it costs to fix the known defects. A defect count can be analyzed to give us a better idea about the development process. For example, each defect can be traced to a phase of the software development process where the defect got introduced. Improvements should be made to the phases where a large fraction of the defects get introduced. Similarly, improvements should be made to the phases where critical defects are introduced. A second example of analyzing defect count is to identify the modules containing a large number of defects. A third example of analyzing defect count is separating defects detected during development from those detected

during operation. If a large number of defects are found during operation, one can conclude that the test process needs much improvement. A fourth example of using defect count is to be able to compare different modules in terms of *defect density*.

Similarly, rework cost can be analyzed in two parts: prerelease rework cost and postrelease rework cost. The prerelease rework cost is a measure of development efficiency, whereas the postrelease rework cost is a measure of the delivered quality of the system.

Another concept of software quality, commonly known as McCall's quality factors, was proposed by McCall, Richards, and Walters in 1977 [6]. They studied the concept of software quality in terms of *quality factors* and *quality criteria*. A quality criterion is an *attribute* of a quality factor. McCall et al. identified 23 quality criteria. Some examples of quality criteria are modularity, traceability, simplicity, and completeness. In a nutshell, a quality factor represents a behavioral characteristic of a system, and McCall et al. suggested 11 quality factors: correctness, reliability, efficiency, integrity, usability, maintainability, testability, flexibility, portability, reusability, and interoperability. The 11 quality factors are categorized into three classes: product operation, product revision, and product transition. Product operation concerns correctness, reliability, efficiency, integrity, and usability. Product revision concerns maintainability, testability, and flexibility. Product transition concerns portability, reusability, and interoperability.

A global initiative for understanding the concept of software quality has been performed by experts around the world. Their collaborative effort has led to the standardization of the quality concept by the ISO in the form of documents ISO 9126 and ISO 9000:2000. The document ISO 9126 is about quality characteristics, whereas the ISO 9000:2000 document is a quality assurance standard. What is called quality factor in the McCall model is called *quality characteristic* in the ISO 9126 model. However, there are several differences between the two models. The ISO 9126 model focuses on characteristics *visible* to the users, whereas the McCall model emphasizes *internal* quality as well.

It is useful to note a few concerns about the quality models discussed in this chapter. There is no consensus about what high-level quality factors are most important at the top level. Moreover, there is no consensus about what is a top-level quality factor and what is a low-level, that is, concrete, quality attribute/subcharacteristic.

There are three components of the ISO 9000:2000 standard, namely, ISO 9000, ISO 9001, and ISO 9004. The ISO 9000 document is about fundamentals and vocabulary, the ISO 9001 document is about requirements, and the ISO 9004 document contains guidelines for performance improvements.

The ISO 9000 document is based on eight principles: customer focus, leadership, involvement of people, process approach, system approach to management, continual improvement, factual approach to decision making, and mutually beneficial supplier relationships. The ISO 9001 document deals with the following five kinds of requirements: systemic requirements, management requirements, resource requirements, realization requirements, and remedial requirements.

LITERATURE REVIEW

Several books and articles have been written about software quality assurance and quality models. The January 1996 issue of *IEEE Software* was devoted to software quality. The special issue presents a few papers addressing the following topics: contribution of standards to product quality, quality outcomes in terms of business value, support for quality-based design and inspection, software quality in consumer electronics, and a case study of early quality prediction in telecommunications systems.

In the book *Inroads to Software Quality* [10], Jarvis and V Crandall focus on two approaches to quality assurance. The first approach is to use quality-oriented methods such as total quality management, ISO 9000, and SEI CMM levels. Their second approach is based on the idea of building quality into deliverables at the outset and throughout product development.

In the book *Software Quality Assurance: Principles and Practice* [11], Godbole presents quality assurance principles and practice in a comprehensive manner. The book puts much emphasis on planning for quality assurance, product quality and process quality, software testing, inspection and walkthrough, and quality improvement models, such as the ISO 9000 and CMM models.

In the book *Software Quality Assurance* [12], Galin presents a comprehensive overview of software quality and software quality assurance (SQA). Additional topics covered in Galin's book are different components of a software quality assurance system, such as SQA architecture, preproject software quality components, SQA components in the project life cycle, software quality infrastructure components, and management components of software quality. For example, contract review and development and quality plans are preproject software quality components. Similarly, reviews and software testing strategies are project life-cycle components. Examples of infrastructure components are procedures and work instructions and staff training. Some management components of software quality are project progress control and software quality metrics. An example component of standards, certification, and evaluation is quality management using ISO 9000 and CMM.

REFERENCES

1. D. A. Garvin. What Does "Product Quality" Really Mean? *Sloan Management Review*, Fall 1984, pp. 25–43.
2. M. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber. Capability Maturity Model, Version 1.1. *IEEE Software*, July 1993, pp. 18–27.
3. M. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber. How ISO 9001 Compares with the CMM. *IEEE Software*, January 1995, pp. 74–83.
4. B. Kitchenham and S. L. Pfleeger. Software Quality: The Elusive Target. *IEEE Software*, January 1996, pp. 12–21.
5. T. Gilb. *Principles of Software Engineering Management*. Addison-Wesley, Reading, MA, 1987.
6. J. A. McCall, P. K. Richards, and G. F. Walters. *Factors in Software Quality*, Vol. 1, ADA 049014. National Technical Information Service, >Springfield, VA, 1977.
7. International Organization for Standardization (ISO). Quality management systems—fundamentals and Vocabulary, ISO 9000:2000. ISO, Geneva, December 2000.

8. International Organization for Standardization (ISO). Quality Management Systems—Requirements, ISO 9001:2000. ISO, Geneva, December 2000.
9. International Organization for Standardization (ISO). Quality Management Systems–Guidelines for Performance Improvements, ISO 9004:2000. ISO, Geneva, December 2000.
10. A. Jarvis and V. Crandall. *Inroads to Software Quality*. Prentice-Hall, Englewood Cliffs, NJ, 1997.
11. N. S. Godbole. *Software Quality Assurance*. Alpha Science International, Pangbourne, UK, 2004.
12. D. Galin. *Software Quality Assurance*. Addison-Wesley, Reading, MA, 2005.

Exercises

1. Briefly explain the five different views of software quality.
2. Briefly explain how one can measure the user's view of software quality.
3. Briefly explain how one can measure the manufacturer's view of software quality.
4. Briefly explain McCall's quality factors and quality criteria.
5. Briefly explain McCall's categorization of quality factors into three quality criteria.
6. Briefly explain the ISO 9126 quality characteristics.
7. Compare McCall's quality model with the ISO 9126 quality model.
8. State some difficulties in applying the McCall and ISO 9126 quality models.
9. Briefly explain the ISO 9000:2000 (Fundamental) document for quality assurance.
10. Briefly explain the ISO 9001:2000 (Requirements) document for quality assurance.

CHAPTER 18

Maturity Models

> We are the product of 4.5 billion years of fortuitous, slow biological evolution. There is no reason to think that the evolutionary process has stopped. Man is a transitional animal. He is not the climax of creation.
> — *Carl Sagan*

18.1 BASIC IDEA IN SOFTWARE PROCESS

The concept of a *process* plays an important role in today's software development. Without a repeatable process, the only repeatable results you are likely to produce are errors [1]. In the software engineering context, a process comprises a set of activities that are executed to develop software products. The activities find expressions in the form of methods, techniques, strategies, procedures, and practices. Those activities heavily rely on information repositories such as documents, standards, and policies. Different processes are driven by different goals and availability of resources. The concept of a process is applied not just to develop source code, but to other software-related products, such as a project plan, requirements document, design document, and user manual, as well. It is effective to follow a defined process because of the following benefits:

- The process can be repeated in subsequent projects.
- The process can be evaluated by using a variety of metrics, such as cost, quality, and time to deliver a product.
- Actions can be taken to improve the process to achieve better results.

A process for software development consists of several different processes for its different component tasks. For example, gathering the requirements of a system, constructing a functional specification of a system, designing a system, testing a system, and maintaining a system are different tasks to be performed in the life cycle of a software system. Those different tasks, which are quite different from each

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

other, need different processes to be followed. A requirements gathering process is fundamentally different from a testing process. Similarly, a design process is very different from a testing process. However, some similarities between a maintenance process and a testing process cannot be ruled out because a maintenance task may involve significant changes to the design of the system as well as rigorous testing.

Software testing is treated as a distinct process because it involves a variety of unique activities, techniques, strategies, and policies as explained in the following:

- Testing is performed to achieve the following objectives:

  Reveal defects in software products

  Show the extent to which a software system possesses different quality attributes, such as reliability, performance, stability, and scalability
- Testing begins almost at the same time as a project is conceptualized, and it lasts as long as the system is in existence.
- Testing is carried out by people with differing responsibilities in different phases of software development. Those different levels of testing are commonly known as unit testing, integration testing, system testing, and acceptance testing.
- A number of techniques can be applied at each level of testing for generating test cases.
- A number of different testing strategies can be applied at different levels of testing.
- A number of metrics can be monitored to gauge the progress of testing.
- A number of metrics can be monitored to accurately reflect the different quality levels of a system.
- Testing is influenced by organizational policies, such as the level of quality to be achieved in a product, the fraction of the total budget for a project to be allocated to testing, and the duration of testing.
- Testing can be performed in a combination of manual and automated modes of execution of test cases.

To be able to improve a defined process, organizations need to evaluate its capabilities and limitations. For example, the *capability maturity model* (CMM), developed by the Software Engineering Institute (SEI) at Carnegie Mellon University, allows an organization to evaluate its software development processes. The model supports incremental process improvement. For an organization, it is more practical to develop a continuously improving process than find one which is "just right." By recognizing the importance of testing as a distinct process—much different from an overall process for software development—a separate model, known as a *testing maturity model* (TMM), has been developed to evaluate a testing process. In addition, for an organization to be able to improve its testing process, the *test process improvement* (TPI) model has been developed.

18.2 CAPABILITY MATURITY MODEL

In software development processes we seek three desirable attributes as follows:

1. The products are of the highest quality. Ideally, a product should be free of defects. However, in practice, a small number of defects with less severe consequences are generally tolerated.
2. Projects are completed according to their plans, including the schedules.
3. Projects are completed within the allocated budgets.

However, developers of large software products rarely achieve the above three attributes. Due to the complex nature of software systems, products are released with known and unknown defects. The unknown defects manifest themselves as unexpected failures during field operation, that is, when we know that the system was released with unknown defects. In many organizations, software projects are often late and over budget. Due to the business angle in software development, it is important for the survival of organizations to develop low-cost, high-quality products within a short time. In order to move toward that goal, researchers and developers are devising new techniques and tools. However, introduction of new techniques and tools into a process must be carefully planned to effect improvements in products.

While awarding a contract to an organization for a software product, the customer needs to gain confidence that the organization is capable of delivering the desired product. Such confidence can be gained by evaluating the capabilities of the organizations. The U.S. Department of Defense being a large customer of software systems, it wanted to evaluate the capabilities of its software contractors. The Department of Defense wanted to have a framework to evaluate the maturity of software processes used by organizations. In circa 1986, the SEI at initiated the development of a framework to evaluate process maturity.

The maturity level of a development process tells us to what extent the organization is capable of producing low-cost, high-quality software. Therefore, the evaluation framework is the CMM. After evaluating the current maturity level of a development process, organizations can work on improving the process to achieve the next higher level of process maturity. In the CMM framework, a process has five maturity levels. Before going into the details of different maturity levels, it is useful to have a glimpse of an immature process—an organization can be considered to be immature if it follows immature processes [2].

On the one hand, an immature organization may not have a defined process, and, even if there is one, the organization may not follow it. Developers and managers react to problems when they occur, rather than take preventive measures to eliminate them or reduce the frequency of their occurrences. In other words, product and process problems are resolved in an ad hoc manner. Estimates of cost, schedule, and quality are highly inaccurate due to the absence of a measurement program to gather process data. Hence, projects overrun cost and time estimates by a large factor. There is no measurement program to evaluate product or process quality.

On the other hand, a mature organization carries out its activities in a planned manner. Both process and product characteristics are measured to keep track of progress and quality of products. Estimates are more accurate due to following a rigorous measurement program. Employees are kept abreast of new developments through training. Continual effort is made to improve the quality of products while bringing down costs and lead times. Defined processes are continually updated to take advantage of new techniques, tools, and experience from past projects. As an organization becomes more and more mature, standards and organizational policies play key roles in product development. Organizations become mature in an incremental manner; that is, processes are improved in an evolutionary approach, rather than effecting drastic changes.

As an organization moves from one level to the next, its process capability improves to produce better quality software at a lower cost. The CMM defines five distinct levels of maturity, where level 1 is the initial level and level 5 is the highest level of process maturity.

18.2.1 CMM Architecture

First we explain the concept of a *maturity level* using Figure 18.1 followed by a detailed description of individual levels. Figure 18.1 can be read as follows:

- A maturity level indicates process capability and contains key process areas (KPAs). The KPAs for each level are listed in Figure 18.2.

Figure 18.1 CMM structure. (From ref. 3. © 2005 John Wiley & Sons.)

Figure 18.2 SW-CMM maturity levels. (From ref. 3. © 2005 John Wiley & Sons.)

- Key process areas are expected to achieve goals and are organized by common features.
- Common features contain key practices and address implementation or institutionalization of the key practices.
- Key practices describe infrastructure or activities.
- When the key practices are followed, the goals of the KPAs are expected to be achieved.
- A maturity level is reached by meeting all the goals of all the KPAs at that level.

18.2.2 Five Levels of Maturity and Key Process Areas

The five levels of process maturity and their KPAs are explained as follows. The KPAs for each maturity level are listed in Figure 18.2.

Level 1: Initial At this level, software is developed by following no process model. There is not much planning involved. Even if a plan is prepared, it may not be followed. Individuals make decisions based on their own capabilities and skills. There is no KPA associated with level 1. An organization reaches level 1 without making any effort.

Level 2: Repeatable At this level, the concept of a process exists so that success can be repeated for similar projects. Performance of the proven activities of past projects is used to prepare plans for future projects. This level can be summarized as being *disciplined* because processes are used for repeatability. All the processes are under the effective control of a project management system. The KPAs at level 2 are as follows:

- **Requirements Management:** It is important to establish a common understanding between the customer and developers. Details of a project, such as planning and management, are guided by a common view of customer requirements.
- **Software Project Planning:** This means creating and following a reasonable plan for realizing and managing a project.
- **Software Project Tracking and Oversight:** This means making the progress of a project visible so that management is aware of the status of the project. Corrective actions can be taken if the actual progress of a project significantly deviates from the planned progress.
- **Software Subcontract Management:** This means evaluating, selecting, and managing suppliers or subcontractors.
- **Software Quality Assurance:** This means evaluating processes and products to understand their effectiveness and quality.
- **Software Configuration Management:** This means ensuring the integrity of the products of a project as long as the project continues to exist.

Level 3: Defined At this level, documentation plays a key role. Processes that are related to project management and software development activities are documented, reviewed, standardized, and integrated with organizational processes. In other words, there is organizationwide acceptance of standard processes. Software development is carried out by following an approved process. Functionalities and the associated qualities are tracked. Cost and schedule are monitored to keep them under control. The KPAs at level 3 are as follows:

- **Organization Process Focus:** This means putting in place an organizationwide role and responsibility to ensure that activities concerning process improvement are in fact followed.
- **Organization Process Definition:** Certain practices are useful irrespective of projects. Thus, it is important to identify and document those practices.
- **Training Program:** Individuals need to be trained on an on-going basis to make them knowledgeable in application domains and new developments in software techniques and tools. Training is expected to make them effective and efficient.
- **Integrated Software Management:** This means integrating an organization's software engineering and management activities into a common, defined process. Integration is based on commercial and technological needs of individual projects.

- **Software Product Engineering:** This means following a defined process in a consistent manner by integrating the technical activities to produce software with desired attributes. The activities include requirements elicitation, functional design, detailed design, coding, and testing.
- **Intergroup Coordination:** This means that a software development group coordinates with other groups, such as customers, the marketing group, the and software quality assurance (SQA) group, to understand their needs and expectations.
- **Peer Review:** Work products, such as requirements, design, and code, are reviewed by peers to find defects at an early stage. Peer reviews can be performed by means of inspection and walkthrough.

Level 4: Managed At this level, *metrics* play a key role. Metrics concerning processes and products are collected and analyzed. Those metrics are used to gain quantitative insight into both process and product qualities. When the metrics show that limits are being exceeded, corrective actions are triggered. For example, if too many test cases fail during system testing, it is useful to start a process of root cause analysis to understand why so many tests are failing. The KPAs at level 4 are as follows:

- **Quantitative Process Management:** Process data indicate how well a process is performing. If a process does not perform as expected, the process is improved by considering the measured data.
- **Software Quality Management:** The quality attributes of products are measured in quantitative form to have a better insight into the processes and products. Improvements are incorporated into the processes, and their effectiveness is evaluated by measuring product quality attributes.

Level 5: Optimizing At this level, organizations strive to improve their processes on a continual basis. This is achieved in two steps: (i) Observe the effects of processes, by measuring a few key metrics, on the quality, cost, and lead time of software products and (ii) effect changes to the processes by introducing new techniques, methods, tools, and strategies. The following are the KPAs at level 5:

- **Defect Prevention:** This means analyzing the root causes of different classes of defects and taking preventive measures to ensure that similar defects do not recur.
- **Technology Change Management:** This means identifying useful techniques, tools, and methodologies and gradually introducing those into software processes. The key idea is to take advantage of new developments in technologies.
- **Process Change Management:** This means improving an organization's processes to have a positive impact on quality, productivity, and development time.

18.2.3 Common Features of Key Practices

The key practices in every KPA are organized into five categories called *common features* as explained in the following. Common features are attributes of key practices that indicate whether the implementation or institutionalization of a KPA is effective, repeatable, and lasting.

- **Commitment to Perform:** An organization must show in action—not merely in words—that it is committed to process improvement. Upper management actions, such as establishing organizational policies and allocating resources to process improvement activities, give evidence that an organization is committed to perform in a better way. For example, management can formulate a policy to use established and enduring processes.
- **Ability to Perform:** The ability of an organization to realize a process in a competent manner is determined by the organization's structure, resources, and people to execute the process. Availability of adequately trained personnel and their attitude to change have a positive impact on their ability to perform.
- **Activities Performed:** These describe what needs to be implemented to establish the capability of a process. They include the roles and procedures required to realize KPAs. Typical activities performed to realize KPAs are making plans, executing the plans, tracking the work, and taking corrective actions to ensure that work stays close to the plan.
- **Measurement and Analysis:** Measurement is key to knowing the current status and effectiveness of a process. Measurement involves gathering data that can reveal the progress and effectiveness of processes. The gathered data must be analyzed to have an insight into the processes. Measurement and analysis must be performed to be able to take corrective actions.
- **Verifying Implementation:** After having policies and processes in place, it is necessary that activities be performed by complying with the standard process. Compliance with the standard process can be checked by performing frequent audits and reviews.

18.2.4 Application of CMM

For an organization to be at a certain level of maturity, all the goals of all the KPAs at that level—and all preceding levels too—must be satisfied. For example, for an organization to be at level 2, all six KPAs associated with level 2 must be satisfied. For an organization to be at level 3, the organization must meet all six KPAs associated with level 2 and all seven KPAs associated with level 3. The SEI provides two methodologies to evaluate the current capabilities of organizations: *internal assessments* and *external evaluations*. The SEI developed the *capability maturity model–based assessment internal process improvement* (CBA-IPI) to assist organizations for self-assessment. The CBA-IPI uses the CMM as a reference model to evaluate the process capability of organizations by identifying what KPAs are being satisfied and what need to be improved.

The SEI developed the CMM appraisal framework (CAF) to provide a mechanism for formal evaluation of organizations. The CAF describes the *requirements* and *guidelines* to be used by external assessors in designing CAF-compliant evaluation methods.

18.2.5 Capability Maturity Model Integration (CMMI)

After the development and successful application of the CMM in the software area, which is known as software CMM (CMM-SW), CMMs in other areas were developed as well.

The CMM for software, known as CMM-SW, was first released in 1991 as CMM-SW version 1.0, followed by CMM-SW version 1.1 in 1993. After its first release, many software organizations used it for self and external evaluations. The success of CMM-SW led to the development of CMM in other areas. Thus, the concept of CMM is not software specific. In order to appreciate the wide applicability of CMM, we remind the reader of the following: *A CMM is a reference model of mature practices in a specific discipline used to apprise and improve a group's capability to perform that discipline*. To name a few, there are several CMMs as follows:

- Software CMM
- Systems engineering CMM
- Integrated product development CMM
- Electronic industry alliance 731 CMM
- Software acquisition CMM
- People CMM
- Supplier source CMM

It is apparent from the above examples of the CMM that they are likely to have different characteristics. The CMMs differ in three ways, namely, *discipline*, *structure*, and *definition of maturity*. First, different CMMs are applied to different disciplines, such as software development, system engineering, software acquisition, and people. Second, improvements can be made continuously or in distinct stages. Finally, the definition of maturity is dependent upon the entity under consideration. It is obvious that *people* and *software acquisition* mature in different ways, and their maturity is defined in different ways.

As more than one CMM was applied in an organization, a number of problems surfaced because of the following reasons:

- Different models have different structures, different ways of measuring maturity, and different terms.
- It was difficult to integrate the different CMMs to achieve the common organizational goal which is to produce low-cost, high-quality products within schedule.
- It was difficult to use many models in supplier selection and subcontracting.

Therefore, a pressing need was felt to have a unified view of process improvement throughout an organization. Thus, evolved the idea of CMMI. The CMMI includes information from the following models:

- Capability maturity model for software (CMM-SW)
- Integrated product development capability maturity model (IPD-CMM)
- Capability maturity model for systems engineering (CMM-SE)
- Capability maturity model for supplier sourcing (CMM-SS)

The usefulness of the CMMI is readily recognized by considering the following facts. First, today's complex software systems are often built by using some subsystems developed by other parties. For example, a security module may be purchased from another vendor. Similarly, a communication module may be obtained from a vendor specializing in communication systems. There is a need for evaluating the maturity of the suppliers. Second, large systems often contain a number of diverse components, such as databases, communications, security, and real-time processing. Coexistence and interoperability of the diverse systems, which might have been developed by different vendors, are paramount to the successful operation of larger systems. Consequently, it is important to evaluate the maturity level of an integrated product development process. Third, in general, complex software systems often need to run on specialized execution platforms. For example, Internet routing software runs on specialized hardware and a specialized operating system, rather than the commonly used hardware platforms and operating systems. Those kinds of software need to be developed in the right system context.

18.3 TEST PROCESS IMPROVEMENT

A *test process* is a certain way of performing activities related to defect detection. Some typically desired test activities in software development are as follows:

- Identifying test goals
- Preparing a test plan
- Identifying different kinds of tests
- Hiring test personnel
- Designing test cases
- Setting up test benches
- Procuring test tools
- Assigning test cases to test engineers
- Prioritizing test cases for execution
- Organizing the execution of test cases into multiple test cycles
- Preparing a schedule for executing test cases
- Executing test cases
- Reporting defects

- Tracking defects while resolving them
- Measuring the progress of testing
- Measuring the quality attributes of the software under test
- Evaluating the effectiveness of a test process
- Identifying steps to improve the effectiveness of test activities
- Identifying steps to reduce the cost of testing

The nature of activities in a test process is wide ranging. On the one hand, a test activity can be a simple, straightforward activity such as reporting defects. On the other hand, prioritizing test cases for execution is a nontrivial task. A more complex task is to organize test execution into a few testing cycles by considering all test cases, or a subset thereof, in each test cycle. Similarly, identifying steps to improve the effectiveness of test activities is a nontrivial task.

Now we present a simple example of a test process for system-level testing. This test process is presented in the following steps:

- Categorize the features of a system into k categories.
- Design N_i number of test cases for feature category i for $1 \leq i \leq k$. The number N_i of test cases is denoted by the set T_i.
- Run $T_1 \cup \cdots \cup T_k$ to detect defects.
- Run $T_1 \cup \cdots \cup T_k$ after each defect fix until no more defects are found or it is time to release the system.

One can easily identify a number of deficiencies in the above test process as follows:

- Test tools have not been used.
- Test cases have not been prioritized.
- The entire test suite has been executed in each test cycle, rather than its selected subsets in the later test cycles.

Therefore, it is important to improve test processes, and it is advantageous to follow a defined model for their improvement. The idea of improving test processes by following a model, namely the TPI model, was first studied by Tim Koomen and Martin Pol. They have extensively discussed the TPI model in a well-written book [4]. A test process needs to be improved for three reasons as explained in the following:

- **Quality:** A better test process should give more insight into the quality characteristics of a system being tested. For example, after running all the test cases twice, we should have a good idea about the reliability level of the system. Here, the intention is not to have a better quality product—though that is always desirable. Rather, the intention is to have an improved test process that gives us better insight into the quality of the system being tested.
- **Lead Time:** A better test process saves testing time, and thereby gives more time to other areas of system development. For example, one can use

the idea of prioritizing the execution of test cases so that difficult-to-fix defects are detected as early as possible. Early detection of difficult-to-fix defects gives more time to the developers to address those defects, thereby shortening the development time.

- **Cost:** A better test process is expected to be carried out with a lower cost and thereby reduces the overall cost of system development. For example, one can judiciously select a subset of the test cases, rather than the entire test suite, for regression testing toward the closing stage of system-level testing during development. By carefully choosing a subset of the test suite, we can reduce the cost of system testing without compromising its effectiveness. Test cases can have a positive impact on development cost by influencing the processes of requirements gathering and software design. For example, while gathering software requirements we can ask the following questions:

  1. Are the requirements directly testable?
  2. If some requirements are not directly testable, can some other requirements be incorporated so that the original requirements are easily testable?

  Therefore, by considering the testability of requirements, it is possible to reduce the development cost. If a requirement is not testable, it is better to identify those requirements and discuss them with the customers early in the development phase. Essentially, what is being done is validation of requirements with test cases much before those requirements become a part of a design—or even a contract between a development organization and a customer.

It is apparent that a better test process gives us more insight into the quality attributes of a system, and it contributes to the development of software products at a lower *cost* in less time. An intuitive approach to improving a test process is as follows:

Step 1. Determine Area for Improvement: We identify an area of testing where we would like to see immediate improvements. For guidelines, we look at the three major areas for improvement, namely, quality, time, and cost, and focus on one precise aspect. Suppose that we want to have a good idea about the performance quality of the system as early as possible.

Step 2. Evaluate Current State of Test Process: It is important to know where we stand with respect to the chosen area for improvement. This is because it is likely to influence where we go next and how to achieve that. Referring to the example of having a good idea about the performance of the system as early as possible, we evaluate *when* the current test process lets us know about the system performance. Such an evaluation can be done by identifying the time when performance-related test cases are executed. If such test cases are executed toward the closing phase of system testing, there is a possibility that such tests are performed earlier.

Step 3. Identify Next Desired State and Means: Once we know the current state of the testing process with respect to the desired area for improvement, we are in a better position to evaluate the extent of possible improvement. One must carefully identify the amount of improvement to be seen. Too much improvement in one state may require significant changes in the test process. Therefore, incremental improvements are preferred. Sometimes, a certain improvement may call for improvements in other, unselected areas. The type and extent of improvements one wants to effect must be carefully selected. For example, one can evaluate the performance of a system under different execution environments, such as a lightly loaded system, moderately loaded system, and heavily loaded system. Therefore, if we are interested in the early performance information in a lightly loaded environment, we can prioritize the performance-related tests to be executed before performing load tests. In the above example, *prioritization* of test execution is an example of the means necessary to effect an improvement.

Step 4. Implement Necessary Changes to Process: Improvements, as identified above, are effected in the test process in a planned manner. Then, the effectiveness of the new test process is evaluated to verify whether or not the desired results have been obtained.

Though the steps of the above test process improvement model appear to be straightforward, their implementation is not. The steps, especially the second and third, are too general, and therefore it is difficult to achieve an agreement among test engineers. Consequently, there is a need for a reference model that can serve as a basis for test improvement. The TPI model of Koomen and Pol encourages and supports gradual improvement in a test process step by step. Too many changes in a test process may not be successful for two reasons. First, too many changes are likely to call for too much resources, which may not be available in one state. Second, people generally resist too many changes being effected at one time.

An important concept revealed in the intuitive test process improvement model of Section 18.3 is the idea of evaluating the *current status* of a test process. The current status of a test process is evaluated from different viewpoints, known as *key areas*—and 20 key areas have been identified. For example, use of *test tools* is a key area in a test process. Similarly, use of *metrics* is another key area. The status of a test process with respect to a key area is represented in terms of one of four *levels* of maturity—A, B, C, and D. The maturity level of a test process with respect to a key area ascends from A to D. In the following, we briefly describe the 20 key areas:

1. *Test Strategy*: A test process needs to detect critical defects early in the testing process. This, for example, can be achieved by prioritizing the execution of test cases. The test strategy needs to find defects at a lower cost. The test strategy should allow us to accurately map requirements with test cases for traceability purpose.

2. *Life-Cycle Model*: A test process can be a complex one involving many smallers activities and many persons and lasting for several months. Therefore, for micromanagement purposes, a test process can be viewed as having a clearly defined life cycle with distinct phases, such as planning, preparation, and execution. Each phase is characterized by a set of activities. For example, scheduling and cost estimation are planning activities. Similarly, designing test cases and setting up test benches are preparation activities. Each activity needs to have a clear structure in the form of objective, input, assumptions, process, output, constraints, dependencies, and tool support. A life-cycle model allows us to have a repeatable, and hence predictable, test process. Repeatability is a key benefit of a life-cycle model.
3. *Moment of Involvement*: Actual execution of test cases cannot start until code is written. However, the testing process can begin at any moment. For example, ideally, planning for testing should begin concurrently with requirements elicitation. This will allow us to verify the testability of each requirement. Sooner and closer interactions between developers and SQA personnel lead to better understanding of testing needs in terms of resources and challenges.
4. *Planning and Estimating*: Planning involves identifying and scheduling the activities to be performed, identifying the constraints, and identifying the resources. To prepare for the planned activities, it is important to be able to accurately estimate the quantity of resources needed. In the absence of a good plan and estimation, a project may run out of time and overrun budget. Thus, a test project needs to be carefully planned and its cost accurately estimated as early as possible.
5. *Test Specification Technique*: A test specification technique is one that allows us to derive test cases from an information source, such as the requirement specification of a system, design documents of a system, and source code of a system. Since there are multiple information sources of a software system, there is a need to identify multiple test specification techniques. Analysis of a test specification technique will tell us what kinds of test cases it will derive. In other words, by studying a test specification technique we will have a better insight into the quality and coverage of testing.
6. *Static Test Technique*: In a broad sense, there are two kinds of testing activities: *static* and *dynamic*. Dynamic testing involves actual execution of code, whereas static testing involves a variety of reviews, such as requirement verification, design review, and code review. The two forms of testing have their own advantages and limitations. However, what is clear is that there is a need for both forms of testing to be performed for the overall testing to be effective and inexpensive.
7. *Metrics*: It is well known that numbers give us a better insight into things—processes and products. The concept of measurement, represented

in the form of metrics, allows us to quantify the qualitative attributes of processes and products. One can easily identify a few quality attributes of a test process, such as productivity in test design, rate of test execution, rate of defect reporting, rate of defect fixing, and number of defects yet to be fixed. Test metrics allow us to track the progress of testing and help us get a handle on the quality level of the system under test. Metrics are also useful in evaluating the effect of process changes.

8. *Test Tools*: Tools can be used in a number of test-related activities as follows: test case generation, automatic test execution, and defect tracking. Test tools have the potential of reducing test design time and test execution time. Test tools can help in running more tests, thereby achieving more extensive testing. Tracking defects through a tool helps us in making the activity less error prone. Finally, tools make the testing tasks less monotonous for test engineers.
9. *Test Environment*: Ideally, system-level testing should be carried out in the system's operational environment. However, sometimes it is very expensive, or impractical, to set up an operational environment in a test laboratory. Therefore, test environments as close to reality as possible are realized in a test laboratory. Components of a test environment include hardware, software, communication equipments, and tools and techniques to emulate the interaction of the system under test with the outside world. For example, actual interaction with the outside world can be emulated by using a traffic generator. Different test environments are needed to perform different categories of testing. For example, we need one environment for performance testing and another environment for security testing. A test environment should accurately reflect the objective of testing. Without careful design of test environments, we will not be able to measure the quality attributes of a system under test. Moreover, if test environments significantly differ from operational environments, it is likely that a large number of new failures are observed during operation. Thus, test environments have a great impact on the delivered quality, cost, and timeliness of testing.
10. *Office Environment*: A well-organized office and timely available resources have a positive impact on test engineers. An office organization includes such elementary things as comfortable rooms, desks, chairs, and so on. A friendly work environment coupled with good human resource policies contributes to employee satisfaction. Satisfied test engineers in a good mood are likely to be more productive.
11. *Commitment and Motivation*: In a test project, several kinds of people are involved, namely, managers, test engineers, and technicians. Managers need to understand the importance of the task on hand and motivate others by providing them with training and resources. Test engineers are the people who design and execute test cases. Technicians provide support services. Even if software testing is not their preferred job, they need to be fully committed to their work. If test engineers stay in their position

in the company for a very short time, one can argue that they were not fully committed to testing.

12. *Test Functions and Training*: A test group needs to be composed of people with different kinds of expertise and training. Some people have management expertise, some are experts in the subject area of the system under test, and some are experts in testing. The test engineers need to have different expertise as well. For example, some test engineers are experts in performance test, some are experts in security test, some are experts in load test, and so on. If there is no right mix of people in a test group, some of them may be trained in some areas. Good interpersonal skills are also required for someone to be a successful test engineer.
13. *Scope of Methodology*: The core of a test process is the concept of a methodology comprising activities, procedures, techniques, regulations, and so on. Ideally, an organization uses one methodology but adapts it to different projects depending upon the special needs of the project. For example, one project may contain a large component of real-time processing, whereas another project contains a large component of the graphical user interface. The differences in project characteristics may require an organization to put different emphasis on test generation and execution techniques. But, essentially, the two projects can be adaptations of a generic methodology. The generic methodology should be flexible enough to lend itself to easy adaptation. On the other hand, it is highly undesirable to devise a new methodology for each new test project.
14. *Communication*: A test group communicates within itself and with the outside groups, namely, the developers, the marketing group, and the customers. Good internal communication is essential to working as a cohesive group with opportunity to keep oneself abreast of the happenings in the group. Communication with customers allows the test group to learn about customer expectations. Communication with developers is essential to understanding the design details of the system. The marketing group, by communicating with the test group, keeps track of the progress of system testing and the current quality level of the system under test.
15. *Reporting*: There are different kinds of reporting activities, such as internal reporting and external reporting, involved in a test process. Reporting defects is an internal activity. The quality of the system under test must be reported to the customers. Reporting to customers falls under external reporting. By giving useful reports to customers, it is possible to build a good relationship with them.
16. *Defect Management*: Test engineers report defects to be fixed by developers. However, the state of a defect is more complex than a fixed-or-not-fixed scenario. For example, a reported defect must be analyzed to understand if it is indeed a defect or a misunderstanding of the test engineers. A true defect is to be taken up for fixing by one or more developers. Therefore, a defect may stay in the open state for many weeks. Once

a defect is claimed to be fixed, test engineers must verify that the defect has indeed been fixed. Thus, it is helpful to follow a life-cycle model to manage defects. By managing defects by using a life-cycle model, we can easily identify the state and progress of testing as well as the quality level of the system under test.

17. *Testware Management*: The products developed during software development are used while testing a system. Such products are requirements specification, functional design specification, source code, a test plan, and a test suite. Since revisions are made to those entities on a need basis, it is possible that several versions of a product exist at the time of system testing. Therefore, the right version of each product should be used for efficient and effective testing. This can be achieved through proper use of testware management tools.

18. *Test Process Management*: Test processes are highly visible because everyone, including developers, marketing managers, and customers, is interested in knowing the progress of testing and the present quality level of the system and its trend. System testing occurs so close to the planned delivery date of the product that it attracts additional attention from all. Therefore, to be able to stick to the schedule and remain within budget, proven practices in management should be followed. An example of managing a test process in a defined manner is as follows: (i) Prepare a plan for the process, (ii) verify the plan, (iii) execute the plan, (iv) track the progress, and (v) take corrective and preventive measures.

19. *Evaluation*: Testing is just one aspect of quality assurance. The quality of an entire system can be improved by evaluating the quality of all the products, apart from source code, built during development. For example, it is useful to evaluate requirements and design documents, user manuals, and the different tools used in system development, including testing tools. The concept of test-related evaluation can be applied throughout the development process. By evaluating other products before carrying out execution-based testing, an organization can reduce the cost of software development. This is because evaluation of intermediate products allows for early detection of defects.

20. *Low-Level Testing*: The test processes in an organization cannot focus on system-level testing alone. Before an entire system is subject to the rigors of system-level testing, much testing, in the form of unit level and integration level, needs to be carried out by individual programmers and integration test engineers, respectively. Without a strong belief in unit testing and integration testing, it is merely a consequence that system-level testing will fall apart; that is, no meaningful system testing can be performed if basic things fail to work. The efficacy of low-level testing, namely, unit and integration testing, is derived from the fact that defects are detected and fixed much early in the development phase.

Levels of Maturity of Key Areas A test process is evaluated with respect to the 20 key areas explained above. Specifically, one is interested in knowing to what extent a certain key area has matured. All the key areas may not mature to the same extent because of the fact that an organization might not put an equal amount of emphasis on all. The maturity levels of the key areas are denoted by A, B, C, and D, where A represents the lowest level of maturity and D the highest level. The maturity of a key area increases as we move from A to B, B to C, and so on. Each level is better than its predecessor in one or more ways from the group of three desired parameters: quality, time, and money. Our intention to seek improvements in quality, time, and money (cost) is consistent with the previous discussion of the need to improve a test process in Section 18.3.

For a key area to reach a certain level, the key area must meet the requirements, called *check points*, associated with that level and all the levels below it for that key area. For example, a test process pegged at level C must meet all the requirements for levels A, B, and C. If a test process does not meet the requirements of level A, the process stays at an initial level below A. The requirements for each key area to reach a certain maturity level are given in Table 18.1. By analyzing Table 18.1, we make the following observations:

- All key areas do not have four maturity levels. For example, the key area test specification technique has two levels of maturity, namely, A and B, and B is the highest maturity level for this key area. For the key area test specification techniques, the two distinct requirements of importance are informal techniques and formal techniques, in that order. If test cases are never put in writing, then the maturity of test specification techniques remains at the initial level, which is below A. The number of maturity levels is determined by *all* the requirements a key area is expected to meet and how conveniently we can group those requirements in increasing order of strength so that progress from one group to the next results in a distinct improvement and a better test process. If there are not many requirements for a key area to meet, there is no need to have four levels of maturity.
- The highest maturity level of a key area need not be D. In the key area test specification techniques, the highest maturity level is B.
- There exist dependencies between maturity levels of key areas. In the given context, dependency means that for one key area to reach a certain level of maturity, some other key areas must have reached certain maturity levels. Now let us consider an example as follows. For the key area evaluation to be at level A, it is important that test specification techniques, communication, and defect management are at B. Thus, the maturity levels of all key areas do not improve in lock steps. The idea of maturity dependency calls for improvement in maturity levels of key areas in a *prioritized* manner. For example, we first raise the maturity level of defect management to B before we initiate a process to elevate the maturity level of key area evaluation to level A.

  Based on the ideas of dependencies and prioritization, a test maturity matrix, as shown in Table 18.2, is constructed. The test maturity matrix

TABLE 18.1 Requirements for Different Maturity Levels

Key area	Level A	Level B	Level C	Level D
Test strategy	Strategy for single high-level test	Combined strategy for high-level tests	Combined strategy for high-level tests plus low-level tests or evaluation	Combined strategy for all test and evaluation levels
Life-cycle model	Planning, specification, execution	Planning preparation, specification, execution, and completion		
Moment of involvement	Completion of test basis	Start of test basis	Start of requirements definition	Project initiation
Estimating and planning	Substantiated estimating and planning	Statistically substantiated estimating and planning		
Test specification technique	Informal techniques	Formal techniques		
Static test techniques	Inspection of test basis	Checklists		
Metrics	Project metrics (product)	Project metrics (process)	System metrics	Organization metrics (more than one system)
Test tools	Planning and control tools	Execution and analysis tools	Extensive automation of the test process	
Test environment	Managed and controlled test environment	Testing in most suitable environment	Environment on call	
Office environment	Adequate and timely office environment			
Commitment and motivation	Assignment of budget and time	Testing integrated in project organization	Test engineering	
Test functions and training	Test manager and testers	(Formal) methodical, technical, and functional support, management	Formal internal quality assurance	

TABLE 18.1 (*Continued*)

Key area	Level A	Level B	Level C	Level D
Scope of methodology	Project specific	Organization generic	Organization optimizing, R&D activities	
Communication	Internal communication	Project communication (defects, change control)	Communication within organization about the quality of the test process	
Reporting	Defects	Progress (status of tests and products), activities (costs and time, milestones), defects with priorities	Risks and recommendations, substantiated with metrics	Recommendations have a software process improvement character
Defect management	Internal defect management	Extensive defect management with flexible reporting facilities	Project defect management	
Testware management	Internal testware management	External management of test basis and test object	Reusable testware	Traceability system requirements to test cases
Test process management	Planning and execution	Planning, execution, monitoring, and adjusting	Monitoring and adjusting within organization	
Evaluation	Evaluation techniques	Evaluation strategy		
Low-level testing	Low-level test life cycle (planning, specification, and execution)	White-box techniques	Low-level test strategy	

Source: From ref 4. © 1999 Pearson Education.

TABLE 18.2 Test Maturity Matrix

	Scale													
	Controlled						Efficient					Optimizing		
Key area	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Test strategy		A	—	—	—	—	B	—	—	—	C	—	D	
Life-cycle model		A	—	—	B									
Moment of involvement			A	—	—	—	B	—	—	—	C	—	D	
Estimating and planning				A	—	—	—	—	—	—	B			
Test specification techniques		A	—	B										
Static test techniques					A	—	B							
Metrics						A	—	—	B	—	—	C	—	D
Test tools					A	—	—	B	—	—	C			
Test environment				A	—	—	—	B	—	—	—	—	—	C
Office environment				A										
Commitment and motivation		A	—	—	—	B	—	—	—	—	—	C		
Test functions and training				A	—	—	B	—	—	—	C			
Scope of methodology					A	—	—	—	—	—	B	—	—	C
Communication			A	—	B	—	—	—	—	—	—	C		
Reporting		A	—	—	B	—	C	—	—	—	—	D		
Defect management		A	—	—	—	B	—	C						
Testware management			A	—	—	B	—	—	—	C	—	—	—	D
Test process management		A	—	B	—	—	—	—	—	—	—	C		
Evaluation							A	—	—	B				
Low-level testing					A	—	B	—	C					

Source: From ref. 4. © 1999 Pearson Education.

shows that the overall maturity of a test process can be represented on a scale of 1–13. In the following, we classify the 13 scales of maturity of a test process into three distinct qualitative segments.

Maturity Levels of Test Processes The 13 scales of maturity shown in Table 18.2 are divided into three qualitative groups, namely, *controlled*, *efficient*, and *optimizing*, as discussed below:

- **Controlled:** The group of scales 1–5 is labeled as controlled. A test process executed in a controlled manner means that all component activities are planned, and those are executed in phases according to a planned strategy. In this group some key areas have achieved maturity level B, whereas the rest have achieved maturity level A.
- **Efficient:** After having established mechanisms to execute the component activities of a test process in a controlled manner (scales 1–5), more effort needs to be made to achieve efficiency in testing. The maturity levels of key areas falling on scales 6–10 emphasize efficiency. In this group of

scales, all the key areas, except evaluation, are raised to at least level B with some being at C. For example, the key area moment of involvement is improved to levels B and C in the efficient group. A level C maturity of moment of involvement means involving a test process at the requirements definition phase. Such early thinking about testing of requirements allows the SQA team to design test cases that require less rework cost. By having the key area static testing techniques mature to level B, the cost of testing can be reduced by employing requirements reviews, design reviews, and code reviews.

- **Optimizing:** The three test process levels 11–13 fall in the optimizing group. In this group, all the key areas have reached their respective highest maturity levels. Optimizing a test process means performing testing tasks in the best possible manner from the standpoints of quality, time, and cost. As time passes, new techniques and tools may be available to solve testing problems. Therefore, a test process must evolve continuously to be able to take advantage of new technologies and tools. Organizations may apply the lessons learned from their past projects in order to do things to achieve better results in quality, while spending less money and less time on testing.

Applying the TPI Model In the following, we explain how to apply the TPI model to improve a test process.

- Analyze the current test process, using Table 18.1, in terms of the 20 key areas and give each key area a rating—A, B, C, or D—as applicable, depending on the maturity level of the key areas.
- Evaluate the current scale, between 1 and 13, of the test process by comparing the current status of the test process with the standard test maturity matrix of Table 18.2.
- Identify the goal of the organization in terms of the next scale, between 1 and 13, to be achieved by the organization. It is evident from the test maturity matrix in Table 18.2 that, in order to move from one scale to the next, one does not have to seek improvements in all key areas. For example, if we want to move from scale 5 to 6, we need improvements in key areas test strategy (A to B), moment of involvement (A to B), static test techniques (A to B), test function and training (A to B), reporting (B to C), evaluation ("initial" to A), and low-level testing (A to B). The other key areas remain at their current levels of maturity—these key areas need not improve to higher maturity levels because their impacts may not be useful at the test process scale 6. For example, too much detailed information concerning test metrics need not be collected if the test process is not going to use them, say, for optimization purpose, at scale 6.
- Finally, actions need to be taken to improve the key areas identified in the preceding step. Once again, Table 18.1 is used to identify the action items.

18.4 TESTING MATURITY MODEL

Similar to the concept of evaluating and improving software development processes, there is a need for a framework to assess and improve testing processes. Continuous improvement of testing processes is an ideal goal of organizations, and evaluation plays a key role in process improvement. This is because an organization must know its present level of maturity before it takes actions to move to the next level. Ilene Burnstein and her colleagues at the Illinois Institute of Technology pioneered the concept of the TMM to help organizations evaluate and improve their testing processes [5, 6].

The TMM describes an evolutionary path of test process maturity in five *levels*, or *stages*, as illustrated in Figure 18.3. The TMM gives guidance concerning how to improve a test process. Each stage is characterized by the concepts of

Figure 18.3 Five-level structure of TMM. (From ref. 5. © 2003 Springer.)

maturity goals, *supporting maturity goals*, and *activities, tasks, and responsibilities* (ATRs), as explained in the following:

- **Maturity Goals:** Each maturity level, except level 1, contains certain maturity goals. For an organization to achieve a certain level of maturity, the corresponding maturity goals must be met by the organization. The maturity goals are specified in terms of testing improvement goals.
- **Maturity Subgoals:** Maturity goals are supported by maturity subgoals. To achieve a maturity goal, it may be necessary to meet several, fine-grained subgoals.
- **Activities, Tasks, and Responsibilities:** Maturity subgoals are achieved by means of ATRs that address issues concerning implementation of activities and tasks. ATRs also address how an organization can adapt its practices so that it can move in-line with the TMM model, that is, move from one level to the next. ATRs are further refined into "views," known as critical views, from the perspectives of three different groups of people: managers, developers and test engineers, and customers (users/clients).

The maturity goals associated with the five levels of the TMM model will be explained in the following.

Level 1. Initial No maturity goals are specified at this level. The TMM level 1 is called the *initial* level. For an organization to be at level 1, nothing special needs to be done. Level 1 represents a scenario where testing is not performed in a planned manner. Testing begins after code is written. At this level, an organization often performs testing to demonstrate that the system works. No serious effort is made to track the progress of testing and the quality level of a product. Test cases are designed and executed in an *ad hoc manner*. Testing resources, such as trained testers, testing tools, and test environments, are not available. In summary, testing is not viewed as a critical, distinct phase in software development.

Level 2. Phase Definition At level 2, the maturity goals are as follows:

- Develop testing and debugging goals.
- Initiate a test planning process.
- Institutionalize basic testing techniques and methods.

(i) Develop Testing and Debugging Goals Separation of testing from debugging is an important growth in the maturity of a testing process. Unit testing and debugging may have some common features such as those being performed by individual programmers. However, their separation becomes more evident when we consider higher levels of testing, for example, integration testing, system-level testing, and acceptance testing. As we move away from unit-level testing to system-level testing, for example, test engineers are more interested in examining the higher level features of the system, and they do not focus on code-level details. On the other hand, debugging is primarily a code-level activity

that is initiated after a defect is detected in the software. The very nature of debugging dictates that it cannot be performed in a planned manner, except for getting ready to use tools in debugging.

Testing encompasses activities that can be carefully planned and executed. Responsibilities for testing and debugging must be properly assigned. Assigning responsibility for debugging is easier—the developer who writes the code for a unit or maintains it is in charge, by default, of debugging the unit. Similarly, assignment of responsibilities for unit testing is easy—the programmer of the unit performs unit testing. Assignment of responsibilities for integration testing, system testing, and acceptance testing is a nontrivial task because of the need for test personnel to possess different expertise. For example, some test personnel need to be experts in performance testing.

Some concrete maturity subgoals that can support this goal are as follows:

- Organizations form committees on testing and debugging, and those committees are supported with necessary funding.
- The committees develop and document testing goals. One example of a testing goal is to track a defect from its initial state of discovery to its final state of being fixed and verified.
- The committees develop and document debugging goals.
- The documented testing and debugging goals are widely used in the organization from managers to developers and test engineers. All the people in the organization are aware of the testing and debugging goals and strive to achieve those goals.

(ii) Initiate a Test Planning Process Planning is an indication of a certain level of maturity of any process. The exact extent of maturity depends upon the scope and execution of a plan. Test planning addresses the following:

- **Identify Test Objectives:** Identifying the objective of a test process is central to its success. Without explicit objectives it will be difficult to monitor and guide the progress of testing. Moreover, without explicit objectives it will be difficult to measure the amount of success achieved in testing. Two categories of objectives can be specified: software quality related and process related. One example of the former category of objectives is: Program units must pass all the unit-level tests before initiating integration tests. Another example of the software quality related objectives is: System-level test cases are grouped into functional test, reliability test, load test, security test, and so on.

  An example of a process-related objectives is: While system testing is in progress, the percentage of test cases passing should be monitored on a weekly basis. Another example is: System testing is reduced time by using multiple test beds.
- **Analyze Risks:** Factors that may adversely affect testing must be identified. An example of a risk in testing is that much emphasis has been put

on the performance aspect of the system, but no test engineer is capable of designing and executing performance tests during system testing.

- **Devise Strategies:** Different kinds of strategies are involved in testing. For example, we can choose one among several integration testing strategies, such as top down, bottom up, and incremental, or a strategy that needs to be followed during the execution of system test cases as discussed in Section 12.7. In fact, every major testing activity requires a certain strategy to be followed.
- **Develop Test Specifications:** Individual test cases must be judiciously designed and documented. There must be an objective attached to the design of each test case. Next, a procedure outlining the steps of the test case is designed. Desired inputs and the expected outcomes corresponding to the test case are identified. The initial condition for executing each test case is also identified. Finally, a test case is properly documented for ease of use and accessibility.
- **Allocate Resources:** Resource allocation is an important task in any planned activity. A number of different kinds of resources are needed for software testing: test engineers with the right expertise, resources for setting up test environments, and resources for managing the testing process. In other words, for each test activity, personnel, funds, and time must be allocated.

The following concrete maturity subgoals can support the above goal at level 2:

- The organization assigns the task of test planning to a defined committee. The committee develops a framework for test planning.
- The committee develops a test plan template, and the template is well documented and made widely available.
- Proper tools are used to create and manage test plans.
- Provisions are put in place so that customer needs constitute a part of the test plan. In other words, customers are involved in the testing process.

(iii) Institutionalize Basic Testing and Methods A number of basic testing techniques and methods are widely known in the industry. For example, unit-level testing can be carried out by focusing on the control flow and data flow aspects in a program unit, and a number of code coverage metrics are associated with those testing techniques. Coverage metrics associated with those techniques allow testers to quantify unit-level testing. Moreover, both white-box and black-box testing methods can be applied at the unit level. As testers move from a unit level to higher levels, such as integration and system levels, black-box testing becomes the norm as it is extremely difficult to apply white-box testing techniques to a large subsystem. One needs to apply one or more techniques to perform black-box testing: functional program testing, equivalence class partitioning, boundary value analysis, decision table, and error guessing. A requirement traceability matrix must be defined to maintain an association between requirements and test cases, that is,

what test cases cover each requirement, and what requirement is covered by each test case. By monitoring what test cases have passed, one knows what requirements have been satisfied so far by the system under test. The above maturity goal can be supported by the following subgoals:

- An expert group is formed to study, evaluate, and recommend a set of basic testing techniques and methods. The group also recommends a few tools to support those testing techniques and methods.
- Management must establish policies to ensure that the recommended techniques and methods are practiced and tools are used throughout the organization in a consistent manner.

Level 3. Integration At level 3, as the name suggests, testing is fully integrated with the development process right from the project planning. Testing is not limited to an activity that starts after coding is over. Neither is testing considered to be solely execution based. Rather, different kinds of test activities are performed throughout the life cycle of a system, and some of those test activities are performed without executing code. At level 3, an organization creates a separate test group consisting of test specialists and test managers, their own resources, and a schedule to be able to inject a sense of quality into software products from project conception. For example, at the time of requirements specification, the test group can influence the process of requirements gathering by (i) confirming that all the requirements are testable and (ii) including their own requirements so that customer requirements are testable. The independent test group evaluates the system from the customer's perspective by directly interacting with the customers and incorporating their views into the test design process. By running independently, the test group focuses on quality-related matters, such as institutionalizing testing techniques and methods, evaluating testing tools, evaluating the testability of requirements, hiring and training test engineers by considering project needs, and defining test metrics. In other words, their objective is to perform test activities so that higher quality software is produced at lower cost and in a short time. The maturity goals at level 3 are as follows:

- Establish a software test group.
- Establish a technical training program.
- Integrate testing into the software life cycle.
- Control and monitor the testing process.

(i) Establish Software Test Group Software testing is a highly specialized task at par with capturing requirements and designing a system. It has been widely recognized that testing activities must be initiated as soon as the planning for a project begins. Therefore, an independent test group must exist to carry out testing without any influence from the developers. The advantages of having an independent test group are as follows:

- The independent test group can develop the desired expertise in testing by hiring the right people and providing training to its members.

- The test group can have a positive impact on software quality by having its members participate in the meetings for requirements review in the requirements gathering phase of the software life cycle. The members from the test group ensure that the requirements are testable. Ensuring the testability of a requirement is important because without a concrete procedure to verify that a software system possesses a certain requirement, the organization cannot convince the customers that their needs are fulfilled.
- Testing involves a variety of specialized tasks, such as test planning, test design, test execution, test scheduling, following test-related standards, gathering and monitoring test metrics, maintaining test cases, and tracking defects. Therefore, scheduling those tasks and procuring the required resources are vital to the completion of system testing on time.
- The quality of a software product is independently evaluated without any interference from the development group. This independence is central to avoiding any intentional or unintentional bias that the developers may have toward their product.
- Customers feel much more confident about a product if it has been tested and evaluated by a test group which is independent of the development group.

Therefore, testing must be performed by an independent test group. Maturity subgoals which support the above goals are as follows:

- An organizationwide test group is formed with leadership, strong support, and funding from the management. The test group must be vested with authority to influence the quality of software products so that formation of the test group is not rendered as a mere exercise.
- The test group must be involved in all stages of the software development, and roles and responsibilities must be assigned to individual group members in appropriate phases of software development.
- Trained and motivated test engineers are assigned to the test group.
- The test group in an organization must communicate with the customers to get a feel for their needs so that testing can be carried out to meet the expectations of the customers.

(ii) Establish Technical Training Program A technical training program is essential to maintaining a group ofqualified and skilled staff for testing. Organizations tend to hire not so skilled staff for testing jobs because of general unavailability of sufficiently qualified test personnel in the industry. Therefore, organizations must make a sustained effort to train test personnel in the concept of quality, test planning, testing methods and techniques, standards, and testing tools. Test engineers learn the concepts and processes of various kinds of reviews, such as design review and code review, which are discussed in Chapter 3.

(iii) Integrate Testing into Software Life Cycle In contrast to level 2, where testing is initiated as a distinct activity after coding is over, at level 3 testing is fully

integrated with a software development life cycle at the start of project planning. Therefore, test planning is initiated very early in the lifetime of a product. Early focus on testing allows the test group to participate in requirements gathering, design, and code reviews. Maturity subgoals to support the above goal are as follows:

- The test phase is partitioned into several activities, such as test planning, unit testing, integration testing, system testing, and acceptance testing, so that the distinct activities are easily performed at the appropriate times in the life cycle of a software product. Different kinds of reviews, such as requirements review, design review, and code review, are identified. All those testing and review activities are integrated into the V-model of testing.
- All the testing and review activities are integrated into an institutionalized V-model of testing, and the organization must ensure that the model is followed.
- The organization supports an effective mechanism for the test group to communicate with developers, customers, and the marketing group. As a result, test engineers learn about the requirements, design details, customer expectations, and marketing philosophy of the organization. The more the test group is knowledgeable about the product, the more it can influence product quality. The upper management of an organization must facilitate such communication.

(iv) Monitor and Control Testing Process Monitoring and control are important aspects of planning. Without monitoring it will not be known if a project is on course, and, consequently, no corrective measures can be taken to control the project. Level 3 concerns several test-related monitoring and control activities to provide visibility to the progress of a testing project. Management can take effective action for a test project to stay as close to the plan as possible when the progress of the project significantly differs from the plan. Progress of a test project can be measured as follows:

- **Amount of Testing effort:** This includes the number of test cases to be designed and the number of test cases to be executed.
- **Cost of Testing:** This represents all the cost in carrying out test-related activities.
- **Schedule:** This concerns the start and finish times of the test activities.

The following maturity subgoals are required to support the above goal:

- The organization develops policies and mechanisms to monitor and control test projects.
- A set of metrics related to the employed test process must be defined, recorded, and made available to all concerned test engineers.

- A set of potential corrective actions and contingency plans must be defined and documented. It may be used when the actual progress of a testing project significantly differs from the plan.

Level 4. Management and Measurement At level 4 of the TMM, testing acquires a much larger scope and is not just another phase in a software development life cycle. The following are the maturity goals at level 4:

- Establish an organizationwide review program.
- Establish a test measurement program.
- Evaluate software quality.

(i) Establish Organizationwide Review Program To augment the execution-based testing, a review program is established to detect defects early in the life cycle of product development and at a lower cost. All software products, such as requirements documents and design documents, are reviewed. Moreover, test plans, test cases, and test procedures are thoroughly reviewed. Thus, reviews are performed throughout the life cycle of a product. Maturity subgoals that support this goal are as follows:

- Management should develop review policies and ensure that those policies are rigorously followed.
- The test group should develop goals, plans, procedures, and recording mechanisms for carrying out reviews.
- Objectives must be clearly defined.
- Members of the test group must be trained so that their effective participation in review processes is ensured.

(ii) Establish Test Measurement Program A test measurement program is established to measure productivity, progress, and quality of work associated with testing. For example, productivity in testing refers to the rate of test design in terms of number of test cases per person-day and the number of test cases executed by one person per day. The number of test cases reviewed by one person per day is another example of productivity in testing. Progress of testing refers to how much work has been completed with reference to how much is needed to be done. Therefore, one must measure at regular intervals all the quantities that represent progress. Some examples of measurable quantities are the number of test cases that have been designed and the number of test cases that have been executed. Quality of a software product is measured in terms of failure intensity, that is, the number of failures observed per unit time. Maturity subgoals that support this goal are as follows:

- Test metrics should be identified along with their goals.
- A test measurement plan should be developed with data collection and analysis.
- An action plan should be developed to achieve process improvement by considering the measured data.

(iii) Evaluate Software Quality An important goal of a software development process is to produce software products of the highest possible quality. A test group must measure the quality of a product to learn what quality a certain process can result in and take measures to improve the process with the objective of improving product quality. Therefore, the test group must identify the quality metrics, such as correctness, reliability, maintainability, and usability, for further improvement. The test group must evaluate the adequacy of a testing process for reliable assessment of the resulting qualities in a product. The maturity subgoals that support this goal are as follows:

- The organization should define quality attributes and quality goals for software products.
- Management should develop policies and mechanisms to collect test metrics to support the quality goals.

Level 5. Optimization, Defect Prevention, and Quality Control Optimization and defect prevention are the concepts at the highest level, namely level 5. Intuitively, optimization means spending less resources to achieve higher quality products, and defect prevention means taking measures throughout the development process so that products are largely defect free. At level 5, the maturity goals are as follows:

- Application of process data for defect prevention
- Statistical quality control
- Test process optimization

(i) Application of Process Data for Defect Prevention At level 5, organizations make a sustained effort to learn from experience to reduce the number of defects in a system. Defect prevention is an important characteristic of mature organizations. Test groups in mature organizations analyze defects and their patterns and perform root cause analysis to better understand the pattern of defects. To prevent the recurrence of common defects in future projects, action plans are developed and processes are improved. Mechanisms must be in place to ensure that the action plans are followed. Such mechanisms include formation of a defect prevention team that interacts with developers to apply defect prevention activities in the development process. Maturity subgoals that support this goal are as follows:

- Management should establish a defect prevention team.
- Defects identified and removed are documented in each phase of the development.
- Each defect is analyzed to get to its root cause.
- Managers, developers, and the test group should interact to develop an action plan to eliminate recurrence of commonly occurring defects.
- Management should put in place a mechanism to implement and track the action plan.

(ii) Statistical Quality Control At level 5, organizations further improve the quality of software products by driving the testing process with statistical sampling, measurement of confidence levels, trustworthiness, and software reliability goals. This goal is stronger than the software quality evaluation goal at level 3. It may be recalled that the quality evaluation goal focuses on different kinds of software qualities, such as functionality, reliability, usability, and robustness. Automated tools are used for defect collection and analysis. Usage modeling is used to perform statistical testing, where a usage model is selected from a subset of all possible usages of the software. From statistical testing one can conclude the general operational performance of the software product. Subgoals that support statistical quality control are as follows:

- The test group establishes high-level measurable quality goals, such as test case execution rate, defects arrival rate, and total number of defects that can be found during testing.
- Managers ensure that the new quality goals form a part of the test plan.
- The test group is trained in statistical testing and analysis methods: Pareto analysis, cause-and-effect diagram, flow chart, trend chart, histogram, scatter diagram, and control chart.
- User inputs are gathered for usage modeling.

(iii) Test Process Optimization Test optimization refers to all those activities which result in continuous improvement of the testing process. An essential element of test optimization is quantification of the testing process so that accurate measurements can be performed to find where further improvements are possible. Test optimization involves the following activities:

- Identify the testing practices that can be improved.
- Define a mechanism to improve an identified practice.
- Put a mechanism in place to track the practice improvement mechanism.
- Continuously evaluate new test-related technologies and tools.
- Develop management support for technology transfer.

Maturity subgoals that support test process optimization are as follows:

- Establish a test process improvement group to monitor the testing process and identify areas for improvement.
- Evaluate new technologies and tools to improve the capability of the testing process. There must be management support to establish policies and mechanisms for this purpose.
- A mechanism is put in place for continual evaluation of the effectiveness of the testing process.
- Test stopping criteria are based on quality goals, which are discussed in Chapter 12.

18.5 SUMMARY

The CMM is a framework for evaluating and improving the present capability of a software development company. As the name suggests, the CMM can be used to assess to what extent the capability of an organization to develop software systems has matured. The model allows one to give a rating to an organization on a scale of 1–5, where 5 represents the highest maturity level. The model allows one to evaluate an organization by examining its current practices. Basically, the CMM is a process improvement approach. In order to move to the next level of maturity, organizations need to adopt new practices relevant to the next maturity level. Level 1 is the *initial* level, level 2 is known as *repeatable*, level 3 is the *defined* level, level 4 is *managed*, and level 5 is the *optimizing* level. Each level is characterized by a set of *key process areas*.

It is useful to recall that the CMM focuses on the entire development process, whereas the TPI model allows us to focus on a test process alone. A test process is a certain manner of performing activities related to defect prevention and detection. Some examples of testing activities are identifying testing goals, preparing a test plan, and designing test cases. A test process needs to be improved for three reasons: (i) A better test process should give more insight into the quality characteristics of a system, (ii) a better test process should reduce testing time, and (iii) a better test process should cost less. Tim Koomen and Martin Pol [4] introduced the idea of improving a test process by proposing the TPI model. Intuitively, any approach to improving a test process consists of the following four steps: (i) Determine an area for improvement, (ii) evaluate the current state of the test process, (iii) identify the next desired state and the means to achieve it, and (iv) execute the necessary changes to the process. A test process is evaluated with respect to 20 key areas, such as test strategy, moment of involvement, planning, and metrics. The maturity level of a test process is assessed on a scale of 1–13. The 13 levels of test process maturity are partitioned into three broad qualitative groups, namely, *controlled*, *efficient*, and *optimizing*.

Finally, the TMM gives guidelines concerning how to improve a test process. In this model, the maturity of a testing process is evaluated on a scale of 1–5. Each of the five maturity levels is characterized by the concepts of maturity goals; maturity goals are supported by means of subgoals and activities, tasks, and responsibilities. The five levels of maturity are known as initial (level 1), phase definition (level 2), integration (level 3), management and measurement (level 4), and optimization, defect prevention, and quality control (level 5).

LITERATURE REVIEW

Numerous books and articles have been written about CMM and CMMI. A comprehensive article about CMM is by Paulk, Curtis, Chrissis, and Weber [2]—the first three authors are from the Software Engineering Institute. Two good books on CMM and CMMI are by Land [3] and Mutafelija and Stromberg [7], respectively. However, revisions are made to the CMM and CMMI documents from time to

time. Thus, it is useful to refer to the main web page of SEI for timely information about CMM and CMMI: http://www.sei.cmu.edu/.

For additional information about the TPI model, the reader is referred to the book entitled *Test Process Improvement* by Koomen and Pol [4]. For additional information about the TMM, the reader is referred to the book entitled *Practical Software Testing* by Burnstein [5].

Another maturity model that is frequently adopted by organizations is known as the Six Sigma maturity model to address quality and customer satisfaction issues. Six Sigma was created by some of America's most gifted CEOs, people like Motorola's Bob Galvin, AlliedSignal's Larry Bossidy, and GE's Jack Welch. Six Sigma is a business-driven, multifaceted approach to process improvement, reduced costs, and increased profits. To achieve Six Sigma, a process must not produce more than 3.4 defects per million opportunities. The Six Sigma DMAIC methodology, consisting of the five steps define, measure, analyze, improve, and control, is an improvement system for existing processes falling below specification and looking for incremental improvement. The Six Sigma DMADV methodology, consisting of the five steps define, measure, analyze, design, and verify, is an improvement system used to develop new processes or products at Six Sigma quality levels. It can also be employed if a current process requires more than just incremental improvement. Interested readers are encouraged to read the following books for detailed discussions of this subject:

M. Harry, *Six Sigma: The Breakthrough Management Strategy Revolutionizing the World's Top Corporations*, Random House, New York, 2000.

T. Pyzdek, *The Six Sigma Handbook*, McGraw-Hill Professional, New York, 2001.

REFERENCES

1. R. R. Macala, Jr., L. D. Stuckey, and D. C. Gross. Managing Domain-Specific, Product-Line Development. *Software Practice and Experience*, Vol. 13, No. 3, 1996, pp. 57–68.
2. M. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber. Capability Maturity Model, Version 1.1. *IEEE Software*, July 1993, pp. 18–27.
3. S. Land. *Jumpstart CMM/CMMI Software Process Improvement*. Wiley, Hoboken, NJ, 2005.
4. T. Koomen and M. Pol. *Test Process Improvement*. Addison-Wesley, Reading, MA, 1999.
5. I. Burnstein. *Practical Software Testing*. Springer, New York, 2003.
6. I. Burnstein, A. Homyen, T. Suwanassart, G. Saxena, and R. Grom. A Testing Maturity Model for Software Test Process Assessment and Improvement. *Software Quality Professional*, September 1999, pp. 1–8.
7. B. Mutafelija and H. Stromberg. *Systematic Process Improvement Using ISO 9001:2000 and CMMI*. Artech House, Boston, MA 2003.

Exercises

1. Briefly explain the CMM architecture.
2. Briefly explain the five maturity levels in the CMM model.
3. Briefly explain the common features of key practices in the CMM model.

4. Briefly explain the idea of a test process.
5. Why is it important to improve a test process?
6. Briefly explain an intuitive approach to improving a test process.
7. Briefly explain how the current status of a test process can be evaluated.
8. Briefly explain the levels of maturity of key areas in the TPI model.
9. Briefly explain the main idea in the TMM.
10. Briefly explain the different levels in the TMM in terms of their maturity goals.

GLOSSARY

A man is born alone and dies alone; and he experiences the good and bad consequences of his karma alone; and he goes alone to hell or the supreme abode.
— *Chanakya*

1xEvolution-data optimized (1xEV-DO) Communication standard for transmitting and receiving data frames over a wireless radio channel using CDMA technology.

Abstract Syntax Notation One (ASN.1) Notation to formally define the syntax of messages to be exchanged among an extensive range of applications involving the Internet.

Acceptance criteria Criteria a system must satisfy to be accepted by a customer and to enable the customer to determine whether to accept the system.

Acceptance test Formal testing conducted to determine whether a system satisfies its acceptance criteria.

Access terminal Can be a mobile phone, laptop, or personal digital assistant (PDA) with a wireless modem.

Accuracy Degree of conformity of a measured or calculated quantity to its actual (true) value.

Adaptive random testing In adaptive random testing, test inputs are selected from a randomly generated set in such a way that these test inputs are evenly spread over the entire input domain.

Adjacent domain Two domains are adjacent if they have a boundary inequality in common.

Asynchronous transfer mode (ATM) Cell relay network protocol which encodes data traffic into small, fixed-sized (53 bytes = 48 bytes of data and 5 bytes of header information) cells. A connection-oriented technology in which a connection is established between the two endpoints before an actual data exchange begins.

Attributes Properties of the service delivered by the system to users.

Authentication Process of verifying the claimed identity of someone or something.

Authentication, authorization, and accounting (AAA) Network server used for controlling access to a network. An authentication process identifies the user. An authorization process implements policies that determine which resources and services a valid user may access. An accounting process keeps track of time and data resources used for billing and usage analysis.

Authorization Process of verifying whether an individual has permission to access a specific resource.

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

Automatable Test case that is a good candidate for automation.

Availability Measure of the readiness of a system. Simply put, availability is the proportion of time a system is in a functioning condition.

Backdoors Mechanism created by a computer program that allows anyone with knowledge of its existence to gain access to the system.

Backup/recoverability test Verifies that a system can be recouped after a failure. It is done by backing up to a point in the processing cycle before any error occurred and reprocessing all transactions that occurred after that point.

Bad fix Fix causing collateral damage.

Basic interconnection test Verifies whether the implementation can establish a basic interconnection before thorough tests are performed.

Basic test Provides a prima facie indication that the system is ready for more rigorous tests.

Behavior test Verifies the dynamic communication systems requirements of an implementation. These are the requirements and options that define the observable behavior of a protocol. A large part of behavior tests, which constitutes the major portion of communication system tests, can be generated from the protocol standards.

Beta testing Testing conducted by potential buyers prior to the release of the product. The purpose of beta testing is intended not to find defects but to obtain feedback from the field to the developers about the usability of the product.

Big-bang integration Integration testing technique in which all the software modules are put together to construct the complete system so that the system as a whole can be tested.

Bit error test (BERT) Involves transmitting a known bit pattern over a channel and then verifying the received pattern for errors.

Black-box testing Also called functional testing, a testing technique that ignores the internal details of a system and focuses solely on the inputs accepted, outputs generated, and execution conditions.

Boot test Verifies that the system can boot up its software image from the supported boot options.

Bot Software agent in Internet parlance. A bot interacts with network services intended for people as if it were a person. One typical use of bots is to gather information. Another, more malicious use of bots is the coordination and operation of an automated attack on networked computers, such as a distributed denial-of-service attack.

Bottom-up integration Integration testing technique in which testing starts from the modules at the outermost branches of a module visibility tree and moves toward the modules making up the "main program."

Boundary inequality From a geometrical viewpoint, a domain is defined by a set of boundary inequalities, where each inequality defines a boundary of the domain.

Boundary value analysis (BVA) The aim of BVA is to select elements that are close to the boundaries of an input domain so that both the upper and lower edges of an equivalence class are covered by test cases.

Branch coverage Selecting program paths in such a manner that certain branches (i.e., outgoing edges of nodes) of a control flow graph are covered by the execution of those paths. Complete branch coverage means selecting some paths such that their execution causes all the branches to be covered.

Build Interim software image for internal testing within the organization. Eventually, the final build will be a candidate for system testing, and such a system may be released to customers.

Business acceptance testing (BAT) Undertaken within the supplier's development organization to ensure that the system will eventually pass the user acceptance testing.

Capability maturity model (CMM) Gives guidelines for improving a software development process. The model facilitates the evaluation of the maturity levels of processes on a scale of 1–5. Level 5 is the highest level of process maturity.

Capability test Checks that the implementation provides the observable capabilities based on the static communication system requirements. The static requirements describe the options, ranges of values for parameters, and timers.

Category partition method (CPM) Systematic, specification-based methodology that uses an informal functional specification to produce formal test specification.

Causal analysis A kind of analysis conducted to identify the root cause of a defect and initiate actions so that the source of the defect is eliminated.

Change request (CR) Formal request by a code reviewer to make a change to the code.

Characterizing sequence Sequences of the *W*-set of an FSM are called the characterizing sequences of the FSM.

Check-in request form For each fix that is checked into a build, a check-in request form is filled out by software developers and reviewed by the build engineering group.

Clean-room process Model introduced by IBM in the late 1980s. The process involves two cooperating teams—development and quality assurance teams—and five major activities: specification, planning, design and verification, quality certification, and feedback. The following ideas form the foundation of the clean-room process: (i) incremental development under statistical quality control (SQC), (ii) software development based on mathematical principles, and (iii) software testing based on statistical principles.

Closed boundary A boundary is closed if the data points on the boundary are a part of the domain of interest.

Closed domain A domain with all its boundaries closed.

Closure error Occurs if a boundary is closed when the intention is to have an open boundary or vice versa.

Collateral damage What occurs when a new feature or a defect fix in one part of the system causes a defect (damage) to another, possibly unrelated part of the system.

Combinatorial testing Test case selection method in which test cases are identified by combining values of several test input parameters based on some combinatorial strategy.

Command line interface test Verifies that the system can be configured in a specific way by using the command line interface.

Commercial off-the-shelf (COTS) components Software components produced by third-party vendor organizations that can be reused in a system. Often, these types of components are delivered without their source code.

Compatibility test Verifies that the system can work in the same manner across all platforms, operating systems, database management systems, and network operating systems.

Competent programmer hypothesis Assumption for mutation analysis, which states that programmers are generally competent, and they do not create "random" programs.

Compliance testing Also called conformance testing, the process of verifying whether a product meets the standard product specifications it was designed to meet.

Computation error Occurs when specific input data cause the program to execute the correct path but the output value is wrong.

Confidentiality Encrypting data by a sender such that only the intended receiver can decrypt it.

Configuration testing Reconfiguration activities during interoperability tests.

Conformance testing Process that verifies whether an implementation conforms to its specification.

Control flow graph (CFG) Graphical representation of the flow of control in a program unit.

Coordinated architecture Enhanced version of the distributed architecture, where the upper and lower testers are coordinated by a test management protocol.

Coupling effect Assumption for mutation analysis which states that if a test suite can reveal simple defects in a program, then it can also reveal more complicated combinations of simple defects.

Cross-functionality group In an organization the set of those groups that have different stakes in a product. For example, a marketing group, a customer support group, a development group, a system test group, a development group, and a product sustaining group are collectively referred to as a cross-functionality group in an organization.

Cyclomatic complexity (McCabe's complexity) Based on the graph theory concept and known as cyclomatic number, represents the complexity of a software module.

Data conversion acceptance criteria Used to measure and report the capability of the software to convert existing application data to new formats.

Data flow anomaly Sequence of "unusual" actions on a data variable, for example, two successive assignments of values to a data variable or referencing an undefined variable.

Data flow graph (DFG) Graphical representation of a program, where nodes represent computations and branches represent predicates, that is, conditions.

Debugging Process of determining the cause of a defect and correcting it; occurs as a consequence of a test revealing a defect.

Decision table Comprises a set of conditions and a set of effects. For each combinations of conditions, a rule exists. Each rule comprises a Y (yes), N (no), or—(don't care) response and contains an associated list of effects or expected results.

Defect Flaw in a software with a potential to cause a failure.

Defect age Period of time from the introduction of a defect to its discovery.

Defect density Number of defects per thousand lines of code.

Defect prevention Preventive measures that can be taken during the development of code to reduce the errors in the program.

Defect priority Measure of how soon the defect needs to be fixed.

Defect removal efficiency (DRE) Ratio of the number of defects discovered in an activity to the number of defects that should have been found.

Defect severity Measure of the extent of the detrimental effect a defect can have on the operation of the product.

Definition of a variable A variable is said to be defined if the variable's memory location explicitly gets a value.

Degraded node test Verifies the operation of a system after a portion of the system becomes nonoperational.

Denial-of-service (DoS) attack Flooding an information system, such as a server, with a large number of requests for service to the point where the information system cannot respond.

Design verification test (DVT) Written and executed by the hardware group before integrating the hardware with the software system. Types of DVTs are diagnostic, electrostatic discharge, electromagnetic emission, electrical, thermal, environmental, acoustics, equipment packaging, safety, and reliability.

Deterministic finite-state machine FSM such that its output and the next state are a function of its current state and the input that is applied.

Device under test (DUT) Manufactured product undergoing testing.

Diagnostic tests Verify that the hardware components of the system are functioning as desired. Examples are power-on self test, Ethernet loop-back test, and bit error test.

Digital signature Encrypted message digest that is appended to the message. Producing a digital signature involves public key encryption and a hash function algorithm.

Distinguishing sequence Input sequence which generates a unique output sequence for a state when the input sequence is applied to an FSM starting at the given state.

Distributed architecture Test architecture where there is a PCO at the upper service boundary and another at the lower service boundary. The PCO at the lower service boundary is at the remote end of the $N-1$ service provider to indirectly control and observe N ASPs and N PDUs. This allows the upper and lower testers to reside in physically separate locations.

Domain error Occurs when specific input data cause the program to execute the *wrong* path in the program.

Dynamic unit testing Execution-based testing methodology in which a program unit is actually executed and its outcomes are observed.

Element management system (EMS) test Verifies EMS functionality, such as monitoring and managing the network elements.

Emulator A software emulator allows computer programs to run on a platform (computer architecture and/or operating system) other than the one for which the programs were originally written. Unlike simulation, which only attempts to reproduce a program's behavior, an emulator attempts to model, to various degrees, the states of the device being emulated.

Encryption Cryptographic technique used to provide confidentiality.

Engineering change (EC) document Provides a brief description of the issues and describes what changes are needed to be done to the original requirement.

Engineering change order (ECO) Formal document that describes a change to the hardware or software that is to be delivered to the customers. This document includes the hardware/software compatibility matrix and is distributed to operation, customer support, and the sales teams of the organization.

Entry criteria Criteria to be met before the start of a testing phase.

Error When an event activates a fault in a program, it first brings the program into an intermediate unstable state, called error, which, if and when it propagates to the output, eventually causes a system failure.

Error guessing Test design technique in which the experience of the testers is used to (i) guess the probable kinds and locations of faults in a system and (ii) design tests specifically to expose them. Designing test cases using the error guessing technique is primarily based on a tester's experience with code similar to the implementation under test.

Error seeding Process of intentionally adding known defects in a computer program for the purpose of estimating the number of defects remaining in the program during the process of testing and fault removal.

Equivalence class partitioning Divides the input domain of the system under test into classes (or groups) of test cases that have a similar effect on the system.

Equivalent mutant Mutant that is not distinguishable from the program under test. Determining whether or not a mutant is equivalent to a program is in general undecidable.

Exit criteria Criteria specifying the conditions that must be met before the completion of a testing phase.

Extended finite-state machine Extension of a finite-state machine (FSM). An EFSM has the capability to perform additional computations such as updating values of variables, manipulating timers, and making decisions. The Specification and Description Language (SDL) provides a framework for specifying a system as one or more EFSMs.

Extensible authentication protocol (EAP) Authentication protocol described in Request for Comments (RFC) 2284. For wireless LANs, the EAP is known as EAP over LAN (EAPOL).

Extreme point Point is a point where two or more boundaries cross.

Extreme programming (XP) Software development methodology which is self-adaptive and people-oriented. XP begins with five values: communication, feedback, simplicity, courage, and respect. It then builds up 12 rules/recommendations which XP projects should follow.

Failure Manifested inability of a program to perform its required function. In other words, it is a system malfunction evidenced by incorrect output, abnormal termination, or unmet time and space constraints.

Failure intensity Expressed as the number of failures observed per unit time.

False negative Occurs when a potential or real attack is missed by an intrusion detection system. The more the occurrences of this scenario, the more doubtful the accountability of the intrusion detection system and its technology.

False positive Commonly known as false alarm, occurs when intrusion detection system reads legitimate activity as being an attack.

Fault Cause of a failure. For example, a missing or incorrect piece of code is a fault. A fault may remain undetected for a long time until some event activates it.

Fault-based testing Testing technique used to show that a particular class of faults is not resident in a program. The test cases are aimed at revealing specific kinds of predefined faults, for example, error guessing, fault seeding, or mutation testing.

Fault (error) seeding Process of intentionally adding known faults in a computer program for the purpose of monitoring the rate of detection and removal of faults and estimating the number of faults remaining in the program. Also used in evaluating the adequacy of tests.

Fault injection Method by which faults are introduced into a program. An oracle or a specification is available to assert that what was inserted made the program incorrect.

Fault simulation Process of inserting faults in a program. The inserted faults are not guaranteed to make the program incorrect. In fault simulation, one may modify an incorrect statement of a program and turn it into a correct program.

Feasible path Path in which there exists an input to cause the path to execute.

Feature Set of related requirements.

First customer shipment (FCS) New software build that is released to the first paying customer.

Finite-state machine (FSM) Automata with a finite number of states. The automata changes its state when an external stimulus is applied. The *state* of an FSM is defined as a stable condition in which the FSM rests until an external stimulus, called an *input*, is applied. An input causes an FSM to generate an observable *output* and to undergo a state transition from the current state to a new state where it stays until the next input occurs.

Frame relay (FR) Physical layer data transmission technique for moving data frames from one computer/router to another computer/router.

Full polling Used to check the status and any configuration changes of the nodes that are managed by an EMS server.

Functional specification document Requirements document produced by software developers to represent customer needs.

Functional testing Testing in which a program P is viewed as a *function* that transforms the input vector X_i into an output vector Y_i such that $Y_i = P(X_i)$. The two key concepts in functional testing are as follows: (i) precisely identify the domain of each input and output variable and (ii) select values from a data domain having *important* properties.

Function point (FP) Unit of measurement to express the amount of business functionality an information system provides to a user. Function points were defined in 1977 by Alan Albrecht at IBM.

Gantt chart Popular bar chart to represent a project schedule.

Gold standard oracle Scheme in which previous version of an existing application system is used to generate expected results.

Graphical user interface test Verifies the look-and-feel interface of an application system.

Handoff Procedure for transferring the handling of a call from one base station to another base station.

Hash function Algorithm that takes an input message of arbitrary length and produces a fixed-length code. The fixed-length output is called a hash, or a message digest, of the original input message.

Hazard State of a system or a physical situation which, when combined with certain environmental conditions, could lead to an accident or mishap. A hazard is a prerequisite for an accident.

High-availability tests Verify the redundancy of individual hardware and software modules. The goal here is to verify that the system recovers gracefully and quickly from hardware and software failure without impacting the operation of the system. It is also known as fault tolerance.

High-level design document Describes the overall system architecture.

Ideal test If we can conclude, from the successful execution of a sample of the input domain, that there are no faults in the program, then the input sample constitutes an ideal test.

Implementation under test (IUT) Implementation subject to tests. An IUT can be a complete system or a component thereof.

Inappropriate action Calculating a value in a wrong way, failing to assign a value to a variable, or calling a procedure with a wrong argument list.

Inappropriate path selection If there is a faulty association between a program condition and a path, then a wrong path is selected, and this is called inappropriate path selection.

Infeasible path Program path that can *never* be executed.

Inferred requirement Anything that a system is expected to do but is not explicitly stated in the specification.

In-parameter-order (IPO) testing Combinatorial testing technique for the generation of test cases that satisfy pairwise coverage.

In-process metrics Monitor the progress of the project and use these metrics to steer the course of the project.

Input vector Collection of all data entities read by a program whose values must be fixed prior to entering the unit.

Inspection Step-by-step peer group review of a work product, with each step checked against predetermined criteria.

Installability test Ensures that the system can be correctly installed in the customer environment.

Integrity checking Verifying whether or not data have been modified in transit.

Internet Protocol (IP) Routing protocol used for moving data across a packet-switched internetwork. The IP is a network layer protocol in the Internet protocol suite.

Internet Protocol Security (IPSec) Network layer security protocol which provides security features, including confidentiality, authentication, data integrity, and protection against data replay attacks.

Interoperability test Verifies that the system can interoperate with third-party products.

Intersystem testing Integration testing in which all the systems are connected together and tests are conducted from end to end.

Intrasystem testing Low-level integration testing with the objective of putting the modules together to build a cohesive system. Intrasystem testing requires combining modules together within a system.

Ishikawa diagram Also known as a *fishbone* diagram or cause-and-effect diagram, shows the causes of a certain event. It was first used by Kaoru Ishikawa in the 1960s and is considered one of the seven basic tools of quality management: histogram, Pareto chart, check sheet, control chart, cause-and-effect diagram, flowchart, and scatter diagram.

JUnit Automated testing framework used by developers who implement program units and unit tests in the Java programming language.

Key process area (KPA) A CMM maturity level contains key process areas. KPAs are expected to achieve goals and are organized by common features.

Lean Methodology that is used to speed up and reduce the cost of a manufacturing process by removing waste. The principle to eliminate waste has been borrowed from the ideas of Taiichi Ohno—the father of the Toyota Production System. The lean development

methodology is summarized by the following seven principles: eliminate waste, amplify learning, decide as late as possible, deliver as fast as possible, empower the team, build integrity, and see the whole. The lean process is a translation of the lean manufacturing principles and practices to the software development domain.

Light emitting diode (LED) test Verifies the functioning of the LED indicator status. The LED tests are designed to ensure that the visual operational status of the system and the submodules is correct.

Lightweight Directory Access Protocol (LDAP) Protocol derived from the X.500 standard and defined in Request for Comments (RFC) 2251. LDAP is similar to a database but can contain more descriptive information. LDAP is designed to provide fast response to high-volume lookups.

Lightweight Extensible Authentication Protocol (LEAP) Cisco-wireless EAP which provides username/password-based authentication between a wireless client and an access control server.

Load and scalability test Exercises the system with multiple actual or virtual users and verifies whether it functions correctly under tested traffic levels, patterns, and combinations.

Local architecture Test architecture where the PCOs are defined at the upper and lower service boundaries of the IUT.

Logging and tracing test Verifies the configuration and operation of logging and tracing functionalities.

Logic fault When a program produces incorrect results independent of resource required, the fault is caused due to inherent deficiencies in the program and not due to lack of resource. The deficiencies are in the form of requirement faults, design faults, and construction faults.

Lower tester Tester entity responsible for the control and observation at the appropriate PCO either below the IUT or at a remote site.

Low-level design document Detailed specification of the software modules within the architecture.

Maintainability Aptitude of a system to undergo repair and evolution.

Management information base (MIB) Database used to manage the devices in a communication network.

Manufacturing view of quality Quality is seen as conforming to requirements. The concept of process plays a key role in the manufacturing view.

Marketing beta Beta testing that builds early awareness and interest in the product among potential buyers.

Mean time between failure (MTBF) Expected time between two successive failures of a system. Technically, MTBF should be used only in reference to repairable items, while MTTF should be used for nonrepairable items. However, MTBF is commonly used for both repairable and nonrepairable items.

Mean time to failure (MTTF) Mean time expected until the first failure of a piece of equipment. MTTF is a statistical value and is meant to be the mean over a long period of time and a large number of units. MTTF is a basic measure of reliability for nonrepairable systems.

Mean time to repair (MTTR) Amount of time between when something breaks and when it has been repaired and is fully functional again. MTTR represents the amount of time that the device was unable to provide service.

Milestone Major checkpoint, or a subgoal, identified on the project or testing schedule.

Mishap Also called an accident, an unintended event that results in death, injury, illness, damage or loss of property, or harm to the environment.

Missing control flow paths There is no code to handle a certain condition. This occurs when we fail to identify the condition and, thereby, fail to specify a computation in the form of a path.

Module test Verifies that all the modules function individually as desired within the system. The intent here is to verify that the system along with the software that controls these modules operates as specified in the requirement specification.

Mutation analysis Involves the mutation of source code by introducing statements or modifying existing statements in small ways. The idea is to help the tester develop effective tests or locate weaknesses in the test data or in the code that are seldom or never accessed during execution.

Network element Network node residing on a managed network and running an SNMP agent.

Network management station Executes management applications that monitor and control network elements.

New technology LAN manager (NTLM) Authentication protocol used in various Microsoft network protocol implementations. NTLM employs a challenge–response mechanism for authentication in which clients are able to prove their identities without sending a password to the server.

Nondeterministic finite-state machine FSM in which the next-state function is not solely determined by its present state and an input. An internal event too can cause a state transition. In addition, given an external input in some states, the next state of the FSM cannot be uniquely determined.

Off point Given a boundary, an off point is a point away from the boundary. One must consider a domain of interest and its relationship with the boundary while identifying an off point.

On-line insertion and removal test Verifies the individual module redundancy including the software that controls these modules.

On point Given a domain boundary, an on point is a point on the boundary or very near the boundary but still satisfying the boundary inequality.

Open boundary Boundary with data points that are *not* a part of the domain of interest.

Open domain Domain with all its boundaries open with respect to the domain.

Operational profile Set of operations supported by a system and their probability of occurrence. An operational profile is organized in the form of a tree structure, where each arc is labeled with an action and its occurrence probability.

Oracle Mechanism that verifies the correctness of program outputs. An oracle can be a specification, an expert, a body of data, or another program.

Original equipment manufacturer (OEM) Company that builds products or components which are used in other products sold by another company, often called a value-added reseller, or VAR. An OEM typically builds a product to an order based on the designs of the VAR. For example, hard drives in a computer system may be manufactured by a corporation separate from the company that assembles and markets computers.

Orthogonal array (OA) testing Combinatorial testing technique for selecting a set of test cases from a universe of tests and making testing efficient and effective. OA testing is based on a special matrix called a latin square in which the same symbol occurs exactly once in each row and column.

Orthogonal defect classification (ODC) Scheme for classifying software defects and guidance for analyzing the classified aggregate defect data.

Packet data serving node (PDSN) Provides access to the Internet, intranets, and application servers for mobile stations utilizing a CDMA2000 radio access network (RAN). Acting as an access gateway, a PDSN entity provides simple IP and mobile IP access, foreign agent support, and packet transport for virtual private networking. It acts as a client for an authentication, authorization, and accounting (AAA) server and provides mobile stations with a gateway to the IP network.

Pairwise coverage Requires that, for a given number of input parameters to the system, each possible combination of values for any pair of parameters be covered by at least one test case. It is a special case of combinatorial testing.

Pairwise testing Integration testing technique in which only two interconnected systems are tested in an overall system. The purpose of pairwise testing is to ensure that the two systems under consideration can function together, assuming that other systems within the overall environment behave as expected.

Parametric oracle Scheme in which an algorithm is used to extract some parameters from the actual outputs and compares them with the expected parameter values.

Pareto principle States that 80% of the problems can be fixed with 20% of the entire effort. It is also known as the 80–20 rule.

Partition testing Testing technique in which the input domain of the program is divided into nonoverlapping subdomains; next, one test input is selected from each subdomain. The basic assumption is that all the elements within a subdomain essentially cause the system to behave the same way and that any element of a subdomain will expose an error in the program as any other element in the same domain.

Path Sequence of statements in a program or a program unit. Structurally, a path is a sequence of statements from the initial node of a CFG to one of the terminating nodes.

Path predicate Set of predicates associated with a path.

Path predicate expression Interpreted path predicate.

Perfect oracle Scheme in which the system (IUT) is tested in parallel with a trusted system that accepts every input specified for the IUT and "always" produces the correct result.

Performance fault Causes a program to fail to produce the desired output within specified resource limitations.

Performance test Determines how actual system performance compares to predicted system performance. Tests are designed to verify response time, execution time, throughput, resource utilization, and traffic rate.

Ping Computer network tool used to test whether a particular host is reachable across an IP network. Ping works by sending ICMP "echo request" packets to the target host and listening for ICMP "echo response" replies. Using interval timing and response rate, ping estimates the round-trip time and packet loss rate between hosts.

Point of control and observation (PCO) Well-designated point of interaction between a system and its users.

Point-to-point protocol (PPP) Data link protocol commonly used to establish a direct connection between two nodes over serial cable, phone line, trunk line, and cellular telephone.

Power cycling test Verifies that a system consistently boots and becomes operational after a power cycle.

Power of test methods Used to compare test methods. The notion of *at least as good as* is an example of comparing the power of test methods. A test method M is at least as good as a test method N if, whenever N reveals a fault in a program P by generating a test, method M reveals the same fault by generating the same test or another test.

Power-on self-test (POST) Determines whether or not the hardware components are their proper states to run the software.

Predicate Logical function evaluated at a decision point.

Predicate coverage Exploring all possible combinations of truth values of conditions affecting a selected path for all paths.

Predicate interpretation Symbolically substituting operations along a path in order to express the predicates solely in terms of the input vector and a constant vector.

Product view of quality The central hypothesis in this view is: if a product is manufactured with good internal properties, then it will have good external qualities.

Program mutation Making a small change to a program to obtain a new program called a mutant. A mutant can be equivalent or inequivalent to the original program. Program mutation is used in evaluating the adequacy of tests.

Protected extensible authentication protocol (PEAP) Method to securely transmit authentication information, including passwords, over a wireless network.

Quality assurance (QA) (i) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements and (ii) a set of activities designed to evaluate the process by which products are developed or manufactured.

Quality circle (QC) Volunteer group of workers, usually members of the same department, who meet regularly to discuss the problems and make presentations to management with their ideas to solve the problems. Quality circles were started in Japan in 1962 by Kaoru Ishikawa as another method of improving quality. The movement in Japan was coordinated by the Union of Japanese Scientists and Engineers (JUSE).

Quality control Set of activities designed to evaluate the quality of developed or manufactured products.

Quality criterion Attribute of a quality factor that is related to software development. For example, modularity is an attribute of the architecture of a software system.

Quality factor Behavioral characteristic of a system. Some examples of high-level quality factors are correctness, reliability, efficiency, testability, portability, and reusability.

Quality management The focus of a quality management group is to ensure process adherence and customize software development processes.

Quality metric Measure that captures some aspect of a quality criterion. One or more quality metrics are associated with each criterion.

Quick polling Used to check whether a network element is reachable by doing a ping on the node using the SNMP Get() operation.

Radio access network (RAN) Part of a mobile telecommunication system. It implements a radio access technology. Conceptually, it lies between the mobile phones and the core network (CN).

Random testing Test inputs are selected randomly from the input domain of the system.

Referencing a variable A variable is said to be referenced if the value held in the variable's memory location is fetched.

Regression testing Selective retesting of a system or a component to verify that modifications have not caused unintended effects and that the system or the component still complies with its specified requirements.

Regulatory test Ensures that the system meets the requirements of government regulatory bodies.

Release note Document that accompanies a build or a released software. A release note contains the following information: changes since the last build or release, known defects, defects fixed, and added features.

Reliability test Measures the ability of the system to keep operating over an extended period of time.

Reliable criterion A test selection criterion is reliable if and only if either all tests selected by the criterion are successful or no test selected by the criterion is successful.

Remote architecture Architecture where the IUT does not have a PCO at the upper service boundary and no direct access to the lower service boundary is available.

Remote authentication dial-in user service (RADIUS) AAA protocol for applications such as network access and IP mobility.

Requirement Description of the needs or desires of users that a system is supposed to implement.

Reset sequence Input sequence that puts an implementation to its initial state independent of the state that the implementation is in before the reset sequence is applied.

Rework cost Cost of fixing the known defects.

Robustness test Verifies how robust a system is, that is, how gracefully it behaves in error situation or how it handles a change in its operational state.

Root cause analysis (RCA) Class of problem solving methods aimed at identifying the root causes of problems. The practice of RCA is predicated on the belief that problems are best solved by attempting to correct or eliminate root causes, as opposed to merely addressing the immediately obvious symptoms.

Safety assurance A safety assurance program is established in an organization to eliminate hazards or reduce their associated risks to an acceptable level.

Safety critical software system Software system whose failure can cause loss of life.

Sandwich integration Testing technique in which the software modules are integrated using a mix of top-down and bottom-up techniques.

Scaffolding Computer programs and data files built to support software development and testing but not intended to be included in the final product. Scaffolding code simulates the functions of components that do not exist yet and allow the program to execute. Scaffolding code involves the creation of stubs and test drivers.

Scalability test Verifies whether the system can scale up to its engineering limits.

Scrum Project management method for software development. The approach was first described by Takeuchi and Nonaka in "The New Product Development Game" (*Harvard Business Review*, January/February 1986). It is an iterative, incremental process for developing any product or managing any work.

Secure shell (SSH) Set of standards and an associated network protocol that allow establishing a secure channel between a local and a remote computer. SSH is typically used to log in to a remote machine and execute commands.

Secure socket layer (SSL) Protocol that provides endpoint authentication and communication privacy over the Internet using cryptography.

Security Branch of computer science which deals with protecting computers, network resources, and information against unauthorized access, modification, and/or destruction.

Serviceability Ability of technical support personnel to debug or perform root cause analysis in pursuit of solving a problem with a product. Serviceability is also known as supportability.

Shewhart cycle Also referred to as the Deming cycle after W. Edwards Deming, named after Walter Shewhart, who introduced the concept in his book *Statistical Method from the Viewpoint of Quality Control*, Dover Publications, New York, 1986. It is a continuous improvement cycle known as plan, do, check, and act (PDCA).

Shifted boundary Shifted boundary error is said to occur if the actual boundary is parallel to but not the same as the boundary of interest.

Shrink wrap Material made of polymer plastic with a mix of polyesters. When heat is applied to this material, it decreases in size so that it forms a seal over whatever it was covering. The shrink wrap provides a tamper-evident seal that helps ensure freshness and discourage pilfering. Shrink wrap is commonly found on CDs, DVDs, software packages, and books.

Simple Network Management Protocol (SNMP) Part of the IP suite as defined by the Internet Engineering Task Force. The protocol is used by network management systems for monitoring network-attached devices for conditions that warrant administrative attention.

Simulator Imitation of some real thing, state of affairs, or process. The act of simulating something generally entails representing certain key characteristics or behaviors of a selected physical or abstract system.

Six Sigma Set of practices originally developed by Motorola to systematically improve processes by eliminating defects. The term Six Sigma refers to the ability of highly capable processes to produce output within specification. In particular, processes that operate with Six Sigma quality produce at defect levels below 3.4 defects per (one) million opportunities.

Softer handoff Handoff procedure in which a user-level communication uses two sectors of a single base station simultaneously.

Soft handoff Handoff procedure in which a user-level communication uses two base stations simultaneously.

Software image Compiled software binary.

Software reliability Failure intensity of a software system operating in a given environment.

Specification and Description Language (SDL) High-level specification language which is built around the following concepts: *system*, which is described hierarchically by elements called systems, blocks, channels, processes, services, signals, and signal routes; *behavior*, which is described using an extension of the FSM concept; *data*, which are described using the concept of abstract data types and commonly understood program variables and data structures; and *communication*, which is asynchronous.

Spiral model Also known as the spiral life-cycle model, a systems development method (SDM) used in information technology (IT). This model of development combines the features of the prototyping model and the waterfall model. The spiral model was defined by Barry Boehm in his 1988 article "A Spiral Model of Software Development and Enhancement" (IEEE Computer, May 1988, pp. 61-72).

Spoilage Metric that uses defect age and distribution to measure the effectiveness of testing.

Stakeholder Person or organization that influences a system's behavior or that is impacted by the system.

Statement coverage Selecting paths in such a manner that certain statements are covered by the execution of those paths. Complete statement coverage means selecting some paths such that their execution causes all statements to be covered.

Static unit testing Non-execution-based unit testing. In static unit testing, a programmer does not execute the unit; rather, it involves formal review or verification of code.

Statistical oracle Special case of parametric oracle in which statistical characteristics of the actual test results are verified.

Statistical testing Testing technique which uses a formal experimental paradigm for random testing according to a usage model of the software. In statistical testing a model is developed to characterize the population of uses of the software, and the model is used to generate a statistically correct sample of all possible uses of the software.

Stress test Evaluates and determines the behavior of a software component when the offered load is in excess of its designed capacity.

Stub "Dummy subprogram" that replaces a module that is called by the module to be tested. A stub does minimal data manipulation, such as print verification of the entry, and returns control to the unit under test.

Sustaining phase Optimizing and refining software that is working and focusing much more solidly on customers and competitors to ensure that one does not lose what has been acquired.

Sustaining test engineer Test engineer responsible for testing the product in its sustaining phase.

System integration test (SIT) Testing phase in which software components, hardware components, or both are combined and tested to evaluate their interactions.

System resolution test Probes to provide definite diagnostic answers to specific requirements.

System testing Comprehensive testing undertaken to validate an entire system and its characteristics based on the requirements and the design.

Technical beta Testing conducted to obtain feedback about the usability of the product in a real environment with different configurations. The idea is to obtain feedback from a limited number of users who commit considerable amount of time and thought to their evaluation.

Telnet Network-based application that is used to provide user-oriented command line login sessions between hosts on the Internet.

Testability requirement Requirement that it is possible to construct a test objective which will determine if a system property has been satisfied.

Test adequacy Goodness of a test. If a test does not reveal any fault in a program, it does not mean that there are no faults in the program. Therefore, it is important to evaluate the goodness of a test.

Test architecture Abstract architecture described by identifying the points closest to the IUT at which control and observation are specified. The abstract test architectures can be classified into four major categories: local, distributed, coordinated, and remote.

Test automation Using test tools to execute tests with little or no human intervention.

Test case Pair of input and the expected outcome. A test case covers a specific test objective.

Test case design yield (TCDY) Commonly used metric to measure the test case design effectiveness.

Test case effectiveness Measure of the quality of test cases in terms of their fault revealing capability.

Test case escaped Sometimes defects are found in the testing cycle for which there are no test cases designed. For those defects new test cases are designed, which are called test case escaped.

Test case library Compiled library of reusable test steps of basic utilities that are used as building blocks to facilitate the development of automated test scripts.

Test coordination procedure Set of rules to coordinate the actions of the upper and the lower testers.

Test cycle Partial or total execution of all the test suites planned for a given system testing phase. System testing involves at least one test cycle.

Test data Element of the input domain of a program. Test data are selected by considering some selection criteria.

Test-driven development (TDD) Software development methodology in which programmers write unit tests before the production code.

Test driver Program that invokes a unit under test, passes inputs to the unit under test, compares the actual outcome with the expected outcome from the unit, and reports the ensuing test result.

Test effectiveness Measure of the quality of the testing effort.

Test effort Metric specifying the cost and the time required to create and execute a test case in person-days.

Test environment Setting in which system tests are executed. It is also known as a test bed.

Test event Atomic interaction between the IUT and an upper or lower tester.

Test first Software development methodology in which the programmers write unit tests before the code.

Testing maturity model (TMM) Gives guidance concerning how to improve a test process. The maturity of a test process is represented in five levels, or stages, namely, 1–5. Each stage is characterized by the concepts of maturity goals, supporting maturity goals, and activities, tasks, and responsibilities (ATRs).

Testing and test control notation (TTCN) Programming language dedicated to testing of communication protocols and web services. Up to version 2 the language was unconventionally written in the form of tables, and the language used to be called Tree and Tabular Combined Notation (TTCN) and was renamed to Testing and Test Control Notation in version 3.

Test management protocol Protocol used to implement test coordination procedures by using test management protocol data units (TM-PDUs) in the coordination architecture.

Test objective Description of what needs to be verified in order to ensure that a specific requirement is implemented correctly.

Test oracle Can decide whether or not a test case has passed. An oracle provides a method to (i) generate expected results for the test inputs and (ii) compare the expected results with the actual results of the implementation under test.

Test predicate Description of the conditions or combination of conditions relevant to the correct operation of a program.

Test prioritization Ordering the execution of test cases according to certain criteria.

Test process Certain manner of performing activities related to defect detection.

Test process improvement (TPI) model Allows one to evaluate the maturity levels of test processes. The current status of a test process is evaluated from 20 viewpoints, known as key areas. The status of a test process with respect to a key area is represented in terms of one of four levels of maturity—A, B, C, and D. Level A is the lowest level of maturity, and maturity level ascends from A to D.

Test purpose Specific description of the objective of the corresponding test case.

Test selection Carefully selecting a subset of the test suites on the basis of certain criteria. A chosen subset of the test suites are used to perform regression testing.

Test selection criterion Property of a program, a specification, of a data domain.

Test suite Group of test cases that can be executed as a package in a particular sequence. Test suites are usually related by the area of the system that they exercise, by their priority, or by content.

Test tool Hardware or software product that replaces or enhances some aspect of human activity involved in testing.

Test vector Also called test input vector, an instance of the input to a program.

Tilted boundary Error that occurs if the actual boundary intersects with the intended boundary.

Top-down integration A kind of integration testing technique in which testing starts at the topmost module of the program, often called the "main program," and works toward the outermost branches of the visibility tree, gradually adding modules as integration proceeds.

Total quality control (TQC) Management approach for an organization centered on quality and based on the participation of all its members and aiming at long-term success through customer satisfaction and benefits to all members of the organization and to the society. Total quality control was the key concept of Armand Feigenbaum's 1951 book, *Quality Control: Principles, Practice, and Administration*. Republished in 2004 as *Total Quality Control*, McGraw-Hill, New York.

Traceability matrix Allows one to make a mapping between requirements and test cases both ways.

Transcendental view of quality Quality that can be recognized through experience but not defined in some tractable form.

Transfer sequence Minimum-length input sequence that brings an implementation from its initial state into a given state.

Transition tour State transitions defined in an FSM specification are executed at least once by applying an input sequence to an implementation, starting from the initial state of the FSM. Such an input sequence is called a transition tour of the FSM.

Transmission Control Protocol (TCP) Core protocol of the IP suite. Applications on networked hosts can create connections with one another using the TCP; data segments are transmitted over a TCP connection for higher reliability. The protocol guarantees reliable and in-order delivery of data segments. TCP supports many of the Internet's popular applications, including the World Wide Web, e-mail, and secure shell.

Transport layer security (TLS) Provides endpoint authentication and communication privacy over the Internet using cryptography.

Tunneled transport layer security (TTLS) Similar to the TLS protocol, but client authentication is extended after a secure transport connection has been established.

Undefinition of a variable A variable is said to be undefined if the variable's memory location holds a value which is not meaningful anymore.

Unique input–output (UIO) sequence Essentially an input sequence such that the corresponding output sequence uniquely identifies the state that an implementation was in before the UIO sequences was applied.

Unified modeling language (UML) Standardized specification language for object modeling. UML is a general-purpose modeling language that includes a graphical notation used to create an abstract model of a system.

Unit Program unit or module that may be viewed as a piece of code implementing a "low"-level function.

Unit testing Testing a program unit in isolation. Unit testing is performed by the programmer who wrote the program unit.

Unit under test Program unit that is being tested in the context of an emulated environment.

Upgrade/downgrade test Verifies that the system software build can be upgraded or downgraded.

Upper tester Tester entity that controls and observes the upper service boundary of the IUT.

Usability test Means of measuring how well people can use some human-made object, such as a web page, a computer interface, a document, or a device, for its intended purpose.

Usage profile Software profile that characterizes operational use of a software system. Operational use is the intended use of the software in the intended environment.

User acceptance testing (UAT) Conducted by the customer to ensure that the system satisfies the contractual acceptance criteria.

User view of quality Extent to which a product meets user needs and expectations.

Valid criterion A test selection criterion is valid if and only if whenever a program under test contains a fault, the criterion selects a test that reveals the fault.

Validation Process of ensuring that the software meets its customer's expectations.

Value-based view of quality The central idea in the value-based view is how much a customer is willing to pay for a certain level of quality.

Verdict A test verdict is a statement of pass, fail, or inconclusive that is associated with a test case. Pass means that the observed outcome satisfies the test purpose and is completely valid with respect to the requirement. Fail means that the observed outcome is invalid with respect to the requirement. An inconclusive verdict means that the observed outcome is valid with respect to the requirement but inconclusive with respect to the test purpose.

Verification Process of ensuring the correspondence of an implementation phase of a software development process with its specification.

Virtual circuit (VC) Communication arrangement in which data from a source user may be passed to a destination user over more than one real communication circuit during a single period of communication; the switching is hidden from the users. A permanent virtual circuit (PVC) is a virtual circuit established for repeated use between the same data terminal equipments (DTE). In a PVC, the long-term association is identical to the data transfer phase of a virtual call. Permanent virtual circuits eliminate the need for repeated call setup and clearing. On the other hand, switched virtual circuits (SVCs) are generally set up on a per-call basis and are disconnected when calls are terminated.

Virus Software component that is capable of spreading rapidly to a large number of computers but cannot do so all by itself. It has to spread using the assistance of another program.

Walkthrough Review where a programmer leads a review team through a manual or simulated execution of the product using predefined scenarios.

Waterfall model Sequential software development model in which development is seen as flowing steadily downward—like a waterfall—through the phases of requirements analysis, design, implementation, testing, integration, and maintenance. The origin of the term "waterfall" is often said to be an article (Managing the Development of Large Software Systems: Concepts and Techniques, in Proceedings of WESCON, August 1970, pp. 1-9. Reprinted in ICSE, Monterey, CA, 1987, pp. 328-338) published in 1970 by W. W. Royce.

White-box testing Testing methodology in which one primarily takes into account the internal mechanisms, such as code and program logic, of a system or component.

Worm Software component that is capable of, under its own means, infecting a computer system.

***W*-set** Set of input sequences for an FSM. When the set of inputs is applied to an implementation of the FSM in an intended state, one expects to observe outputs which uniquely identify the state of the implementation.

Zero-day attack Presents a new and particularly serious kind of threat. Developed specifically to exploit software vulnerabilities before patches are available, these attacks are not recognized by traditional security products: they enter a network undetected, giving absolutely no time to prepare for a defense.

INDEX

One of the great defects of English books printed in the last century is the want of an index.

— *Lafcadio Hearn a.k.a Koizumi Yakumo*

1+1 redundancy, 207
1xEV-DO, 207–208
2-way tests, *See* pairwise testing
3-way tests, 235
4-way tests, 235
80/20 ratio, 507
80/20 rule, *See* Pareto analysis
AAA server, 208
ability to perform, 553
Abnormal (A), 114–115
abstract syntax notation one (ASN.1), 201
ACC, *See* acceptance criteria change
acceptance
  beta, 435
  criteria, 451, 461
  criteria change (ACC), 464
  test engineer, 464
  test execution, 463–464
  test in XP, 466–467
  test plan, 461–463
  test report, 464–466
  testing, 17, 450
access
  audit, 529
  control, 529
  terminal (AT), 208
accident, *See* mishap
accomplish, 508
accounting management, 199
accuracy, 453, 529, 531
acoustic test, 176
actions , 365
active mode, 207
activities performed, 553
activities, tasks, and responsibilities (ATRs), *See* ATRs
activity, 428
actual behavior, 19
actual outcome, 15
adaptability, 533
adapters, 185
adaptive maintenance, 440–441
adaptive random testing, 254–255
adequacy
  criteria, 47
  criterion, 44
  of testing, 42–45
  properties, 47
adequate, 47
adjacent domain, 136, 142
adjustable, 508
Administrator, 401
AETG, 260
age, 429
agile software development, 18, 22
airlink, 208
Albrecht, A. J. , 380
all path coverage, 96–97
all-combination testing, 235
all-c-uses, 122
all-c-uses/some-p-uses, 123
all-defs , 121
all-du-paths, 123
all-pair testing, *See* pairwise testing
all-p-uses, 122
all-p-uses/some-c-uses, 123

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

all-uses, 123
analyzability, 533
approximate solution, 144
arrays, 227–228
arrival and resolution of defects (ARD), 425
ASN.1, *See* abstract syntax notation one
ASP, 291
aspect-oriented def-use, 131
aspect-oriented programs, 131
assertions, 61
Assign State, 328
Assigned, 410
assumptions, 364
asynchronous transfer mode (ATM), 335
ATRs, 569
attractiveness, 533
authentication, 199, 208, 458
author, 56
authorization, 199, 208, 394
automatable, 403
automated teller machine (ATM), 313–315
automation progress, 417
automation test group, 498
availability, 203, 456, 458

B mark, 217
backdoors, 204
backup , 454
bad fix , 418
base station controller (BSC), *See* radio node controller
base transceiver station (BTS), *See* radio node
basic input-output system (BIOS), 175
basic
  interconnection tests, 197
  model, 487–488
  testing, 571
  tests, 193, 194–196
BAT, *See* business acceptance testing
Bayonet project, 426–428
Bazooka project, 420–423
Beck, Kent, 73
behavior tests, 197
Beizer, B., 131, 110
BERT, *See* bit error test
beta release criteria, 436
beta testing, 435–436
big-bang, 173
BIOS, *See* basic input-output system
BIT, *See* built-in test
bit error test (BERT), 196
black-box
  component testing, 185
  testing, 20–21, 163
Blocked, 349
Boomerang, 442–444
boot tests, 194
boring and time consuming, 396
bot, 459
bottom-up, 171–173
bound checker , 77
boundary
  inequalities, 141
  tilt, 149, 152
  value analysis (BVA), 246–248
  value tests, 205
brainstorm, 433
branch coverage, 98–99, 127
breakpoint, *See* debugger
breakpoints debugger, *See* debugger
BSMI mark, 217
buffer allocation, 211
buffers overflow, 60
buffers underflow, 60
build, 165
built-in test (BIT), 186–187
Burnstein, Ilene, 568
business acceptance testing (BAT), 17, 450
business goal, 461
BVA, *See* boundary value analysis

calendar time, 474
candidate set, *See* adaptive random testing
capability maturity model (CMM), 5, 547–548
capability tests, 197
capture the response, 399
Card, David N., 432
career path, 511–512
carry through, 508
categories, 256–257
categorization of errors, 433
category partition method (CPM), 256–258
causal analysis , *See* defect causal analysis
cause-and-effect diagram, *See* Ishikawa diagram
CBA-IPI, 553
CCB, *See* change control board

CCC mark, 217
CE mark, 216–217
certification of COTS components, 188
CFG, *See* control flow graph
Chainsaw project, 430
change control board (CCB), 178
change request (CR), 54, 60
changeability, 533
characterization of test cycles , 362
characterizing sequence (CS), 287–290
characterizing set, 287
check points, 563
check-in request form , 166
Chillarege, Ram, 9
choice construction, 41
choices, 256–257
cleanroom software engineering, 19
cleanup, 346, 397, 400
CLI, *See* command line interface tests
clock time, 474
Closed, 410
closed boundary, 142
closed domain , 142
closed inequality boundary, 147–150
Closed State, 330
closure error, 143, 150, 152
CMM, *See* capability maturity model
CMM appraisal framework (CAF), 554
CMMI, 554–555
CMM-SE, 555
CMM-SS, 555
CMM-SW, 555
code auditor, 77
code review, 54
code review checklist, 58
co-existence, 533
coincidental correctness, 107, 261
collateral damage, 71, 376, 418
combinatorial testing, 259
command line interface (CLI) tests, 196
commercial off-the-shelf (COTS)
    component, 186
Commit State, 328
Commitment, 560
commitment to perform, 553
common features, 553
communication, 561
  commonality, 529
  systems tests, 196–197
communicativeness, 529
compatibility, 459
compatibility matrix, 178
compatibility tests, 209
competent, 502
competent programmer hypothesis, 68
competitive edge, 454
complete path, 121
complete testing, 13
completely specified, 279
completeness, 452, 529, 459
compliance
  information, 329
  marks, 216
  statistics, 437
  tests, 357
component, 158, 174, 298–299
component software, 174
comprehensive, 502
computation error, 135
conciseness, 529
confidentiality, 203, 458
configuration
  management, 199
  management system, 80–81, 84, 551
  maximum, 194
  minimum, 194
  tests, 209
conformance testing, 273
considerate, 502
consistency, 514, 529
constraints, 258
construction fault, 35
continual improvement, 536
control dominating system, 266–267
control flow
  definition, 20, 89
  graph (CFG), 64, 89, 90–93
  testing, 64, 89
control testing process, 574
controlled, 502, 506
coordinated architecture, 294
coordinating interview, 506–507
corrective
  action, 434
  maintenance, 440–441
  measures, 419
correctness, 452, 523–524
correctness of acceptance tests, 467
cost, 557
cost of testing, 574

COTS, *See* commercial off-the-self component
coupling effect, 68
coverage
  definition, 14
  all path, 96–97
  fault, 290
  matrix, 321–322
  metrics, 321
  pairwise, 240
  PCO, 313
  predicate, 100–101
  state, 276, 308
  statement, 97–98
  transition, 276, 308
CPM, *See* category partition method
CPPUnit, 73
cpu time, 209
cpu utilization, 211
CR, *See* change request
crash defects (CD), 424
Create state, 345
criteria
  acceptance, 451, 461
  adequacy, 47
  beta release, 436
  data flow testing, 121–124
  dual-use variables, 226
  entry, 368
  exit, 366
  FCS readiness, 437
  GUI testing capability, 395
  input variables , 225
  McCall's quality, 529
  multiple-type variables, 226–227
  output variables, 225–226
  path selection, 89
  path selection, 94–101
  pricing , 395
  quality, 6, 527–528
  revert and extension, 365
  system integration entry, 182
  system integration exit, 182
  test adequacy for GUI, 219
  test development, 393
  test execution, 394
  test maintenance, 394
  test management, 394
  test results, 394
  tool evaluation, 393
  vendor qualification, 395
Critical, 410, 502
Crosby, Philip, 5, 431
cross-functionality
  groups, 409
  tests, 204
CSA mark, 216–217
C-Tick mark, 217
cumulative failure, 476
CUPRIMDS, 461
current vector, 280
c-use, 117
customer, 450, 463
customer focus, 535
CUTE, 79
cyclomatic complexity, *See* McCabe complexity measure

daily build, 165
DART, 79
data commonality, 529
data conversion, 453–454
data conversion testing, 204
data declarations, 296
data dominating system, 265–266
data flow
  definition, 20
  analysis, 109–110
  anomaly, 113–115
  graph (DFG), 64, 116–119
  testing, 64, 115–116, 127
  testing criteria, 121–124
data integrity, 453
DAVE, 129
DCA, *See* defect causal analysis
dd (type1) anomaly, 113–115
debriefing meeting, 511
debuggers, 77
debugging
  definition, 68, 77
  backtracking, 69
  brute force, 68
  cause elimination, 69
  goals, 569
  heuristic, 69–70
decision point, 91
decision tables , 248–252
Decline State, 330
decrement process, 487
decryption, 203

def-clear, 122
defeat a test, 39
defect
  definition, 9–10
  age, 442
  causal analysis (DCA) , 431
  count, 522
  density, 522
  dynamic model, 442
  management, 561–562
  prevention, 60, 432, 552, 576
  priority, 410
  removal efficiency (DRE), 441
  report, 418
  review meeting, 419
  schema, 412
  severity, 410–411
  tracking system, 464
  type, 429
  arrival rate (DAR), 424
  closed rate (DCR), 424
  rejected rate (DRR), 424
Define (d), 114–115
Defined and referenced (R), 114–115
Defined but not referenced (D), 114–115
definition clear-path, 119
definition of maturity, 554
degraded node tests, 207
Deleted state, 346
Deming prize, 2
Deming, W. Edwards, 2, 4, 5
denial-of-service (DoS), 458
dependability, 468
dependable, 508
Deprecated state, 347
design fault, 35
design verification test (DVT), 174
determine the verdict, 399
determined, 509
deterministic, 279
develop testing, 569
development rework cost, 523
development test group, 498
diagnostic test, 175, 195–196
digital signature, 453
discipline, 554
distance between programs, 47
distinguishing sequence (DS), 284–287
distributed architecture, 293
DMADV, 579
DMAIC, 579
documentation, 174, 461
documentation tests, 194, 215–216
documented, 398
documenters, 77
domain
  definition, 136
  error, 64, 136–137
  testing, 64, 137–138
dominance, 108
dormant defect, 442
DoS, *See* denial-of-service
Draft state, 345
DRE, *See* defect removal efficiency
drive the test, 399
DS-tree, 285, 288
du (type3) anomaly, 114–115
du-path, 121
Duplicate, 410
DVT, *See* design verification test
dynamic analysis, 7
dynamic unit testing, 53, 62–64

EAP, 204
easy to automate, 396
EC, *See* engineering change
Eclat, 76
ECO, *See* engineering change order
effective domains, 155
effectiveness of test cases, 11
efficacy of testing, 45
efficiency, 524, 531
efficient, 566–567
EFSM, 302–307
EGT system, 79
electrical test, 176
electromagnetic emission test , 175
electrostatic discharge (ESD) test, 175
element management systems (EMS) tests, 198–202
EMC compliance, 216
emission and immunity, *See* EMC compliance
EMS, *See* element management system tests
emulators, 77
encryption, 203
end-to-end testing, 163
end-to-end validity, 183
endurance, 184

engineering change (EC), 329, 437
engineering change order (ECO), 177–180
enlarged domain, 148–149, 151–152
entry criteria, 182
entry node, 93–94, 118
entry point, 88
environmental test, 176
equality boundary, 153–154
equipment handling, 176
equivalence class, 244–245, 256
equivalence class partitioning , 244–246
error
  definition, 9
  guessing, 20, 255–256
  tolerance, 529
ESD, *See* electrostatic discharge test
Estelle, 18, 130
estimating, 559
Ethernet loop back tests, 196
evaluate software quality, 576
evaluation, 562
events, 291
exact solution, 144
exception handler routine, 61
executed set, *See* adaptive random testing
execution status of test (EST), 420
execution efficiency, 529
execution environment, 478
execution time , 209, 473
exhaustive testing, 39
exit criteria, 182, 366
exit node, 93–94, 118
exit point, 88
expandability, 529
expectation, 513
expectation dependent , 42
expected outcome, 15
exterior point, 145
external evaluations, 553
external test architectures, 291
extreme point, 142
eXtreme Programming (XP), 8, 18, 22, 71, 466, 481

factorial, 19
factors, 236–237
factual approach, 537
FAD, 410
Fagan, Michael, 54
Failed, 349
failure, 9, 472–473
failure containment, *See* degraded node tests
failure intensity, 476–478, 487
false negative, 399, 453, 502
false positive, 399, 453, 502
fast learner, 509
fault
  definition, 9, 472–473
  coverage, 290
  injection, 20, 68, 84
  management, 199
  seeding, 20, 45, 441–442, 447
  simulation, 20
  tolerance, 531, *See also* high availability tests
  based testing, 20
  free program, 45
FCC mark, 216–217
FCS, *See* first customer shipment
FCS blocker, 437
FCS readiness criteria , 437
feasible
  definition, 90
  data flow, 126
  entry-exit path , 98
  path, 126, 135
feature
  definition, 321
  tests, 204
Feigenbaum, Armand V., 3
finite-state machine (FSM), 18, 270–271
Finkelstein, A. C. W., 325
first customer shipment (FCS), 437
fishbone diagram, *See* Ishikawa diagram
five-C characteristics, 502
FLASH card, 194
Flexibility, 525
Ford, Henry, 3
formal verification, 83
frame relay (FR), 335
FrAtm, 336
FSM, *See* finite-state machine
full polling, 213
function, 222
function as designed (FAD), *See* FAD
function as designed count, 423
function points, 380
functional specification, 18, 56
functional testing, 20–21, 64, 222–232

functional validity, 183
functionality
  definition, 531
  tests, 193, 196–204
functionally related, 230
functions and training, 561
fundamental theorem, 33
fUnit, 73

Galin, D., 544
Gantt chart, 389
Garvin, D. A., 5, 519
generality, 529
Get, 201
GetBulk, 201
GetNext, 201
Gilb's technique, 521
global c-use, 119
glue, 185
goal-oriented, 509
goals, 363
gold standard oracle, 253
GOST-R certification, 217
Gotel, O. C. Z., 325
graphical user interface (GUI)
    tests, 202
green bin, 370
GUI, *See* graphical user interface tests
GUI testing capability criteria , 395
guidelines for automation, 395–396

HALT, *See* highly accelerated life tests
Hamlet, R., 10
hardware independence, 529
hazards, 216
High, 410
high availability tests, 206–207
high risk, 396
high-level design, 56
highly accelerated life tests, 177
Hold, 410
homogeneous state block, 285
homogeneous vector, 280
Howden, William E., 222, 253
Humphrey, Watts S., 432
hybrid analysis, 7
hypothesis, 70

ideal test, 32
ideal test selection, 38
IETF, 534
impact, 429
implementation under test (IUT), 291
Implement State, 328
implications, 108
IMSL, 222
in parameter order (IPO), 240–243
inappropriate action, 36
inappropriate path selection, 35
in-circuit emulators, 77–78
includes, 124
incremental, 164
independent, 398
independent test group, 572
infeasible, 90, 104, 126
infeasible path, 107–108, 135
inferred requirements, 335
information sharing, 514
in-house training, 512
initial, 569
initial path vector, 280
initial state block, 285
initial vector, 280
inject faults, 107
innovative, 509
inopportune input, 13
in-process metrics, 419, 446
input
  classifier, 136
  condition, 244
  data, 106
  domain, 14, 19, 64, 135, 244–245, 252, 233
  sequence, 279
  vector, 101–102
inspection, 54
installability, 459–460, 533
installation procedure, 215
instrumentation, 529
instrumentation code, 60
integrate testing into software life cycle, 572–574
integrated software management, 551
integration of hardware/software, *See* software/hardware integration
integration test group, 26, 496
integration testing, 17, 158–159
integrity, 203, 525
integrity checking, 454
intended behavior, 19

interacting quality processes, 537
interactive debuggers, *See* debuggers
interface
  errors, 160–162
  integrity, 183
  stress, 183
intergroup coordination, 552
interior point, 145
inter-method testing, 130–131
internal assessments, 553
interoperability, 459, 526, 531
interoperability tests, 193, 208–209
interprocedural data flow testing, 131
intersystem testing, 163
interviewing, 507
intraclass testing, 130–131
intramethod testing, 130–131
intrasystem testing, 163
Invalid , 349
invalid input, 13, 244
invariants, 61
involvement of people, 536
IPD-CMM, 555
IPO, *See* in parameter order
IPSec, 204
IRAM safety mark, 217
Irreproducible, 410
irreproducible defects (IRD), 424
Ishikawa diagram, 4, 431, 433–434
Ishikawa, Kaoru, 3–4, 433
ISO
  900 principles, 535–537
  9001, 535
  9001 requirements, 537–541
  9126, 6, 521, 530–534
IUT, *See* implementation under test

job profiling, 505
job requisition, 504–505
Jones, Capers, 384
junior engineers, 501
JUnit, 73
Juran, Joseph, 3, 431
JUSE, *See* Union of Japanese Scientists and Engineers
just in time, 377

key areas, 558, 564–565
key process areas (KPAs), 549–550
Kitchenham, B., 5
Kohavi, Z., 288
Koomen, T. and Pol, M., 556
Korat, 76
KPAs, *See* key process areas

LDAP, 204
lead time, 556–557
leadership, 535
lean principle, 2
learnability, 521,533
LED, *See* light emitting diode tests
less than perfect, 377, 461
less volatile, 396
levels, 237
life cycle
  model, 559
  of a defect, 366–367, 409
  of an individual requirement, 323–324
  of a test case, 345
light emitting diode tests, 195
limitation of testing, 45–46
Lipton, Dick, 65
load and stability tests, 194
load tests, 213
local architectures, 291–292
log the verdict, 400
logarithmic model, 489
logging and tracing tests, 198
logic fault, 34
loop-free path, 121
LOTOS, 316
Low, 410
lower tester, 292–293
low-level design, 56
low-level testing, 562

machine identification, *See* distinguishing sequence
maintainability, 457,525,531
maintainable, 398
maintenance, *See* software maintenance
maintenance mode, 187
Malcolm Baldrige National Quality Award, 5
malleable, 508
management and measurement, 575
management information base (MIB) , 200, 202
management requirements, 538
manually difficult, 396

manufacturing view, 6, 452, 520
marketing beta, 435
Martin, Robert C., 72
MATLAB, 227
matrix
  coverage, 321–322
  traceability, 325, 358
maturity, 531
maturity goals , 569
maturity
  level, 549–551
  level 1: initial, 550
  level 2: repeatable, 551
  level 3: defined, 551
  level 4: managed, 552
  level 5: optimizing, 552
maturity models, 546
maturity subgoals, 569
McCabe complexity measure, 79, 108
McCall, J. A., 523, 527, 529
McCall's quality
    criteria, 529
    factors, 524
MD, *See* message digest
mean number of failures, 487
mean time between failures (MTBF), 177, 214, 456, 475
mean time to failure (MTTF), 214, 475
mean time to repair (MTTR), 214, 457, 475
measurement and analysis, 553
measuring quality, 521
measuring test execution, 22–23
Medium, 410
memory
  carving, 211
  leak, 211
  leak detectors, 78
  usage, 209
  utilization, 211
mentoring, 512–513
merge point, 91
message digest (MD), 453
message passing interface, 160
meticulous, 508
metrics
  spoilage, 442–443
  test, 559–560
  test effectiveness, 441
  code review, 60
  coverage, 321
  monitoring defects, 419
  monitoring test execution, 419
  test case design effectiveness, 350–351
  test case effectiveness , 23
  test design preparedness, 349–350
  test effort effectiveness, 23
MIB, *See* management information base
MIC mark, 217
milestones, 2, 53, 387
Mills, H. D., 19, 447
mishap , 216
missing control-flow path, 35
missing paths, 21
mitigating action, 434
modeling test results, 347–349
moderator, 56
modular, 397
modularity, 529
module, 158, 296
module tests, 197–198
moment of involvement, 559
monitor testing process, 574
monitoring test execution, 22–23
Mothra, 68
motivation, 560
MTBF, *See* mean time between failures
MTTF, *See* mean time to failure
MTTR, *See* mean time to repair
multiple-experiments, 288
Musa, John D., 10, 20, 492–493
mutant
  definition, 65
  dead, 65
  equivalent, 65
  killable , 65
  killed, 65
  stubborn, 65
mutation analysis, 20, 65
mutation score , 65
mutation testing, 65
mutation adequate, 65
mutually beneficial, 537
Myers, G., 68

NE, *See* network element
network analyzers, 80
network element (NE), 198
network management station (NMS), 200
New, 410
next-state function, 277–278

NMS, *See* network management station
NOM mark, 217
nondeterministic finite-state machine (NFSM), 316
nonlinear predicate, 155
nonterminal, 281, 285
non-testable, 46
normal mode, 187
NTLM, 204
numeric variable, 224
NUnit, 73

OA, *See* orthogonal array
objective of testing, 10–11
object-oriented programs, 51, 130
observable failure, 207
observation, 271
observers, 57
ODC, *See* orthogonal defect classification
OEM, *See* original equipment manufacturer
OFF point, 145
office environment, 560
off-the-shelf (OTS) component, 184–187
Ohna, Taiichi, 2
OIR, *See* on-line insertion and removal tests
omniscient debugger (ODB), *See* debuggers
ON point, 144
on-line help, 215
on-line insertion and removal tests, 206
ON-OFF point, 155
on-site commercial training, 512
Open, 410
open boundary, 142
open domain, 142
open inequality boundary, 150–153
Open State, 326
openfiles(), 91–92, 96
operability, 521, 529, 533
operation rework cost, 523
operational profile, 10, 19, 457, 482–484
operational system testing, 186
optimization, 567, 576
oracle
  definition, 12, 45, 253
  gold standard, 253
  parametric, 253
  perfect, 253
  statistical, 253–254
organization process definition, 551
organization process focus, 551
original equipment manufacturer (OEM), 178, 435, 451
orthogonal array (OA), 236–240
orthogonal defect classification (ODC), 428–431
open system interconnection (OSI), 291
OTS, *See* off-the-shelf component
output domain , 19, 64, 233
output function, 277–278
output sequence, 287–289
outstanding defects (OD), 424

packaging, 176
packet data serving node (PDSN), 208
pair programming, 73
pairwise
  coverage, 240
  testing, 163–164, 235
  validity, 183
parametric oracle, 253
Parasoft, 110
Pareto analysis, 430–431, 433
Pareto, Vilfredo, 431
Parhami, Behrooz, 9
partition testing, 260
partition the input domain, 39
Passed, 349
path
  definition, 88–89, 135
  predicate, 102, 142
  predicate expression, 104
  selection criteria, 89, 94–101
  vector, 280
PCMCIA card, 194
PCO
  definition, 269
  coverage, 313
PDCA cycle, *See* Shewhart cycle
PDU, *See* protocol data unit
PEAP, 204
peer review, 552
perfective maintenance, 440–441
perfect oracle, 253
performance
  characteristics, 455
  fault, 35
  management, 199
  monitors, 79
  test group, 498

tests, 193, 209–210
permanent virtual connection (PVC), 335
perturbation
analysis, 84
technique, 155
testing, 83–84
PhAge, *See* defect age
phase definition, 569
plan-do-check-act (PDCA) cycle, *See* Shewhart cycle
planned versus actual execution (PAE), 420
planning, 559
point of control and observation (PCO), *See* PCO
point-to-point protocol (PPP), 208, 238
portability, 525–526, 531
ports, 298–299
POST, *See* power-on self test
post-conditions, 61
Postponed, 410
power cycling tests, 206
power-on self test (POST), 195–196
precise, 508
pre-conditions, 61
predicate coverage, 100–101
predicate interpretation, 103
presenter, 56
preventive
action, 52, 434
measures, 60, 419
pricing criteria , 395
principal engineers, 501
prioritization
a test maturity matrix, 563
of requirements, 326
of test cases, 371–372
probability of failure-free operation, 477
procedure call interface, 159
process approach, 536
process change management, 552
product
operation, 526–527
revision, 526–527
transition, 526–527
view, 6, 452, 520
program
dependent, 41
inference, 47
mutation, 45
path, *See* path
adequate, 47
programming, 56
protocol data unit (PDU), 202, 291, 297–298
prototyping model, 481
PSB mark, 217
public forum training, 512
p-use, 117
push-to-talk, 209
PyUnit, 73

QA, *See* quality assurance
QC, *See* quality circle
QMS, *See* quality management system
Qualifier, 429
qualitative segments, 566
quality
assurance (QA), 499
attribute, 461
characteristics, 6, 468, 530–531
circle (QC), 4, 431
control , 500
controller, 500
criteria , 6, 527–528
evaluate software, 576
factors, 6, 523–528
management group, 499
management system (QMS), 537
McCall's criteria, 529
McCall's factors, 524
measuring, 521
metrics , 530
model, 6
objective, 537
of the system, 556
policy, 537
software, 5–7, 519, 530
subcharacteristics, 532
quantitative process management, 552
quick polling, 213
quick study, 509

radio access network (RAN), 208
radio node (RN), 207
radio node controller (RNC), 207
RADIUS, 204
random testing, 10, 127, 252–254
Rao, C. R., 236
RCA, *See* defect causal analysis
reaching definitions, 131

read-only memory (ROM), 175, 194
realization requirements, 540
recognition, 515
recordkeeper, 56
recoverability, 531
recovery, 454
recruiting process, 504
red bin , 369
reduced, 279–280
reduced domain, 147–148, 150–151
reduced flow graph, 108
Rees, Roderick, 9
Reference (r), 114–115
regression test, 17, 194, 214–215, 404–405, 418
regulatory tests, 194, 216–218
relationship building, 509
release note, 166, 216
Released state, 346–347
reliability, 456, 471–472, 524, 531
reliability models, 486–491
reliability tests , 177, 194, 214
reliable criterion, 33, 38
remedial requirements , 541
remote architecture, 295
repeatability, 396
repeatable process, 546
repeatable results, 546
replaceability, 533
reporting, 561
requirement
  definition, 56
  engineering, 326
  fault, 34
  identification, 322–330
  inferred, *See* inferred requirements
  life-cycle, 323
  management, 551
  of a system, 36
  schema , 324
  traceability, 325
reset sequence, 278
Resolved, 410
resource requirements , 539
resource utilization, 209, 533
response time, 209
results driven, 509
retaining testers, 511
returnaverage(), 92, 97, 118
reusability, 526
reusable, 398
revealing criterion, 39
revert and extension criteria, 365
Review State, 327, 346
review
  code, *See* code review
  functional specification, 332
  program, 575
  requirement, 331–332
  system test plan, 356
reviewers, 56
reward system, 513
rework-cost, 522
robust and reliable, 397
robustness
  tests, 193, 204–208
  of a system, 458
ROM, *See* read-only memory
root cause
  analysis (RCA), *See* defect causal analysis
  defect analysis, *See* defect causal analysis
runs, 237

SA, *See* safety assurance
SABS mark, 217
safety assurance (SA), 218
safety tests, 177
scaffolding, 62
scalability
  of a system, 460
  test group, 498
  tests, 193, 210–211
schedule, 574
scheduling system testing, 387
scope of methodology , 561
screening resume, 505–506
Scrum, 18, 481
scrupulous, 508
SDL, 18, 303
secure shell (ssh), 401
security
  definition, 531
  management, 199
  tests, 203–204
selection criteria
    for dual-use variables, 226
    for input variables, 225
    for multiple-type variables, 226–227
    for output variables, 225–226

self-documentation, 529
self-sufficient, 398
senior engineers, 501
sequential computation, 91
service interworking, 336, 379, 390
serviceability, 458
Set, 201
setup, 346, 397, 399
shared memory interface, 159–160
Shelved, 410
Shewhart cycle, 2
Shewhart, Walter Andrew, 3
shifted-boundary error, 143
shrink-wrap, 366
silent fail-over , 207
simple, 397
simple network management protocol (SNMP), 200–202
simple path, 120
simplicity, 529
simulators, 80
singleton state block, 285
singleton vector, 280
SIT, *See* system integration testing
Six Sigma Maturity Model, 579
SNMP, *See* simple network management protocol
soft handoffs, 212
softer handoffs, 212
software
  acquisition, 554
  configuration management, *See* configuration management system
  efficiency, 529
  engineering institute (SEI), 547
  image, 165
  inspection support, 78
  maintenance, 440
  module, *See* module
  process, 546
  product engineering, 552
  project planning, 551
  project tracking, 551
  quality, 5–7, 519, 530
  quality assurance (SQA), 551
  quality assurance (SQA) group, 499
  quality management, 552
  reliability, 10, 477–480
  subcontract management, 551
  system independence, 529
  hardware integration, 174
source, 429
special training, 512
specification dependent, 42
specification adequate , 47
SQA, *See* software quality assurance
SSL, 204
stability, 533
stability tests, 213
stages, 568
stakeholders, 10, 323, 409
standardization, 514
standby mode, 207
start-up time, 456
state coverage, 276, 308
state verification, 278–279
stateless systems, 11, 265
statement coverage, 97–98
state-oriented systems, 11, 265
state transition, 278
state transition sequence, 273
static analysis, 7
static code analyzer, 78
static test technique, 559
static unit testing, 53
statistical oracle, 253
statistical quality control (SQC), 2, 577
statistical testing, 493
Stinger project , 425–426
stop testing, 43
stories, 466–467
story, 71
stress tests, 193, 211–213
stress the system, 456
strictly includes, 124
strongly connected, 280
structural testing, 20–21
structure, 554
stubs, 62
subdomain, 39, 136, 244
Submit State, 326
subroutine arguments, 229
substructure, 229
subsystem, 158
suitability, 531
supplier, 464
sustaining
  phase, 439–440
  test engineers, 440
  test group, 498

Swanson, E. B., 440
Symstra, 76
Synergetic, 508
system
  definition, 158
  approach, 536
  evolves, 325
  integration group, 162
  integration test plan, 180–184
  integration testing (SIT), 158, 164–173
  resolution tests, 197
  test automation, 391–392
  test execution, 408
  test group, 26, 497
  test plan, 356
  test team, 500
  testing (ST), 17
  testing entry criteria, 368
  under test (SUT), 269, 274
systematic error, 433
systematized, 508
systemic requirements , 537
system-level
  fault injection testing, 185
  testing, *See* system testing

tactful, 509
tactically, 73
tactics, 509
Taguchi, Genichi, 236
Tailoring, 185
Target, 429
taxonomy of system tests, 192–193
TCDY, *See* test case design yield
TCE, *See* test case escaped
TDD, *See* test driven development
team building, 513–515
team player, 508
technical
  beta, 435
  leaders , 501
  training program, 573
technology change management, 552
tenacious, 509
terminal node, 281
terminal state block, 285
test (sub)group, 335
test adequacy, *See* adequacy of testing
  criteria for GUI, 219
test approach, 357
test architectures, 291–295
test automation,
  benefits, 24–26
  design characteristics, 397
  infrastructure, 400–401
  practices, 401
  structure, 399
  tools, 392–395, 405
test beds allocation, 416
test case, 22, 300–302
test case creation effort, 384–385
test case dependency, 274
test case design yield (TCDY), 350
test case escapes, 420
test case escaped (TCE), 23, 350, 352
test case execution effort, 385
test case library, 401
test case life cycle, *See* life cycle of a test
  case
test case scheduler, 274
test case schema, 346
test case verdicts, 299–300
test case validation, 258
test case verification, 352
test cases allocation, 416
test coordinate procedures, 292–293
test coordination, 15
test coverage analyzer, 78
test cycle, 362
test data, 36, 230
test data generator, 78–79
test design, 22
test design factors, 321
test design process, 345–347
test development criteria, 393
test driven development (TDD), 22, 71
test driver, 62
test effort estimation, 377
test environment, 358–361, 514–515, 560
test execution, 347, 364
test execution criteria, 394
test execution rate, 417
test execution strategy, 361
test factory, 346
test groups, 496
test harness, 79
test input, 101–106, 244–245
test levels, 16–18
test maintenance criteria, 394
test management criteria, 394

test managers, 500–501
test maturity model (TMM), 6, 568–577
test measurement program, 575
test method, 42
test milestones , 387
test objective, 14, 22, 345
test objective identification, 334–335
test oracle, *See* oracle
test plan, 21
test plan for system integration, *See* system integration test plan
test planning process, 570
test platform, 400
test predicate, 36
test process improvement (TPI), 6, 555–567
test process management, 562
test process optimization, 577
test purpose, *See* test objective
test report, 16, 438–439
test result schema, 348
test result criteria, 394
test selection criterion, 146–154
test selection procedure, 369
test space, 486
test specification technique, 559
test steps, 22, 346, 397
test strategy, 558
test suite
  definition, 335
  FrAtm, 342
  schema , 348
  structure, 358
test system, 274
test team organization, 26
test tools, 24–26, 560
test vector, *See* test data
test verdict, 15
testability, 525, 533
testable requirements, 331
testbed, *See* test environment
tester-developer ratio, 497
test first, 71
testing activities, 14–16
testing and test control notation, *See* TTCN-3
testing effort, 574
testing system, 40
testing theory, 31
testware management, 562
Theory of
  Goodenough and Gerhart, 32–37
  Gourlay, 39–42
  Weyuker and Ostrand, 37–39
thermal test, 176
thorough, 508
thorough test, 33
throughput, 209
tidy, 508
tilted-boundary error, 143
time behavior, 533
timeliness, 458
TLS, 204
TMM, *See* test maturity model
TMP, 294
TMPDU, 294
tool evaluation criteria, 393
tools, 401
top-down, 167–170
total quality control (TQC), 3
total quality management (TQM), 236
TPI, *See* test process improvement
Traceability, 529
traffic generators, 80
training, 512, 529
training program, 551
transcendental view, 5, 452, 519–520
transfer sequence, 278
transition coverage, 276, 308
transition tour, 273–276
Trap, 201
Trigger, 429
trivially valid, 38
troubleshooting guide, 70, 216, 437
trustworthy, 508
TSL, 256, 258
TTCN-3, 256, 295–302
TTLS, 204
types of interfaces, 159

UAT, *See* user acceptance testing
UIO sequence, 279–284
UIO tree, 280–281
UML, *See* unified modeling language
Undefine (u), 114–115
Undefined (U), 114–115
Understandability, 521, 533
unified modeling language (UML), 481
Union of Japanese Scientists and Engineers (JUSE), 2

unique input/output (UIO) sequence, *See* UIO sequence
unit testing, 16, 51–53
Untested, 348–349
Updated state, 346–347
Upgradability, 459–460
upgrade/downgrade tests, 195
upper tester, 292–293
ur (type2) anomaly, 114–115
usability, 455, 521, 525, 531
usability testing, 202
usage profile, *See* operational profile
use space, 486
user, 450
user acceptance testing (UAT), 17, 450
user manuals, 215
user view, 6, 452, 520

valid criterion, 33, 38
valid input, 13, 244
validation, 7–8
value-based view, 6, 452, 520–521
VCCI mark, 217
vector perturbation, 280
vendor qualification criteria, 395
verbal communication, 509
verification, 7–8
verification sequence, 278
Verification State, 329
verifying implementation, 553
version , 165
version control, 80–81
Vinter, D., 322
Virus, 459
Visionary, 509
V-model, 16

Wait, 410
waiting time, 209
walkthrough, 54
Waterfall model, 18, 481
web-based application, 461
white bin, 370
white-box testing, 20–21, 163
wireless data network, 207–208
w-method, 287
worm, 459
wrapper, 185
written communication, 508
w-set, 287–288

XP, *See* eXtreme Programming
xUnit, 73

yellow bin, 369
Yourdon, Edward, 54

Z, 18, 256, 258
zero-day attack, 203
zero-defect, 377