

SOFTWARE TESTING

Principles, Techniques and Tools

M G LIMAYE

```
function deleteRegistration()
{
    $oResult = $oRegisterAssistance->DeleteRegistration();
    if($oResult)
    {
        $oStructuretrans->DeleteRegistration();
    }
}

function startToolAssistance()
{
    $oUserPercentage->Set();
    $oDataAssistance->Set();
    $oPercentage->Set();
    return $oDataAssistance;
}

function registration()
{
    $oDataAssistance->Set();
    $oDataAssistance->Set();
    $oDataAssistance->Set();
}
```

Software Testing

Principles, Techniques and Tools

M G Limaye

Ex-Certified Software Quality Analyst
IRCA Registered Auditor



Tata McGraw-Hill Publishing Company Limited
NEW DELHI

McGraw-Hill Offices

New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto



Tata McGraw-Hill

Published by the Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nager, New Delhi 110 008

Software Testing : Principles, Techniques and Tools

Copyright © 2009 by, Tata McGraw-Hill Publishing Company Limited

No part of this publication may be reproduced or distributed in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise or stored in a database or retrieval system without the prior written permission of the publishers. The program listings (if any) may be entered, stored and executed in a computer system, but they may not be reproduced for publication.

This edition can be exported from India only by the publishers,
Tata McGraw-Hill Publishing Company Limited

ISBN (13): 978-0-07-013990-9

ISBN (10): 0-07-013990-3

Managing Director : *Ajay Shukla*

General Manager : Publishing—SEM & Tech Ed : *Vibha Mahajan*

Sponsoring Editor : *Shalini Jha*

Jr Sponsoring Editor : *Nilanjan Chakravarty*

Jr Editorial Executive : *Tina Jajoriya*

Sr Copy Editor : *Dipika Dey*

Production Executive : *Suneeta S Bohra*

General Manager : Marketing—Higher Education & School : *Michael J Cruz*

Product Manager : SEM & Tech Ed: *Biju Ganesan*

Controller—Production : *Rajender P Ghansela*

Asst. General Manager—Production : *B L Dogra*

Information contained in this work has been obtained by Tata McGraw-Hill, from sources believed to be reliable. However, neither Tata McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that Tata McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

Typeset at Digital Domain IT Services Pvt. Ltd, Kolkata, India and printed at Lalit Offset Printers, 219, F.I.E., Patparganj, Industrial Area, Delhi-110092

Cover Printer: SDR Printers

RAXYCRZFDZAZR

The McGraw-Hill Companies

This book is written as per the syllabus of major universities as well as competitive examinations conducted by international bodies. Independent of any specific technology, application or software vendor references, this book follows a general approach to the software testing process providing a comprehensive synthesis of fundamental concepts, tools, and quality assurance. The terminologies used are in sync with the current industry provision.

Salient Features An integrated approach to test management, techniques and process requirement activities is the hallmark feature of this book. It showcases a practical way of software testing using testing tools, test case designing and testing process. Testing domains with managerial, technical and process orientation include new technologies such as eBusiness and mobile. The book gives due coverage to risk analysis, minimisation, regression testing, data flow testing and review programs. In addition to the breadth of testing topics covered, the book demonstrates the importance of testing and the need for testers to be alert, capable, skilled, and creative.

This book covers

- Quality concepts which is the foundation for software testing
- Concept of product risks and risk based testing
- Details about test planning, test case writing, and defect management
- Various types of system including different testing methodologies
- Detailed analysis of acceptance testing
- Details of quantitative analysis and metrics
- Details of reviews, walkthroughs, and inspection
- Test process improvement models

Rich pedagogy includes

- More than 100 Solved Examples
- More than 200 Chapter-end Exercises
- More than 50 Frequently Asked Questions (FAQs) in Job Interviews
- 6 Real Life Test plans

Note: Flowcharts and templates used are only for illustration purpose and may vary from place to place, organisation to organisation, and customer to customer.

Organisation The structuring of the book offers a practical discussion on topics related to software testing management and process that are critical for delivering high-quality software. It covers the techniques and tools used, processes followed, test management, risk analysis, quality assurance, and related costs integral to software testing.

This book is divided into 5 sections and 17 chapters along with a glossary, list of abbreviations, references, frequently asked questions in job interviews, and appendix covering case studies.

Part I Quality Assurance This section covers the basic software quality concepts.

- **Chapter 1 Introduction to Quality** This chapter describes basic definitions of quality and basic premises of quality management with different views of quality of product from different stakeholders' perspectives. It also defines various quality approaches.
- **Chapter 2 Software Quality** This chapter provides a basic understanding that quality expectation for different products is different, and lays a foundation of quality management approach.

Part II Basic Concepts of Software Testing This section clarifies the basic aspects one must know while studying software testing as a discipline.

- **Chapter 3 Fundamentals of Software Testing** This chapter differentiates between ‘TQM testing’ and ‘Big Bang’ approaches. It also defines different methodologies used in testing such ‘Black Box Testing’, ‘White Box Testing’ and ‘Gray Box Testing’. The last part deals with test processes including process of defining test policy, test strategy, and test plan.
- **Chapter 4 Configuration Management** This chapter gives an overview of configuration management process followed by work products during Software Development Life Cycle (SDLC).
- **Chapter 5 Risk Analysis** This chapter deals with product risk management. It describes possible risks when manual operations are replaced by automated systems. It also describes the risk management process and control mechanisms applied to reduce risk.
- **Chapter 6 Software Verification and Validation** This chapter explains verification and validation as two parts of software testing that complement each other. It describes various activities done under verification and validation during different stages of SDLC.
- **Chapter 7 V-Test Model** This chapter establishes ‘V model’ or ‘validation’ model and ‘VV model’ or ‘verification and validation’ model. It also defines roles and responsibilities of three critical entities in software development.
- **Chapter 8 Defect Management** This chapter defines the defect management process and how it can be used for process improvements. It elucidates the entire defect life cycle.

Part III Testing Techniques and Tools This section covers various levels of testing during SDLC and special testing on and above functionality. It also covers the tools which can be used in testing life cycle.

- **Chapter 9 Levels of Testing** This chapter details various verification and validation activities associated with various stages of SDLC starting from proposal. It also clarifies various ways of integration testing.
- **Chapter 10 Acceptance Testing** This chapter offers an exposition of the acceptance testing process. It starts with acceptance test plan and acceptance test criteria applicable for different systems. It then gives an overview of acceptance test process and various methods of acceptance testing such as alpha, beta and gamma testing.
- **Chapter 11 Special Testing (Part I)** This chapter focuses on testing of specialised systems such as UI testing, compatibility testing, internationalisation testing, security testing, performance testing, recovery testing, and installation testing.
- **Chapter 12 Special Testing (Part II)** This chapter covers special testing such as OO testing, mobile testing, eCommerce and eBusiness testing, control testing, COTS testing, client server testing, web application testing, and agile testing.
- **Chapter 13 Testing Tools** This chapter discusses the process of procuring tools from outside, viz. COTS or vendor developed tools. It also covers guidelines for procurement of tools and advantages/disadvantages of tool usage in SDLC.

Part IV Testing Process This section covers the actual process of testing—starting from test policies, test strategies, test plans, test cases till test reports and quantitative analysis.

- **Chapter 14 Test Planning** The foundation of test related artifacts created and used in testing at various levels are presented in this chapter. This may be termed as test harness, test suite, etc. This chapter covers the following in detail.
 - ✓ Test policy
 - ✓ Test strategy

- ✓ Test planning
- ✓ Test scenario definition
- ✓ Test case definition
- ✓ Test data definition

It deals with test estimation techniques, and the usage of different standards at organisation level, national level, customer level, and international level. It also describes the roles and responsibilities of different people in test teams.

- **Chapter 15 Test Metrics and Test Reports** This chapter offers various reports prepared during different levels of verification and validation activity. It also discusses benchmarking and measurement process deployment in an organisation.
- **Chapter 16 Qualitative and Quantitative Analysis** This chapter covers quantitative and qualitative data collection and uses of qualitative and quantitative tools required for data analysis and decision making during and after testing.

Part V Test Process Management

- **Chapter 17 Test Process Improvement** This chapter explains the need for test process improvement along with a detailed exposition of the test process improvement models.

Online Learning Centre The accompanying web supplement of the book at <http://www.mhhe.com/limatest> is periodically updated and includes resources for instructors and students. Instructors can avail solution manual, chapter-wise PowerPoint slides with diagrams, and notes to supplement lecture presentations, and additional exercises. Students can access a host of resources including a sample chapter, chapter wise-exercises, links to reference material, additional test cases, and an exhaustive test bank.

Acknowledgements A note of acknowledgement is due to the reviewers of this book for their in-depth comments and thoughtful criticisms.

Bindu Goel

*Guru Gobind Singh Indraprastha University,
New Delhi*

Rizwan Ahmad

*Anjuman College of Engineering & Technology,
Nagpur*

Vinita Gaikwad

Thakur College of Science & Commerce, Mumbai

N K Srinath

R V College of Engineering, Bangalore

Abdaal Khaleeqe

Tata Consultancy Services

I would be obliged if readers send their valuable suggestions for further improvement of the book. You may contact me at the following email id—tmh.csefeedback@gmail.com. Kindly mention the title and author name in the subject line.

Milind Limaye

CONTENTS

Preface
Walkthrough

v
xvii

Part I Quality Assurance

1. Introduction to Quality	3–29
1.1 Introduction	3
1.2 Historical Perspective of Quality	4
1.3 What is Quality? (Is it a fact or perception?)	4
1.4 Definitions of Quality	6
1.5 Core Components of Quality	7
1.6 Quality View	10
1.7 Financial Aspect of Quality	15
1.8 Definition of Quality	17
1.9 Customers, Suppliers and Processes	18
1.10 Total Quality Management (TQM)	18
1.11 Quality Principles of ‘Total Quality Management’	19
1.12 Quality Management Through Statistical Process Control	22
1.13 Quality Management Through Cultural Changes	23
1.14 Continual (Continuous) Improvement Cycle	24
1.15 Quality in Different Areas	25
1.16 Benchmarking and Metrics	26
1.17 Problem Solving Techniques	26
1.18 Problem Solving Software Tools	27
2. Software Quality	30–59
2.1 Introduction	30
2.2 Constraints of Software Product Quality Assessment	30
2.3 Customer is a King	31

2.4	Quality and Productivity Relationship	32
2.5	Requirements of a Product	34
2.6	Organisation Culture	36
2.7	Characteristics of Software	38
2.8	Software Development Process	38
2.9	Types of Products	43
2.10	Some Other Schemes of Criticality Definitions	44
2.11	Problematic Areas of Software Development Life Cycle	45
2.12	Software Quality Management	50
2.13	Why Software Has Defects?	50
2.14	Processes Related to Software Quality	51
2.15	Quality Management System Structure	52
2.16	Pillars of Quality Management System	53
2.17	Important Aspects of Quality Management	54

Part II Basic Concepts of Software Testing

3. Fundamentals of Software Testing	63–115	
3.1	Introduction	63
3.2	Historical Perspective of Testing	64
3.3	Definition of Testing	65
3.4	Approaches to Testing	66
3.5	Popular Definitions of Testing	70
3.6	Testing During Development Life Cycle	73
3.7	Requirement Traceability Matrix	75
3.8	Essentials of Software Testing	77
3.9	Workbench	78
3.10	Important Features of Testing Process	80
3.11	Misconceptions About Testing	82
3.12	Principles of Software Testing	83
3.13	Salient Features of Good Testing	84
3.14	Test Policy	85
3.15	Test Strategy or Test Approach	85
3.16	Test Planning	85
3.17	Testing Process and Number of Defects Found in Testing	87
3.18	Test Team Efficiency	87
3.19	Mutation Testing	88
3.20	Challenges in Testing	89
3.21	Test Team Approach	90
3.22	Process Problems Faced by Testing	94
3.23	Cost Aspect of Testing	95
3.24	Establishing Testing Policy	99
3.25	Methods	99
3.26	Structured Approach to Testing	100
3.27	Categories of Defect	101
3.28	Defect, Error, or Mistake in Software	101

3.29	Developing Test Strategy	102
3.30	Developing Testing Methodologies (Test Plan)	102
3.31	Testing Process	105
3.32	Attitude Towards Testing (Common People Issues)	107
3.33	Test Methodologies/Approaches	107
3.34	People Challenges in Software Testing	111
3.35	Raising Management Awareness for Testing	111
3.36	Skills Required by Tester	112
4.	Configuration Management	116–131
4.1	Introduction	116
4.2	Configuration Management	116
4.3	Cycle of Configuration Management	117
4.4	Configuration Management Process	119
4.5	Auditing Configuration Library	122
4.6	Configurable Item	123
4.7	Baselining	123
4.8	Few More Concepts About Configuration Library	124
4.9	Storage of Configurable Items in Library	124
4.10	Using Automated Configuration Tools	126
4.11	Configuration Management Planning	127
5.	Risk Analysis	132–160
5.1	Introduction	132
5.2	Advantages of Automated System	132
5.3	Disadvantages of Automated System	133
5.4	Risk	133
5.5	Constraints (Caveats)	134
5.6	Project Risks (Project Management Risks)	134
5.7	Product Risks	136
5.8	Risks Faced Due to Software Systems	137
5.9	Software Implementation Risks	147
5.10	Identification of Risks	148
5.11	Types of Software Risks	150
5.12	Handling of Risks in Testing	151
5.13	Types of Actions for Risk Control Management	152
5.14	Risks and Testing	154
5.15	Assumptions in Testing	155
5.16	Testing as a Risk Reduction Program (Prioritisation in Testing)	156
5.17	Risks of Testing	156
6.	Software Verification and Validation	161–190
6.1	Introduction	161
6.2	Verification	162
6.3	Verification Work Bench	163
6.4	Methods of Verification	164
6.5	Types of Review on the Basis of Stage/Phase	171

6.6 Examples of Entities Involved in Verification	174
6.7 Reviews in Testing Lifecycle	175
6.8 Coverage in Verification (Test Designing)	177
6.9 Concerns of Verification	179
6.10 Validation	180
6.11 Validation Work Bench	181
6.12 Levels of Validation	182
6.13 Coverage in Validation (Prioritisation/Slice Based Testing)	183
6.14 Acceptance Testing	185
6.15 Management of Verification and Validation (V & V)	186
6.16 Software Development Verification and Validation Activities	188
7. V-Test Model	191–202
7.1 Introduction	191
7.2 V Model For Software	191
7.3 Testing During Proposal Stage	194
7.4 Testing During Requirement Stage	194
7.5 Testing During Test-Planning Phase	195
7.6 Testing During Design Phase	196
7.7 Testing During Coding	197
7.8 VV Model	198
7.9 Critical Roles and Responsibilities	200
8. Defect Management	203–218
8.1 Introduction	203
8.2 Defect Classification	204
8.3 Defect Management Process (Approach)	206
8.4 Defect Life Cycle	207
8.5 Defect Template	209
8.6 Defect Management Process (Fixing and Root Cause of Defect)	212
8.7 Estimate Expected Impact of a Defect	214
8.8 Why Defect Management Needs a Risk Discussion?	215
8.9 Techniques for Finding Defects	216
8.10 Reporting a Defect	216

Part III Testing Techniques and Tools

9. Levels of Testing	221–234
9.1 Introduction	221
9.2 Proposal Testing	221
9.3 Requirement Testing	222
9.4 Design Testing	224
9.5 Code Review	225
9.6 Unit Testing	225
9.7 Module Testing	226
9.8 Integration Testing	226
9.9 Big-Bang Testing	230
9.10 Sandwich Testing	231

9.11	Critical Path First	232
9.12	Subsystem Testing	232
9.13	System Testing	232
9.14	Testing Stages	233
10.	Acceptance Testing	235–257
10.1	Introduction	235
10.2	Acceptance Testing Criteria	236
10.3	Importance of Acceptance Criteria	237
10.4	Some Famous Acceptance Criteria	237
10.5	Alpha Testing	241
10.6	Beta Testing	242
10.7	Gamma Testing	243
10.8	Acceptance Testing During Each Phase of Software Development	244
10.9	Consideration of Alpha and Beta Acceptance Testing Process	245
10.10	What Does Software Acceptance Enable?	246
10.11	Customer's Responsibilities in Acceptance Testing	246
10.12	Fits for Acceptance Testing	246
10.13	Define Acceptance Criteria	247
10.14	Criticality of Requirements	248
10.15	Factors Affecting Criticality of the Requirements	249
10.16	Developing Acceptance Test Plan	251
10.17	Software Acceptance Plan	251
10.18	User Responsibilities in Acceptance Test plan	253
10.19	Executing Acceptance Plan	255
11.	Special Tests (Part I)	258–292
11.1	Introduction	258
11.2	Complexity Testing	259
11.3	Graphical User Interface Testing	260
11.4	Compatibility Testing	262
11.5	Security Testing	268
11.6	Performance Testing, Volume Testing and Stress Testing	270
11.7	Recovery Testing	272
11.8	Installation Testing	274
11.9	Requirement Testing (Specification Testing)	275
11.10	Regression Testing	276
11.11	Error Handling Testing	277
11.12	Manual Support Testing	278
11.13	Intersystem Testing	278
11.14	Control Testing	279
11.15	Smoke Testing	279
11.16	Sanity Testing	280
11.17	Adhoc Testing (Monkey Testing, Exploratory Testing, Random Testing)	280
11.18	Parallel Testing	280
11.19	Execution Testing	281
11.20	Operations Testing	281

11.21	Compliance Testing	282	
11.22	Usability Testing	282	
11.23	Decision Table Testing (Axiom Testing)	283	
11.24	Documentation Testing	284	
11.25	Training Testing	285	
11.26	Rapid Testing	285	
11.27	Control Flow Graph	286	
11.28	Generating Tests on the Basis of Combinatorial Designs	288	
11.29	State Graph	289	
12. Special Tests (Part II)			293–316
12.1	Introduction	293	
12.2	Risk Associated with New Technologies	294	
12.3	Process Maturity Level of Technology	296	
12.4	Testing Adequacy of Control in New Technology Usage	297	
12.5	Object-Oriented Application Testing	298	
12.6	Testing of Internal Controls	299	
12.7	'COTS' Testing	302	
12.8	Client-Server Testing	306	
12.9	Web Application Testing	307	
12.10	Mobile Application Testing (PDA Devices)	309	
12.11	eBusiness/eCommerce Testing	310	
12.12	Agile Development Testing	312	
12.13	Data Warehousing Testing	313	
13. Testing Tools			317–330
13.1	Introduction	317	
13.2	Features of Test Tool	318	
13.3	Guidelines for Selecting a Tool	318	
13.4	Tools and Skills of Tester	319	
13.5	Static Testing Tools	319	
13.6	Dynamic Testing Tools	319	
13.7	Advantages of Using Tools	320	
13.8	Disadvantages of Using Tools	320	
13.9	When to Use Automated Test Tools	320	
13.10	Testing Using Automated Tools	321	
13.11	Difficulties While Introducing New Tools	323	
13.12	Process of Procurement of COTS (Readily Available Tool from Market)	324	
13.13	Procurement of Tools from Contractor	326	
13.14	Advantages of Tools Developed By External Organisations	327	
13.15	Contracting a Software	327	
13.16	Process of Procurement of Tools from Contractor	328	

Part IV Testing Process

14. Test Planning		333–383	
14.1	Introduction	333	
14.2	Test Policy	334	

14.3	Content of Test Policy in General	335
14.4	Test Strategy	336
14.5	Content of Test Strategy in General	336
14.6	Test Planning	338
14.7	Test Plan	338
14.8	Quality Plan and Test Plan	340
14.9	Quality Plan Template	341
14.10	Test Plan Template (System Testing)	345
14.11	Guidelines for Developing the Test Plan	353
14.12	Test Administration Definition	354
14.13	Test Estimation	354
14.14	Test Standards	356
14.15	Building Test Data and Test Cases	357
14.16	Test Scenario	358
14.17	Test Cases	361
14.18	Essential Activities in Testing	363
14.19	Template for Test Cases	363
14.20	Test Scripts	366
14.21	Test Management Software	368
14.22	Test Log Document	369
14.23	Effective Test Cases	369
14.24	Test File	370
14.25	Building Test Data	371
14.26	Generation of Test Data	378
14.27	Tools Used to Build Test Data	380
14.28	Roles and Responsibilities in Testing Life Cycle	380
14.29	Test Progress Monitoring	382
15. Test Metrics and Test Reports		384–424
15.1	Introduction	384
15.2	Testing Related Data	385
15.3	Defect Data	386
15.4	Efficiency/Productivity Data	388
15.5	Categories of the Product/Project Test Metrics	391
15.6	Estimated, Budgeted, Approved and Actual	392
15.7	Resources Consumed in Testing	393
15.8	Effectiveness of Testing	394
15.9	Defect Density	395
15.10	Defect Leakage Ratio (Defect Life)	396
15.11	Residual Defect Density (RDD)	397
15.12	Test Team Efficiency	397
15.13	Test Case Efficiency	397
15.14	Rework	398
15.15	MTBF/MTTR	398
15.16	Implementing Measurement Reporting System in an Organisation	399
15.17	Test Reports	402
15.18	Project Test Status Report	403

15.19 Test Reports	406
15.20 Integration Test Report	407
15.21 System Test Report	407
15.22 Acceptance Test Report	408
15.23 Guidelines for Writing and Using Report	408
15.24 Final Test Reporting	408
15.25 Test Status Report	409
15.26 Benchmarking	419
16. Qualitative and Quantitative Analysis	425–451
16.1 Introduction	425
16.2 Types of Data	425
16.3 Qualitative and Quantitative Data	426
16.4 Types of Qualitative Data	427
16.5 Types of Quantitative Data	428
16.6 Quality Tools	431
16.7 Non-Statistical Tools	431
16.8 Statistical Tool Background	440
16.9 Statistical Tools	441
Part V Test Process Management	
17. Test Process Improvement	455–462
17.1 Introduction	455
17.2 Change in Perception About Testing	455
17.3 Problems Concerning Testing Process	456
17.4 The Need for Test Process Improvement	457
17.5 Test Process Maturity	457
17.6 Test Process Improvement Model	458
17.7 Test Process Improvement Model Stages	460
17.8 Graphical Representation of Improvements	461
<i>Appendix</i>	463
<i>Frequently Asked Questions</i>	489
<i>Glossary</i>	503
<i>List of Abbreviations</i>	514
<i>References</i>	517
<i>Index</i>	518

VISUAL



WALKTHROUGH

3

FUNDAMENTALS OF SOFTWARE TESTING

Objectives

A quick overview of the concepts that will be discussed in the chapter

Introduction

Basic background provides an advance organiser of the information to come

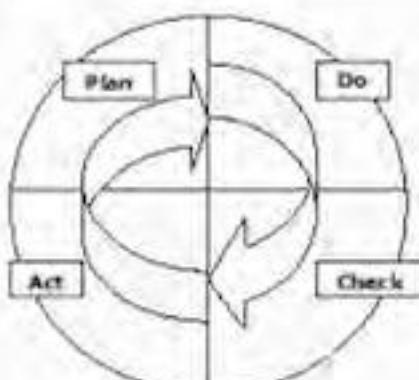


FIG 1.4

Continuous improvement cycle (PDCA cycle)

Table 1.1 Defect phases and corresponding status

Defect phases	Typical status
New defect found:	Open
Defect assigned to concerned person (generally module lead, project lead, team lead etc.)	Assigned name of person
Defect is duplicate	Duplicate
Defect is as per requirements	Not a defect
Defect cannot be reproduced	Cannot be reproduced
Defect assigned to person who will be fixing it	Assigned name of person
Defect fixed by the identified person	Fixed
Defect verified by module lead, team lead, project lead etc.	Verified
Defect being assigned to tester	Assigned name of person
Defect found to be fixed	Closed
Defect found to be open	Reopen

Figures (65) and Tables (53)

Excellent study tools,
Figures and Tables visually enhance
comprehension and retention
of the text material

8.2 DEFECT CLASSIFICATION

Defects may be defined as anything which makes the customer unhappy. It may be a result of wrong implementation of requirements, something missing that is required by the customer, or putting something more (extra) than required. Defects may be classified in different ways under different schemes. One of the schemes may be as follows.

8.2.1 REQUIREMENT DEFECTS

Requirement-related defects arise in a product where one fails to understand what is required by the customer. These defects may be due to customer guy, where the customer is unable to define his requirements, or producer guy, where developing team is not able to make a product as per requirements.

Requirement-related defects may be further classified as given below.

Functional Defects: These defects are mostly about the functionalities present/absent in the application which are expected/required by the customer. Generally, system testing concentrates on functionality as the first part of testing. Non-working functions and functions not provided as required may represent functional defects.

Interface Defects: These defects talk about various interfaces essential as per the requirement statement. Generally, these defects may be due to user interface problems, problems with compatibility to other systems including hardware, etc.

8.2.2 DESIGN DEFECTS

Design defects generally reflect the way of design creation or its integration while creating a product. The customer may or may not be in a position to understand these defects, if structures are not correct. They may be due to problems with design creation and implementation during software development life cycle.

Design related defects may be classified as follows.

Algorithmic Defects: Algorithms in defect may be introduced in a product if designs of various decision are not handled correctly. Some applications have a control over complexity and usage of various algorithms to correlate requirements correctly into coding. Various presentations and correlations may cause defects introduction in the product.

2.9 TYPES OF PRODUCTS

Similar to software development methodologies, software products have some peculiarities defined in criticalities of software. Criticality of the software defines how much important it is to a user/developer.

There are various schemes of grouping the products on the basis of criticality to the user. Few of them are listed below.

2.9.1 LIFE AFFECTING PRODUCTS

Products, which directly/indirectly affect human life are considered as the most critical products in the world from user's perspective. Such products generally come under the purview of regulatory and safety requirements, as addressed to normal customer requirements. The quality requirements are more stringent, and testing is very critical for such products as failures may result into loss of life or dismemberment of a user. This type of product may be further grouped into 3 different categories.

- Any product failure resulting into death of a person. These will be the most critical products as they can affect human life directly.
- Any product failure which may cause permanent dismemberment to a patient. These are second-level criticality products.
- Any product failure which may cause minor injury and does not result into anything as defined above.
- Other products which do not affect health or safety directly.

Such software needs huge testing to try each and every conceivable fault in the product. It talks about very high level of confidence that application will not fail under normal and abnormal situations.

2.9.2 PRODUCT AFFECTING HUGE SUM OF MONEY

A product which has direct relationship with loss of large sum of money is second in the list of criticality of the product. Such products may need large testing efforts and have many regulatory as well as security requirements. E-commerce and e-business softwares may be put in this category. Security, confidentiality, and accuracy are some of the important quality factors for such products.

These products also need very high confidence level and hence testing will represent the criticality. They need testing lesser than the products which directly/indirectly affect human life.

Sections and Sub-sections

Neatly divided Sections and Sub-sections
within each chapter enable students
to grasp the subject matter in a
logical progression of ideas and concepts

Risk Analysis 159

An organization implements a cheque printing software. There is a probability that 1% of the cheques printed by this software are wrong. The cost associated with each wrong cheque is Rs 500. The organization prints 10,000 cheques in a year.

- Calculate the risk of implementation of this software to user.
- If control is applied to this software, cost of control is Rs 10,000. It can reduce the probability of wrong cheque printing to 0.1%. Suggest whether you will recommend the controls or not.

Answer:

a. Probability of printing a wrong cheque
 Total number of cheques printed in a year
 Total wrong cheques printed in a year
 Cost associated with each wrong cheque printed
 Total risk due to this software

- 1%	= 10,000	= 1% of 10,000	= 100
		- Rs. 500	= Rs. 50,000

b. Probability of printing wrong cheque after implementing controls
 Total number of cheques printed in a year
 Total wrong cheques printed in a year
 Cost associated with each wrong cheque printed
 Total risk due to this software

- 0.1%	= 10,000	= 0.1% of 10,000	= 10
		- Rs. 500	= Rs. 500 x 10
			= Rs. 5,000

Saving due to control
 Cost of control

= 50,000 - 5,000	= 45,000
- 10,000	

Since cost of control is less than the saving achieved, we will recommend this control.

If software is implementing a date limitation, suggest the controls which can be used to ensure that the user enters the date correctly. Expected behavior of an application is that the user will enter month in 2 digits in 1st text box, date in 2 digits in 2nd text box and year in 4 digits in 3rd box.

Indicative/Selective control:
 When a user enters a date, the controls detect whether it has been entered correctly or not. Suppose, a user enters 31 in 1st text box, 12 in second text box and 2009 in 3rd text box, it gives an error message that date must be entered in YYYY-MM-DD format. Control is of no use, if date returned is 01 in 1st box, 01 in 2nd box and 2009 in 3rd box as application cannot understand whether user wants to enter 7th Jan or 1st Jul and both can be possible dates.

Suggestive control:
 One option can be that there will be the letters "MM" in 1st text box indicating that month is to be entered in this box in 2 digits, letters "DD" in 2nd box indicating that date is to be entered in this box in 2 digits, and letter "YYYY" in third box indicating that year is to be entered in this text box in 4 digits.

Auto - corrective control:
 Control will be similar to suggestive control but if a user enters 31 in 1st text box, it automatically takes 01/01/2009. If user enters "1" in 2nd text box, it automatically takes 01 as 01 and if 107 is entered in 3rd text box, it automatically converts it into 2007.

Preventive control:
 Application will not allow typing in text boxes but provides a calendar control. When user selects a date, it automatically populates the three text boxes on the basis of date selected.

Worked Examples

Solved examples interspersed throughout the book reinforce understanding of key topics

Illustrations (76) >>>

Abundant and relevant examples illustrate applications of software testing concepts

Illustration 9.1

Many requirement statements have words like 'Application must be user friendly' or 'Application performance must be fast'. Such requirements cannot be implemented as nobody knows the exact requirements.

One may have to take queries to understand the meaning of user friendliness or fast performance. If possible such statements must have some numerical figures to illustrate them. Tester will be able to write exact test case on the basis of these numbers.

- Complete:** Requirement statement must be complete and must talk about all business aspects of the application under development. It must consider all permutations and combinations possible when application is used in production. Any assumption must be documented and approved by the customer.

Illustration 9.2

Sometimes requirement statement does not specify the transaction. If requirement statement says 'Enter valid Login and Password', then it is not a complete requirement as it does not specify how one should enter valid Login and how to go to Password. It could be by 'tab' or by clicking the mouse or it will automatically go to Password.

If any part of transaction requires an assumption, it indicates incomplete requirements. One has to go back to customer or business analyst/system analyst to get this clarified.

- Messy/ambiguous:** Requirements must be measurable in numeric terms as far as possible. Such definition helps to understand whether the requirement has been met or not while testing the application. Qualitative terms like 'good', 'sufficient', and 'high' must be avoided as different people may attach different meanings to these words.

Illustration 9.3

Sometimes, requirement statement may states that—enter valid credential and click 'Login' to go to next page. It is not very clear how much time it will take to reach next page.

These can not be verified under usability as performance testing.

14.30 TIPS FOR SUCCESSFUL TESTING

- Define new policy and new strategy. People in an organization must understand the intent of senior management.
- Make a complete quality plan and test plan. All verification and validation activities must be covered under a plan.
- Plan sufficient time for testing. Use organizational baselines for estimating test efforts. In case, they are not available, one may look for heuristic approach or expert's judgment, as the case may be.
- Use organization level, customer level, national level or international level techniques and forums for new plans, test scenarios, and test cases. These give uniformity.
- Define test data by using some technique such as equivalence partitioning, state transition, boundary value analysis and error guessing.
- Define and understand roles and responsibilities of test teams and stakeholders for making testing a success.

Tips

Pragmatic guidance helps you make right decisions or avoid common problems while testing software

Summary

Provides a snapshot of important chapter concepts

Summary

This chapter covers qualitative and quantitative analysis which can be used to make continuous/ continual improvements. Quantitative data may be captured using 100% sampling or various other sampling methods. The following tools have been studied in this chapter:

- Checklist
- Checklists
- Rootcause
- Cause and effect diagram/ Ishikawa diagram/ Fish bone diagram
- Flowcharting
- Force field analysis
- Tree chart
- Relationship diagram
- Affinity diagram
- Pareto chart (80–20 rule, ABC analysis, Ishikawa/ fish bone)
- Pie chart
- Bar chart
- Run chart
- Scatter plot
- Histogram
- Control chart
- Balance sheet/and

- Describe compatibility testing.
- Describe user interface testing.
- Describe different types of compatibility.
- What are the concerns in compatibility testing?
- Describe the challenges faced by tester during internationalization testing.
- Describe security testing.
- Describe machine backup types.
- Describe system disaster recovery types.
- Describe installation, migration and upgrade testing.
- Explain requirement (specification) testing.
- Explain regression testing.
- Explain error handling testing.
- Explain manual support system testing.
- Explain inter-system testing.
- Explain control testing.
- Differentiate between smoke testing and sanity testing.
- What are the advantages and disadvantages of ad hoc testing (exploratory, random or monkey testing)?
- Explain execution testing.
- Explain operations testing.
- Differentiate between usability testing, execution testing and operations testing.
- Explain compliance testing.
- Explain usability testing.
- Explain how decision tables are used to optimize testing efforts.
- Explain the concept of documentation testing.
- Explain training testing.



Review Questions (231)

Spur students to apply and integrate chapter content

CASE STUDY 2

Name:	<input type="text"/>
Surname:	<input type="text"/>
Address:	<input type="text"/>
<input type="button" value="Add"/> <input type="button" value="Delete"/> <input type="button" value="Search"/> <input type="button" value="Quit"/>	

Fig. 6.2 Screen layout for creating employee master

Appendix

*Case studies show detail test planning
and test case writing using practical examples*

TEST PLAN

- **Introduction:** This application is intended to create a database for employees in an organization. All the fields are mandatory. Records can be added, searched and deleted as required.
- **Intended Audience:** This test plan is made for system testing of this application. The test plan will be managed by:
 - Development manager and development team
 - Customer
 - System management
- **Intended Users:** This application will be used by human resource department for adding a new employee, searching an existing employee and deleting an existing employee.

GLOSSARY**THE TERMINOLOGY OF QUALITY**

The terms mentioned here are selected as some of the widely used terms in the quality industry. The definition of terminologies could vary from place to place and from organization to organization. Many times people use the terms without knowing their meaning. I am defining them in my own way which may not be the same as somebody else's. The first step to establish a quality management system in an organization is to develop a vocabulary that can be used by all people involved in development and testing. While communicating with a customer, it is essential to understand the meaning of the terms that the customer is using in the proper perspective.

- **Acceptable Quality Level:** The maximum non-conformance or the number of defects acceptable to a customer may be defined as the acceptable quality level. It is believed that a defect-free product cannot exist and there will be a certain number of defects acceptable to a customer. This may be defined as a part of the acceptance test plan.
- **Acceptance Testing:** This testing is designed and planned by a customer or a customer representative to ensure that the system meets the needs before delivery or during acceptance period. Successful acceptance testing means confirms that the system is ready for production.
- **Adding Value:** Adding something that was not there before and something which the customer sees is known as adding value. Addition of features, functionalities, convenience enhancement with the product, which may help the customer in a better way than expected, may be termed as value addition.
- **Alpha Testing:** Alpha testing is a part of acceptance testing. Alpha testing is conducted in a development environment. It is expected to be done by the customer. But in many places, alpha testing is done by system teams and the customer may just look it over. It is also considered an efficient way to test.

Glossary

Key words (84) have been defined alphabetically in the glossary at the end of the book

REFERENCES

- Guide to the Essential Skills of Knowledge for Certified Software Test Engineers (CSTE)
- Information Systems Examination Board—Practitioner certificate in Software Test Management
- Information Systems Examination Board—Practitioner certificate in Software Test Analysis
- Information Systems Examination Board—Associate Certificate in Software Testing
- Software Test Engineering Foundation
- Software Test Analyst Certified through Delft University of Technology
- Software Test Analyst Certified through University of Twente, Computing
- Software Test Analyst Certified through Vrije Universiteit Technological University
- Software Test Analyst Certified through WUWT
- Software Test Analyst Certified through NHTV Technological University
- Our of Course—Dr William Edwards Deming
- Ukraine Methods for Software Testing—William Henry
- The Art of Software Testing—Glennford Myers

References

A comprehensive list of references at the end of the book give you a lead on quickly finding more information

Frequently Asked Question (47) >>>

An assembled list of commonly asked questions along with detailed answers (with special focus on job interviews)

FREQUENTLY ASKED QUESTIONS

Q. 1 What is 'Software Quality Assurance'?

Answer: Software Quality Assurance involves developing and implementing the entire software development process—monitoring, maintaining, controlling and improving the process, making sure that any agreed-upon standards, test procedures are followed, and ensuring that problems related to processes are found and dealt with so that they are not repeated. It is oriented to 'prevention' rather than 'reaction'. At many places, software quality assurance is the term used for software testing. As far as an organization is concerned, continuous understanding is sufficient. But when one is dealing with the outside world, more popular terminology may be used.

Q. 2 What is 'Software Testing'?

Answer: As per Glennford Myers, software testing involves execution of a system or application under controlled conditions and evaluating the results with respect to expected ones with an intention to find defect. The controlled conditions must include normal as well as abnormal conditions occurring during usage of software in real life. Testing must intentionally attempt to make things go wrong, to determine if wrong things happen during particular when they are not, or things do not happen when they must. It is oriented to 'detection'.

Prevention based testing also concentrates on verification along with validation activities. Reviews, inspection, walkthrough, and audits are also considered as the parts of software testing.

Q. 3 What causes a defect?

Answer: Defect can be caused by a flaw in the application software or by a flaw in the application specifications or both. Testing is commonly assessed to mean executing software and finding errors related to specifications and products. Defect mostly originates due to acceptable process used for product development or testing. Generally, defect is a failure observed in review or testing with respect to defined standards or expected result. Defect may be termed as 'failure' and some cause of failure may be termed 'defect' or 'problem'.

Q. 4 Why does software have bugs?

Answer: There can be many reasons for bugs in software. The main reason of software defects may be attributed to implementation flaws. Any defect may be attributed to the supporting processes.

Part I

Quality Assurance

CHAPTER

1

INTRODUCTION TO QUALITY



OBJECTIVES

This chapter provides a basic understanding of quality management. It describes basic definitions of quality and premises of quality management with different views of product quality from stakeholder's perspectives. This foundation is essential for software testers to understand the criticality of their position in software development life cycle.



1.1 INTRODUCTION

Evolution of mankind can be seen as a continuous effort to make things better and convenient through improvements linked with inventions of new products. Since time immemorial, humans have used various products to enhance their lifestyle. Initially, mankind was completely dependent on

nature for satisfying their basic needs of food, shelter and clothing. With evolution, human needs increased from the basic level to substantial addition of derived needs to make life more comfortable. As civilisation progressed, humans started converting natural resources into things which could be used easily to satisfy their basic as well as derived needs.

Earlier, product quality was governed by the individual skill and it differed from instance to instance depending upon the creator and the process used to make it at that instance. Every product was considered as a separate project and every instance of the manufacturing process led to products of different quality attributes. Due to increase in demand for same or similar products which were expected to satisfy same/similar demands and mechanisation of the manufacturing processes, the concept of product specialisation and mass production came into existence. Technological developments made production faster and repetitive in nature.

Now, we are into the era of specialised mass production where producers have specialised domain knowledge and manufacturing skill required for particular product creation, and use this knowledge and skill to produce the product in huge quantities. The market is changing considerably from monopoly to fierce competition but still maintaining the individual product identity in terms of attributes and characteristics. There are large number of buyers as well as sellers in the market, providing and demanding similar products, which satisfy similar demands. Products may or may not be exactly the same but may be similar or satisfying similar needs/demands of the users. They may differ from one another to some extent on the basis of their cost, delivery schedule to acquire it, features, functionalities present/absent, etc. These may be termed as attributes of the quality of a product.

We will be using the word ‘product’ to represent ‘product’ as well as ‘service’, as both are intended to satisfy needs of users/customers. Service may be considered as a virtual product without physical existence but satisfying some needs of users.

1.2 HISTORICAL PERSPECTIVE OF QUALITY

Quality improvement is not a new pursuit for mankind. The field of quality and quality improvement has its roots in agriculture. Early efforts of quality improvement in agriculture may be attributed to statistical research conducted in Britain, in early 20th century, to assist farmers in understanding how to plan the crops and rotate the plan of cultivation to maximise agricultural production while maintaining the soil quality at the same time.

This work inspired Walter Shewhart at Bell Laboratories to develop quality improvement programs through planned efforts. He adopted the concepts developed initially for agriculture to implement quality improvement programs for products and to reduce the cost to customer without affecting profitability for the manufacturer. Changes brought in by Walter Shewhart motivated Dr Edward Deming to implement quality improvement programs as a way to improve product quality. He devoted his life to teaching and improving quality methods and practices across the world through ‘Total Quality Management’ methodology.

Dr Deming demonstrated his ideas of ‘Total Quality Management’ through continual improvement in Japan. Dr Joseph Juran also implemented quality improvement through measurement programs using different quality tools for assessment and improvement. Japanese producers fully embraced quality improvement methodologies and started to integrate the concepts of ‘Total Quality Management’ in their industries. The dramatic improvement in quality of products in Japan after 1950 due to the revolutionary ideas of continual improvement through process measurement are still considered legendary.

During last few decades, the Japanese industry has successfully utilised quality tools and ‘Total Quality Management’ methodologies as part of their successful effort to become a leading nation in manufacturing and supplying a vast array of electronics, automotive and other products to the entire world. Quality of the products is established and continually improved in terms of features, consistent performance, lesser costs, reasonable delivery schedule, etc. in order to enhance the satisfaction of the customer. Japanese products started dictating the quality parameters in world market to the extent that many nations adopted quality improvement programs at national level to face the competition from the Japanese industry. Quality of Japanese products stems from the systematic organisation and understanding of processes used in all aspects of product development, and introduction of tools and methodologies that permit monitoring and understanding about what is happening in different processes of manufacturing and management of interactions of those processes. Japanese quality improvement programs created the sets of interrelated processes which assure the same product quality in repetitive manner and in large number to satisfy the demand of a huge market. Defects are analysed and root causes of the defects are identified and eliminated through continual process improvement. This has helped in optimising the processes to produce better results in repetitive manner.

1.3 WHAT IS QUALITY? (IS IT A FACT OR PERCEPTION?)

What is quality, is an important question but does not have a simple answer. Some people define it as a fact while others define it as a perception of customer/user.

We often talk about quality of a product to shortlist or select the best product among the equals when we wish to acquire one. We may not have a complete idea about the meaning of quality or what we are looking for while selecting a product, if somebody questions us about the reason for choosing one product over the

other. This is a major issue faced by the people working in quality field even today as it is very difficult to decide what contributes customer loyalty or first-time sale and subsequent repeat sale. The term ‘quality’ means different things to different people at different times, different places and for different products. For example, to some users, a quality product may be one, which has no/less defects and works exactly as expected and matches with his/her concept of cost and delivery schedule along with services offered. Such a thought may be a definition of quality—‘**Quality is fitness for use**’.

However, some other definitions of quality are also widely discussed. Quality defined as, ‘**Conformance to specifications**’ is a position that people in the engineering industry often promote because they can do very little to change the design of a product and have to make a product as per the design which will best suite the user’s other expectations like less cost, fast delivery and good service support. Others promote wider views, which may include the attribute of a product which satisfies/exceeds the expectations of the customer. Some believe that quality is a judgment or perception of the customer/user about the attributes of a product, as all the features may not be known or used during the entire life of a product. Quality is the extent to which the customers/users believe that the product meets or surpasses their needs and expectations. Others believe that quality means delivering products that,

- Meet customer standards, either as defined by the customer or defined by the normal usage or by some national or international bodies. (Standard may or may not be as defined by the supplier of the product. Typically for consumer goods, standard is defined by market forces and likes and dislikes of users in general.)
- Meet and fulfill customer needs which include expressed needs as well as implied requirements derived by business analysts and system analysts. Expressed needs are available in the form of requirement statement generated by users while implied needs definition may require supplier to understand customer business and provide the solution accordingly.
- One must try to meet customer expectations as maximum as possible. If something is given more than the requirements of the customer, it should be declared before transition, so that customer surprises can be avoided. At the same time, if some aspect of the specified requirements has not been included in the product, it should be declared. This is essential so that the customer may understand the deliverables accordingly. Expectations may be at the top of customer needs and may be useful in creating brand loyalty through customer delight. (Trying to get customer delight after informing customer about it.)
- Meet anticipated/unanticipated future needs and aspirations of customers by understanding their businesses and future plans. One may need to build a software and system considering some future requirements. Every product including software has a life span and due to technological inventions as well as new ways of doing things, older systems become obsolete, either technically or economically. How much of the future must be considered for the given product may be a responsibility of the customer or the supplier or a joint responsibility. Every product has some defined life span and one may have to extrapolate future needs accordingly.

Others may simply ignore these definitions of quality and say, ‘**I'll know the quality of a product when I see it**’. It seems that we all ‘**know**’ or ‘**feel**’ somehow what the meaning of quality is, though it is very difficult to put it in exact words. Something that fulfills/exceeds customer’s preconceived ideas about the quality is likely to be called as a quality product.

We will try to examine tools and methods which can be used to improve product quality through process approach, add value through brainstorming by producers, consumers, customers and all stakeholders about new features which may be included/old features which may be excluded from the product, decrease costs,

improve schedule and help products to conform better with respect to the expressed and implied requirements. Use of quality tools and methodologies can help people engage in production related activities to improve quality of the products delivered to final user and achieve customer satisfaction.

1.4 DEFINITIONS OF QUALITY

For achieving quality of a product, one must define it in some measurable terms which can be used as a reference to find whether quality is really met or not. There are many views and definitions of quality given by stalwarts working in quality improvement and quality management arena. These definitions describe different perceptions toward quality of products. Some of these are,

1. Customer-Based Definition of Quality Quality product must have '**Fitness for use**' and must meet customer needs, expectations and help in achieving customer satisfaction and possibly customer delight. Any product can be considered as a quality product if it satisfies its purpose of existence through customer satisfaction. This definition is mainly derived by an approach to quality management through '**Quality is fitness for use**'.

2. Manufacturing-Based Definition of Quality This definition is mainly derived from engineering product manufacturing where it is not expected that the customer knows all requirements of the product, and many product level requirements are defined by architects and designers on the basis of customer feedback/survey. Market research may have to generate requirement statement on the basis of perception of probable customers about what features and characteristics of a product are expected by the market. A quality product must have a definition of requirement specifications, design specifications, etc. and the product must conform to these specifications. The development methodologies used for the purpose must be capable of producing the right product in first go and must result into a product having no/minimum defects. This approach gives the definition of '**Conformance to requirements**'.

3. Product-Based Definition of Quality The product must have something that other similar products do not have which can help the customer satisfy his/her needs in a better way. These attributes must add value for the customer/user so that he/she can appreciate the product in comparison to competing products. This makes the product distinguishable from similar products in the market. Also, the customers must feel proud of owning it due to its inherent attributes and characteristics.

4. Value- Based Definition of Quality A product is the best combination of price and features or attributes expected by or required by the customers. The customer must get value for his investment by buying the product. The cost of a product has direct relationship with the value that the customer finds in it. More value for the customer helps in better appreciation of a product. Many times it is claimed that '**People do not buy products, they buy benefits**'.

5. Transcendent Quality To many users/customers, it is not clear what is meant by a 'quality product', but as per their perception it is something good and they may want to purchase it because of some quality present/absent in the product. The customer will derive the value and may feel the pride of ownership.

Definitions 2, 3 and 4 are traditionally associated with the idea of a product quality that,

- A product must have zero/minimum defects so that it does not prohibit normal usage by the users. When users buy a product, they expect minimum/no failures.

- A product must be something that people will want to receive as it satisfies their needs and supports their expectations. It must suffice the purpose of its existence from the customers view.
- It can be purchased at a reasonable price with relation to the value users may derive from it. The customer may undertake cost benefit analysis of the product and if benefits equal or exceed cost, it may be bought.

Customer centric product development forces the industry to look outside its own premises and thought process, making it compulsory to understand the customer. This forces the producer to create products that prospective customers may want to buy and not ones that designers think people want to receive.

The most interesting definition of quality is, '**I do not know what it is, but if I'm delighted by acquiring it, I'll buy it!**' This means that those products are better in quality which possess some characteristics that attract customers to purchase them. Though the requirements of a product may differ from customer to customer, place to place and time to time, in general, the product must be,

- Less expensive with higher returns or higher values for the customer, satisfying cost benefit analysis. Directly or indirectly, every owner would be performing cost benefit analysis before arriving at a decision to purchase something.
- Inherent of required features or attributes expected by the customer which make it fit for use. If product is of no use to users, they will never purchase it.
- Without any defect or with few defects so that its usage is uninhibited and failure or repairs would be as less as possible. If the product is very reliable, it may be liked by prospective buyers.
- With desirable cosmetic attributes.

Often, we are not aware that we want a certain product, but when we see their attributes, we feel like buying it. It may include cosmetic requirements like user interfaces, ease of use, etc.

Many of the above statements are based upon the users or customers perception about quality of the product that they wish to buy. Some of these characteristics are often attributes of products delivered by Japanese manufacturers to consumer market. Mainly, the contributors to quality would be that the failure rates are less, repairs are easier and fast, products are consistent in performance and work better than other similar products in the market and do not give any surprises to customers during use. The reasons for such improvement in the quality are a continuous/continual improvement in all aspects of product development through requirement capturing, design, development, testing, deployment and maintenance.

There is one more angle to the definition of quality of a product. Any improvement in the product quality must result into a better product, and it must give benefits to the customer finally. The benefits may be in terms of more or better features, less wait time, less cost, better service, etc. Thus, quality improvement has direct relationship with fulfilling customer requirements or giving more and more customer satisfaction.

Many people speak about not only achieving customer satisfaction but exceeding customer expectations to achieve customer delight. There is a signal of caution about exceeding customer expectations. Any feature which surprises a customer may not be appreciated by him/her and may be termed as a defect. Exceeding expectations without informing the customer about what they can expect in addition to defined requirements can be dangerous and may result in rejection of such products.

1.5 CORE COMPONENTS OF QUALITY

Quality of a product must be driven by customer requirements and expectations from the product. Those expectations may be expressed as a part of requirement specifications defined or may be implied one which is generally accepted as requirements. It must have some important characteristics that may help

customer in getting more and more benefits and satisfaction by using the product. Some postulates of quality are,

1.5.1 QUALITY IS BASED ON CUSTOMER SATISFACTION BY ACQUIRING A PRODUCT

Quality is something perceived by a customer while using a product. The effect of a quality product, delivered and used by a customer, on his satisfaction and delight is the most important factor in determining whether the quality has been achieved or not. It talks about the ability of a product or service to satisfy a customer by fulfilling his needs or purpose for which it is acquired. It may come through the attributes of a product, time required for a customer to acquire it, price a customer is expected to pay for it and so many other factors associated with the product as well as the organisation producing or distributing it. All these factors may or may not be governed by the manufacturer alone but may be dependent on quality of inputs. This point stresses a need that a producer must understand the purpose or usage of a product and then devise a quality plan for it accordingly, to satisfy the purpose of the product.

1.5.2 THE ORGANISATION MUST DEFINE QUALITY PARAMETERS BEFORE IT CAN BE ACHIEVED

We have already discussed that quality is a perception of a customer about satisfaction of needs or expectation. It is difficult for the manufacturer to achieve the quality of product without knowing what customer is looking for while purchasing it. If product quality is defined in some measurable terms, it can help the manufacturer in deciding whether the product quality has been achieved or not during its manufacturing and delivery. In order to meet some criteria of improvement and ability to satisfy a customer, one must follow a cycle of 'Define', 'Measure', 'Monitor', 'Control' and 'Improve'. The cycle of improvements through measurements is described below,

- *Define* There must be some definition of what is required in the product, in terms of attributes or characteristics of a product, and in how much quantity it is required to derive customer satisfaction. Features, functionalities, and attributes of the product must be measured in quantitative terms, and it must be a part of requirement specification as well as acceptance criteria defined for it. The supplier as well as the customer must know what 'Must be', what 'Should be' and what 'Could be' present in the product so delivered and also what 'Must not be', 'Should not be' and 'Could not be' present in the product.
- *Measure* The quantitative measures must be defined as an attribute of quality of a product. Presence or absence of these attributes in required quantities acts as an indicator of product quality achievement. Measurement also gives a gap between what is expected by a customer and what is delivered to him when the product is sold. This gap may be considered as a lack of quality for that product. This may cause customer dissatisfaction or rejection by the customer.
- *Monitor* Ability of the product to satisfy customer expectations defines the quality of a product. There must be some mechanism available with the manufacturer to monitor the processes used in development, testing and delivering a product to a customer and their outcome, i.e. attributes of product produced using these processes, to ensure that customer satisfaction is incorporated in the deliverables given to the customer. Deviations from the specifications must be analysed and reasons of these deviations must be sorted out to improve product and process used for producing it. An organisation must have correction as well as corrective and preventive action plans to remove the reasons of deviations/deficiencies in the product as well as improve the processes used for making it.

- **Control** Control gives the ability to provide desired results and avoid the undesired things going to a customer. Controlling function in the organisation, popularly called as ‘quality control’ or ‘verification and validation’, may be given a responsibility to control product quality at micro level while the final responsibility of overall organisational control is entrusted with the management, popularly called as ‘quality assurance’. Management must put some mechanism in place for reviewing and controlling the progress of product development and testing, initiating actions on deviations/deficiencies observed in the product as well as the process.
- **Improve** Continuous/continual improvements are necessary to maintain ongoing customer satisfaction and overcome the possible competition, customer complaints, etc. If some producer enjoys very high customer satisfaction and huge demand for his product, competitors will try to enter the market. They will try to improve their products further to beat the competition. Improvement may be either of two different approaches viz. continuous improvement and continual improvement as the case may be.

1.5.3 MANAGEMENT MUST LEAD THE ORGANISATION THROUGH IMPROVEMENT EFFORTS

Quality must be perceived by a customer to realise customer satisfaction. Many factors must be controlled by a manufacturer in order to attain customer satisfaction. Management is the single strongest force existing in an organisation to make the changes as expected by a customer; it naturally becomes the leader in achieving customer satisfaction, quality of product and improvement of the processes used through various programs of continuous/continual improvement. Quality management must be driven by the management and participated by all employees.

Management should lead the endeavor of quality improvement program in the organisation by defining vision, mission, policies, objectives, strategies, goals and values for the organisation and show the existence of the same in the organisation by self examples. Entire organisation should imitate the behavioral and leadership aspects of the management. Every word and action by the management may be seen and adopted by the employees. Quality improvement is also termed as a ‘cultural change brought in by management’.

Organisation based policies, procedures, methods, standards, systems etc. are defined and approved by the management. Adherence to these systems must be monitored continuously and deviation/deficiencies must be tracked. Actions resulting from the observed deviations/deficiencies shall be viewed as the areas which need improvements. The improvements may be required in enforcement or definition of policies and procedures, methods, standards, etc. Management must have quality planning at organisation level to support improvement actions.

1.5.4 CONTINUOUS PROCESS (CONTINUAL) IMPROVEMENT IS NECESSARY

There was an old belief that quality can be improved by more inspection, testing and rework, scrap, sorting, etc. It was expected that a customer must inspect the product and report the defects or deficiencies so that those will be fixed. Manufacturer was responsible for fixing the defects as and when they were reported by the customer. This added to the cost of inspection, segregation, failure, rework, etc. for the customer and reduced the profit margins for the manufacturer or increased the price for the customer. At the same time, customer’s right to receive a good product was withdrawn.

For improving the competitive cost advantage to producer as well as customer, quality must be produced with an aim of first time right and must be improved continuously/continually. For the customer, total cost of product is inclusive of cost of purchase and maintenance. Total cost is more important to him than the purchase tag. The first step for producing quality is the definition of processes used for producing the product and the cycle of

continuous or continual improvement (Plan–Do–Check–Act or Define–Measure–Monitor–Control–Improve) to refine and redefine processes to achieve targeted improvements. It needs ‘Planning’ for quality, ‘Doing’ as per the defined plans, ‘Checking’ the outcome at each stage with the expected results and taking ‘Actions’ on the variances produced. Refer Table 1.1 for comparison between continuous and continual improvement.

Table 1.1

Comparison of continuous and continual improvement

Continuous improvement	Continual improvement
Continuous improvement is dynamic in nature. The changes are done at every stage and every time to improve further.	Continual improvement is dynamic as well as static change management. The changes are done, absorbed, baselined and sustained before taking next step of improvements.
Continuously striving for excellence gives a continuous improvement	Periodic improvements followed by stabilisation process and sustenance represent continual improvement
It has a thrust on continuous refinement of the processes to eliminate waste continuously	Stabilisation of processes at each iteration of improvement where waste is removed in stages
It has high dependence on people having innovative skills tending towards inventions	Less dependence on people and more dependence on innovation processes
Environment is continuously changed	Changes in environment are followed by stabilisation
Sometimes it creates a turbulence in an organisation, if people are not able to digest continuous change	It may be better suited than continuous improvement. It gives a chance to settle the change before next change is introduced

1.6 QUALITY VIEW

Stakeholders are the people or entities interested in success/failure of a project or product or organisation in general. Every project/product/organisation has several stakeholders interested in its betterment. Quality is viewed differently by different stakeholders of the product/project/organisation as per their role in entire spectrum. Some quality models put all stakeholders for the project and product in six major categories. These stakeholders benefit directly or indirectly, if the project/product/organisation becomes successful, and suffer, if the organisation or project fails.

- **Customer** Customer is the main stakeholder for any product/project. The customer will be paying for the product to satisfy his requirements. He/she must benefit by acquiring a new product. Sometimes, the customer and user can be different entities but here, we are defining both as same entity considering customer as a user. Though sometimes late delivery penalty clauses are included in contract, the customer is interested in the product delivery with all features on defined scheduled time and may not be interested in getting compensated for the failures or delayed deliveries.
- **Supplier** Suppliers give inputs for making a project/product. As an organisation becomes successful, more and more projects are executed, and suppliers can make more business, profit and expansion. Suppliers can be external or internal to the organisation. External suppliers may include people supplying machines, hardware, software, etc. for money while internal suppliers may include other functions such as system administrator, training provider, etc. which are supporting projects/product development.

• **Employee** People working in a project/an organisation may be termed as employees. These people may be permanent/temporary workers but may not be contractual labours having no stake in product success. (Contractual workers may come under supplier category.) As the projects/organisations become successful, people working on these projects/in these organisations get more recognition, satisfaction, pride, etc. They feel proud to be part of a successful mission.

• **Management** People managing the organisation/project may be termed as management in general. Management may be divided further into project management, staff management, senior management, investors, etc. Management needs more profit, recognition, turnover improvements, etc to make their vision and mission successful. Successful projects give management many benefits like expanding customer base, getting recognition, more profit, more business, etc.

There are two more stakeholders in the success as well as failure of any project/product/organisation. Many times, we do not feel their existence at project level or even at organisation level. But they do exist at macro level.

• **Society** Society benefits as well as suffers due to successful projects/organisations. It is more of a perception of an individual looking towards the success of the organisation. Successful organisations/projects generate more employment, and wealth for the people who are in the category of customer, supplier, employee, management, etc. It also affects the resource availability at local as well as global level like water, roads, power supply, etc. It also affects economics of a society to a larger extent. Major price rise has been seen in industry dominated areas as the paying capacity of people in these areas is higher than other areas where there is no such industry.

• **Government** Government may be further categorised as local government, state government, central government, etc. Government benefits as well as suffers due to successful projects/organisations. Government may get higher taxes, export benefits, foreign currency, etc. from successful projects/organisations. People living in those areas may get employment and overall wealth of the nation improves. At the same time, there may be pressure on resources like water, power, etc. There may be some problems in terms of money availability and flow as success leads to more buying power and inflation.

Quality perspective of all these stakeholders defines their expectations from organisation/projects. We may feel that superficially these views may differ from each other though finally they may be leading to the same outcome. If these views match perfectly and there is no gap in the stakeholder's expectations, then organisational performance and effectiveness can be improved significantly as collective efforts from all stakeholders. If the views differ significantly, this may lead to discord and hamper improvement.

Let us discuss two important views of quality which mainly defines the expectations from a project and success of a project at unit level viz. customer's view and developer's view of quality.

1.6.1 CUSTOMER'S VIEW OF QUALITY

Customer's view of quality of product interprets customer requirements and expectation for getting a better product at defined schedule, cost and with adequate service along with required features and functionalities. Customer is paying some cost to get a product because he finds value in such acquisition.

Delivering Right Product The products received by customers must be useful to satisfy their needs and expectations. It may or may not be the correct product from manufacturer's perspective or what business analyst/system designer may think. There is a possibility that development team including testers

may ask several queries about the requirements of the product to get them clarified but the final decision about the requirements definition shall be with customer. If customer confirms that the requirements are correct/not correct, then there is no possibility of further argument about the validity/invalidity of requirements.

Satisfying Customer's Needs The product may or may not be the best product, as per the manufacturer's views or those which are available in the market, which can be made from the given set of requirements and constraints. There are possibilities of different alternatives for overcoming the constraints and implementing the requirements. Basic constraint in product development and testing is that product must be capable of satisfying customer needs. Needs are 'must' among requirements from customer's perspective. They may be part of processes for doing requirement analysis and selection of approach for designing on the basis of decision analysis and resolution, using some techniques such as cost-benefit analysis, etc. which suites best in the given situation. This must help organisation to achieve customer satisfaction through product development and delivery.

Meeting Customer Expectations Customer expectations may be categorised into two parts viz. expressed expectations and implied expectations. Expectations documented and given formally by the customer are termed as 'expressed requirements' while 'implied expectations' are those, which may not form a part of requirement specifications formally but something which is expected by customer by default. It is a responsibility of a developing organisation to convert, as many as possible, implied requirements into expressed requirements by asking queries or eliciting requirements. One must target for 100% conversion of implied requirements into expressed requirements, though difficult, as developer may refuse to accept the defect belonging to implied requirements simply because it is not a part of requirement statement and they may not be aware of such things.

Treating Every Customer with Integrity, Courtesy, and Respect Customer and the requirements assessed (both expressed as well as implied) are very important for a developing organisation as customer will be paying on the basis of value he finds in the product. Definition of the requirements is a first step to satisfy the customer through '**Conformance to Requirements**'. Requirements may be documented by anybody, generally development team, but customer is the owner of the requirements. Organisation shall believe and understand that the customer understands what is required by him. How to achieve these requirements is a responsibility of a developing organisation. He may be given suggestions, sharing some past experiences and knowledge gained in similar projects but manufacturer shall not define requirements or thrust or push the requirements to customer. It is quite often quoted that the customer does not know or understand his requirements. This opinion cannot be bought by anybody.

Customer telephone calls and mails must be answered with courtesy and in reasonable time. The information provided to the customer must be accurate and he/she must be able to depend on this information. The customer is not a hindrance to the project development but he is the purpose of the business and producer must understand this.

1.6.2 SUPPLIER'S VIEW OF QUALITY

Supplier is a development organisation in the context of software application development. Supplier has some expectations or needs, which must be satisfied by producing a product and selling it to customer. Supplier expectations may range from profitability, name in market, repeat orders, customer satisfaction, etc. These expectations may be fulfilled in the following ways,

Doing the Right Things Supplier is intended to do right things for the first time so that there is no waste, scrap, rework, etc. Wastes like rework, scrap, and repairs are produced by hidden factory for which there is no customer. Changes in requirements are considered as problems during product development, if supplier is expected to absorb the costs associated with it. Changes in requirement cause rework of design, development, testing etc. This adds to the cost of development but if the price remains same, it is a loss for manufacturer. It also adds to fatigue and frustration of the people involved in development as they find no value in reworking, sorting, scrapping, etc. Following right processes to get the product required by the customer and achieving customer satisfaction and profits as well as job satisfaction may be the expectations of a supplier. Suppliers may require clear and correct definition of deliverables, time schedules, attributes of product, etc.

Doing It the Right Way A producer may have his own methods, standards and processes to achieve the desired outputs. Sometimes, customer may impose the processes defined by him for building the product on the producer. Ability of development process to produce the product as required by the customer defines the capability of development process. These process definitions may be an outcome of quality standards or models or business models adopted by the supplier or customer. As organisation matures, the processes towards excellence—by refining and redefining processes and process capability—must improve continually/continuously. As the processes reach optimisation, they must result into better quality products and more satisfaction for the customer and also fulfill vendor requirements.

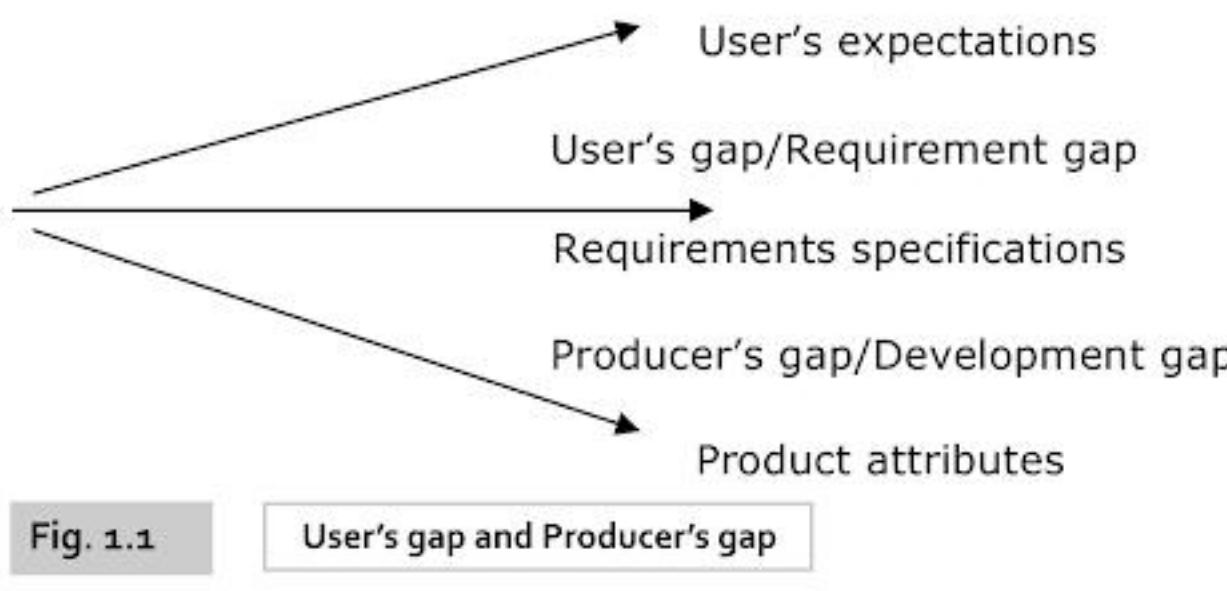
Doing It Right the First Time Doing right things at the first time may avoid frustration, scrap, rework, etc. and improve profitability, reduce cost and improve customer satisfaction for the supplier. Doing right things at the first time improves performance, productivity and efficiency of a manufacturing process. This directly helps in improving profitability and gives advantage in a competitive market. Supplier would always like to follow the capable processes which can get the product right at the first attempt.

Doing It on Time All resources for developing a new product are scarce and time factor has a cost associated with it. The value of money changes as time changes. Thus, money received late is as good as less money received. If the customer is expected to pay on each milestone, then the producer has to deliver milestones on time to realise money on time. Delay in delivery represents a problem with processes, starting from getting requirements, estimation of efforts and schedule and goes upto delivery and deployment.

Difference between the two views discussed above (Customer's view vs Supplier's view) creates problem for manufacturer as well as customer when it comes to requirement gathering, acceptance testing, etc. It may result into mismatch of expectation and service level, which would be responsible for introducing defects in the product in terms of functionalities, features, cost, delivery schedule, etc. Sometimes these views are considered as two opposite sides of a coin or two poles of the Earth which can never match. The difference between two views is treated as a gap.

In many cases, the customer wants to dictate the terms as he is going to pay for the product and the processes used for making it, while the supplier wants customer to accept whatever is produced. Wherever the gap is less, the ability of a product to satisfy customer needs is considered better. At the same time, it helps a development organisation as well as a customer to get their parts of benefits from steady processes. On the contrary, larger gap causes more problems in development, distorted relations between two parties and may result into loss for both. Quality processes must reduce the gap between two views effectively. Effectiveness of a quality process may be defined as the ability of the processes and the product to satisfy both or all stakeholders in achieving their expectations.

Figure 1.1 explains a gap between actual product, requirements for the product and customer expectations from the product. It gives two types of gaps, viz. user's gap and producer's gap.



1.6.3 USER'S GAP/REQUIREMENT GAP

User's gap is a gap between requirement specifications for the product and user expectations from it. This gap focuses on the difference in the final product attributes as defined by requirement statements with respect to the intents of the user. Developers must convert user needs into the product requirement specifications and create the product exactly as per these specifications. This may need understanding and interpretation of customer's business flow and requirements, and how the product is intended to be used by the customer to satisfy his business requirements.

Closing User's Gap User's gap represents the failure seen by the customer in terms of difference between user needs and product specifications. An organisation must apply some processes and methods so that user's gap can be closed effectively or reduced to as little as possible. Methods for closing these gaps may depend on organisation's way of thinking, resource availability, type of customer, customer's thought process, etc. Some of these methods are mentioned below.

Customer Survey Customer surveys are essential when an organisation is producing a product for a larger market where a mismatch between the product and expectation can be a major problem to producer. It may also be essential for projects undertaken for a single customer such as mission critical projects where failure of the project has substantial impact on the customer as well as producer. For a larger market, survey may be conducted by marketing function or business analyst to understand user requirements and collate them into specification documents for the product. Survey teams decide present and future requirements for the product and the features required by the potential customers.

For a single customer, a survey is conducted by business analyst and system analyst to analyse the intended use of the product under development and the possible environment of usage. Domain experts from the development side may visit the customer organisation to understand the specific requirements and workflow related to domain to incorporate it into the product to be developed.

Joint Application Development (JAD) Applications are developed jointly by customer and manufacturer where there is close co-ordination between two teams. In joint application development, users or customers may be overseeing the system development and closely monitor requirements specifications, architecture, designs, coding, testing and test results. The applications produced by this method may follow top-down approach where user interfaces and framework are developed first and approved by the users and then logic is built behind it. Some people may call joint application development as an agile methodology of development where developers collaborate with customer during development.

User Involvement in Application Development This approach works on the similar lines of joint application development (JAD). User may be involved in approving requirement specifications, design specification, application user interfaces, etc. He/she may have to answer the queries asked by developers and provide clarifications, if any. An organisation may develop a prototype, model, etc. to understand user requirements and get approval from the user team. If the organisation is producing products for a larger market, few representative users may be considered for collecting requirement specifications, test case designing and acceptance testing of the application under development.

1.6.4 PRODUCER'S GAP/DEVELOPMENT GAP

Producer's gap is a gap between product actually delivered and the requirement and design specifications developed for the product. The requirement specifications written by business analyst may not be understood in the same way by the development team. There are communication losses at each stage and business analyst and developers/testers may not be located next to each other to provide explanation for each and every requirement. More stages of communication may lead to more gaps and more distortion of requirements. The product so produced and requirement specifications used may differ significantly creating producer's gap.

Closing Producer's Gap Producer's gap represents a failure on part of development team to convert requirements into product. Producer's gap can be seen as the defects found in in-house testing. Producer's gap is due to process failure at producer's place and there must be process improvement plans to close this gap.

Process Definition Development and testing processes must be sufficiently mature to handle the transfer of information from one person or one stage to another during software development life cycle. There must be continuous 'Do' and 'Check' processes to build better products and get feedback about the process performance. Such product development has life cycle testing activities associated with development.

Work Product Review As the stages of software development life cycle progresses, one may have to keep a close watch on artifacts produced during each stage to find if any inconsistency has been introduced with respect to earlier phase. Generation of requirement traceability matrix is an important factor in this approach.

1.7 FINANCIAL ASPECT OF QUALITY

Earlier, people were of the opinion that more price of a product represents better quality as it involves more inspection, testing, sorting, etc. and ensures that only good parts are supplied to the customer. Sales price was defined as,

$$\text{Sales price} = \text{Cost of manufacturing} + \text{Cost of Quality} + \text{Profit}$$

If we consider the monopoly way of life, this approach may be considered good since the price is decided by the manufacturer depending upon three factors described above. Unfortunately, monopoly does not exist in real world. If any product enjoys higher profitability, more number of producers would enter into competition. Number of sellers may exceed number of buyers. When the products produced match in all aspects, the cost would decide the quantity of sale. The competitor who reduces the price, may get more volume of sale if all other things remain constant. Reducing the sales price reduces percentage profit. For maintaining profit, the producer may try to reduce cost of production without compromising on the quality aspect.

Thus, in a competitive environment, the equation changes to

$$\text{Profit} = \text{Sales price} - [\text{Cost of manufacturing} + \text{Cost of Quality}]$$

1.7.1 COST OF MANUFACTURING

Cost of manufacturing is a cost required for developing the right product by right method at the first time. The money involved in resources like material, people, licenses, etc. forms a cost of manufacturing. The cost of manufacturing remains constant over the time span for the given project and given technology and it has a direct relationship with the efforts. It can be reduced through improvements in technology and productivity but it may need longer time frame. The cost involved in requirement analysis, designing, development and coding are the costs associated with manufacturing.

1.7.2 COST OF QUALITY

Cost of quality represents the part of cost of production incurred in improving or maintaining quality of a product. Some people keep cost of manufacturing and cost of quality as separate while others may include them under cost of production. Cost of manufacturing may be supported by cost of quality and there exists an interrelationship between the two costs.

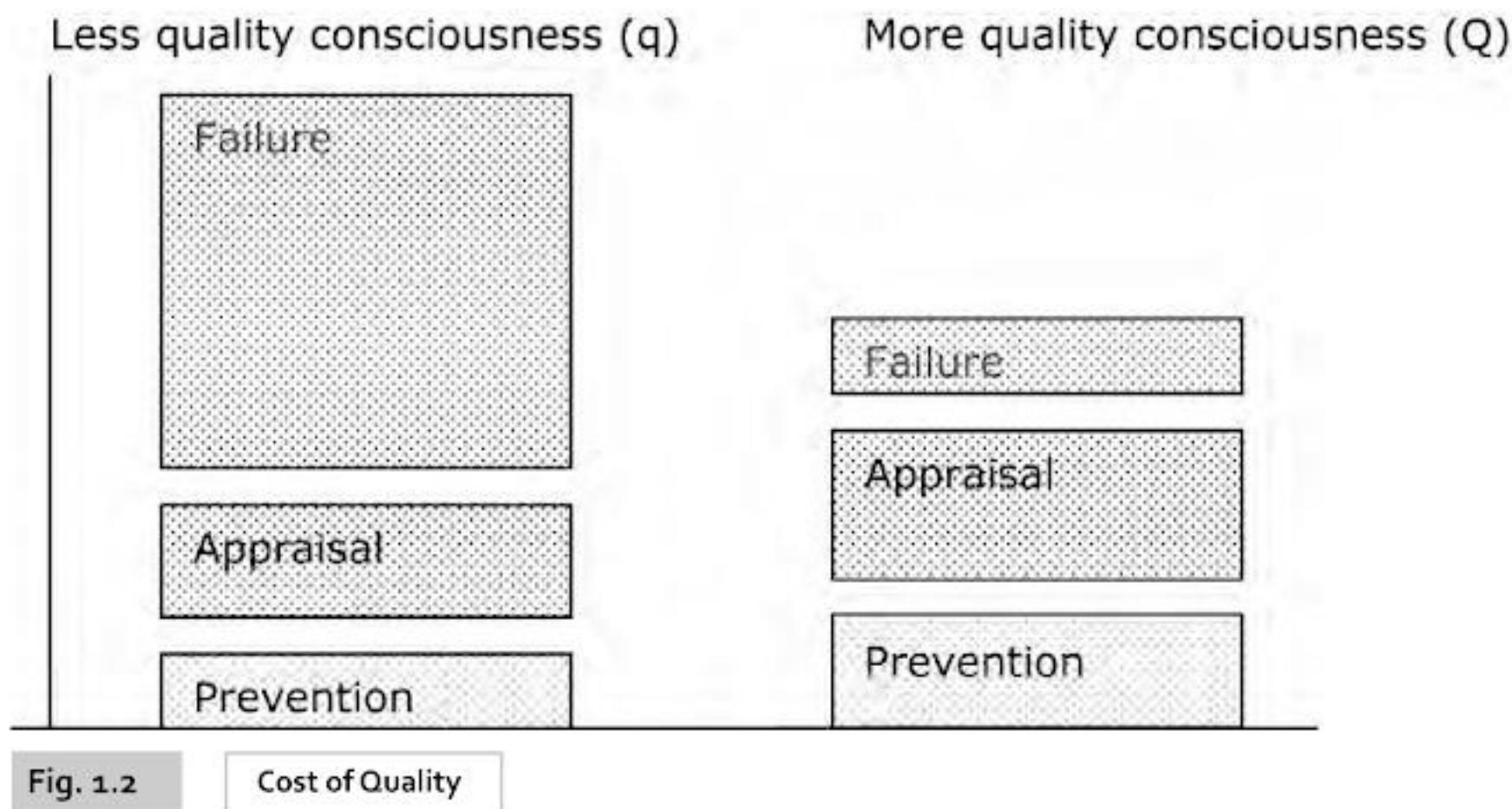
Cost of quality includes all the efforts and cost incurred in prevention of defects, appraisal of product to find whether it is suitable to customer or not and fixing of defects or failures at various levels as and when they are reported and conducting any retesting, regression testing, etc.

Cost of Prevention An organisation may have defined processes, guidelines, standards of development, testing, etc. It may define a program of imparting training to all people involved in development and testing. This may represent a cost of prevention. Creation and use of formats, templates, etc. acquiring various process models and standards, etc. also represent a cost of prevention. This is an investment by an organisation and it is supposed to yield returns. This is also termed as '**Green money**'. Generally, it is believed that 1 part of cost of prevention can reduce 10 parts of cost of appraisal and 100 parts of cost of failure.

Cost of Appraisal An organisation may perform various levels of reviews and testing to appraise the quality of the product and the process followed for developing the product. The cost incurred in first time reviews and testing is called as the cost of appraisal. There is no return on investment but this helps in identifying the process capabilities and process related problems, if any. This is termed as '**Blue money**' as it can be recovered from the customer. Generally, it is believed that 1 part of cost of appraisal can reduce 10 parts of cost of failure.

Cost of Failure Cost of failure starts when there is any defect or violation detected at any stage of development including post delivery efforts spent on defect fixing. Any extent of rework, retesting, sorting, scrapping, regression testing, late payments, sales under concession, etc. represents cost of failure. There may be some indirect costs such as loss of goodwill, not getting customer references, not getting repeat orders, and customer dissatisfaction associated with the failure to produce the right product. The cost incurred due to some kind of failure is represented as cost of failure. This is termed as '**Red money**'. This cost affects the profitability of the project/organisation badly.

On the basis of quality focus, organisations may be placed in 2 categories viz. organisations which are less quality conscious (termed as 'q') and organisations which are more quality conscious (termed as 'Q'). The distribution of cost of quality for these two types may be represented as below. Please refer Fig. 1.2



The bottommost rectangle represents cost of prevention. As the organisation's quality consciousness increases, prevention cost increases substantially due to introduction of various process requirements. The organisation may have defined processes, methods, work instructions, standards, guidelines, templates, formats etc. Teams are supposed to use them while building a product or testing it, and conduct audits which need resources, time and money. Project teams may create project plan, test plan, assess the risks and issues faced during development, decide on the actions to reduce their probabilities/impacts, etc. More cost of prevention may be justifiable for a project/product only if it reduces the cost of failure.

Middle rectangle represents cost of appraisal. As quality consciousness increases, cost of appraisal also increases. An organisation may prepare various plans (such as quality plan and test plan) and then review these plans, write the test cases, define test data, execute test cases, analyse defects and initiate actions to prevent defect recurrence. There may be checklists, guidelines, etc. used for verification and validation activities. The cost incurred in appraisal must be justifiable and must reduce cost of failure.

The topmost rectangle represents cost of failure. Cost of failure includes the cost associated with any failure which may range from rework, retesting, etc. including customer dissatisfaction or loss of revenue, etc. As quality consciousness improves, failure cost must reduce representing better quality of products offered and higher customer satisfaction. Thus the overall cost of quality must reduce as quality consciousness of the organisation increases.

1.8 DEFINITION OF QUALITY

Let us try to redefine and understand the meaning of the term 'Quality' with a new perspective. Many definitions of the word 'Quality' are available and are used at different forums by different people. Most of these definitions show some aspect of quality. While no definition is completely wrong, no definition is completely right also. Few of the definitions prescribed for quality are,

- Predictable Degree of Uniformity, Dependability at Low Cost and Suited to Market** This definition stresses on quality as an attribute of product which is predictable and uniform in

behavior throughout product usage and stresses on the ability of a product to give consistent results again and again. One must be able to predict the product behavior beforehand. It talks about reduction in variability of a product in terms of features, performance, etc. It also defines the dependability aspect of quality product. One must expect reliable results from quality products every time they are used. Quality product must be the cheapest one as it talks about reducing failure cost of quality like rework, scrap, sorting, etc. The most important thing about product quality is that it must help the product to suite the market needs or expectations. This necessitates that the manufacturer should produce those products which can be sold in the market.

- **Degree to Which a Set of Inherent Characteristics of the Product/Service Fulfils the Requirements** This definition of quality stresses the need that the product must conform to defined and documented requirement statement as well as expectations of users. Higher degree of fulfillments of these requirements makes a product better in terms of quality. There must be something in the product, defined as ‘attributes of product’, which ensures customer satisfaction. The attributes must be inborn in the product, indicating capable processes used for developing such a product.
- **Ability of a Product or Service That Bears Upon Its Ability to Satisfy Implied or Expressed Need** This definition of quality of product is derived from the approach of ‘Fitness for use’. It talks about a product achieving defined requirements as well as implied requirements, satisfying needs of customer for which it is being used. Better ability of a product to fulfill requirements makes a product better, from quality perspective.

1.9 CUSTOMERS, SUPPLIERS AND PROCESSES

For any organisation, there are some suppliers supplying the inputs required and some customers who will be buying the outputs produced. Suppliers and customers may be internal or external to the organisation. In the larger canvas, an entire organisation can be viewed as the component in a huge supply chain of the world where products are made by converting some inputs which may act as inputs to the next stage. External suppliers provide input to the organisation and external customers receive the output of the organisation. In turn, suppliers may be customers for some other organisations and customers may be acting as suppliers for somebody else down the line.

Internal Customer Internal customers are the functions and projects serviced and supported by some other functions/projects. System administration may have projects as their customer while purchasing may have system administration as their customer. During value chain, each function must understand its customers and suppliers. Each function must try to fulfill its customer requirements. This is one of the important considerations behind ‘Total Quality Management’ where each and every individual in supply chain must identify and support his customer. If internal customers are satisfied, this will automatically satisfy external customer as it sets the tone and perspective for everybody.

External Customer External customers are the external people to the organisation who will be paying for the services offered by the organisation. These are the people who will be actually buying products from the organisation. As the organisation concentrates on external customer for their satisfaction, it must improve quality of its output.

1.10 TOTAL QUALITY MANAGEMENT (TQM)

‘Total quality management’ principle intends to view internal and external customers as well as internal and external suppliers for each process, project and for entire organisation as a whole. The process and factions

of an organisation can be broken down into component elements, which act as suppliers/customers to each other during the workflow. Each supplier eventually also becomes a customer at some other moment and vice versa. If one can take care of his/her customer (whether internal or external) with an intention to satisfy him, it may result into customer satisfaction and continual improvement for the organisation.

Supply chain relationship may be defined graphically as, shown in Fig. 1.3.

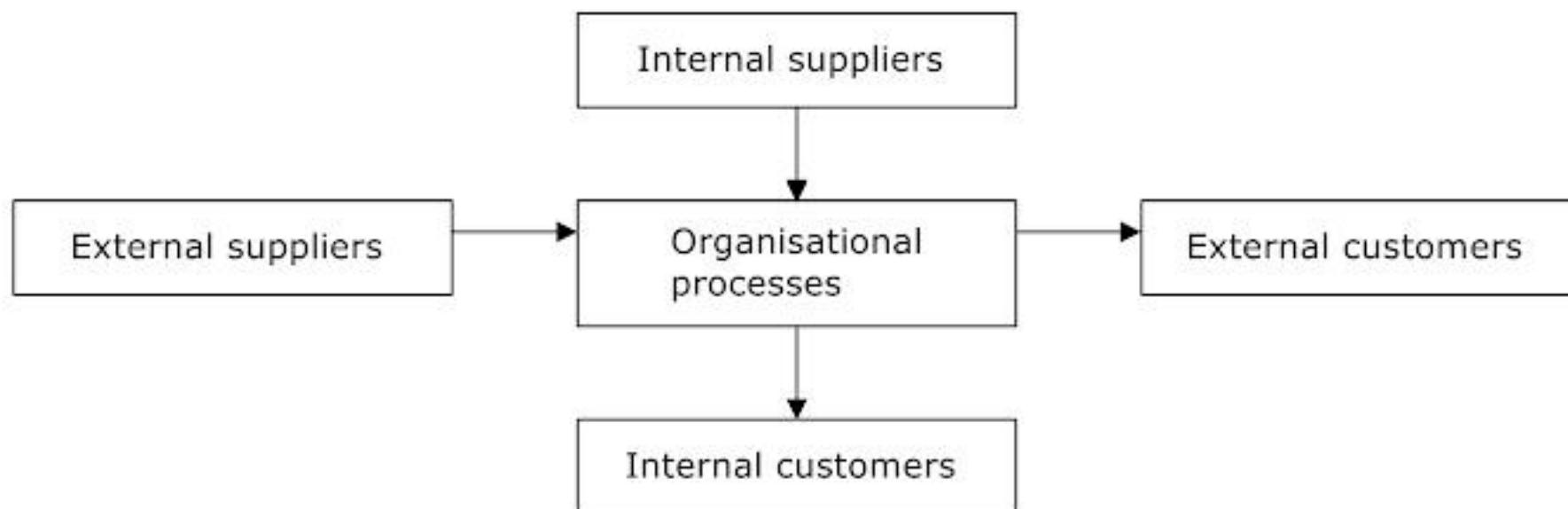


Fig. 1.3

Supply chain relationship between suppliers and customers

'Total quality management' (TQM) is the application of quality principles to all facets and business processes of an organisation. It talks about applying quality methods to the entire organisation whether a given function or part of the organisation faces external customer(s) or not. One clear definition of quality involves satisfying one's customer irrespective of whether he/she is outside or inside the organisation. This implementation of customer satisfaction has different meanings for different parts of an organisation.

Quality Management Approach Dr Edward Deming implemented quality management system driven by '**Total Quality Management**' and '**Continual Improvement**' in Japanese environment. It resulted into repetitive, cost effective processes with an intention to satisfy customer requirements and achieve customer satisfaction. Implementation was inclined toward assessment of quality management system which was adoptive to the utility of tools for understanding data produced by the process measurements. Dr Deming proposed principles for quality management that are widely used by the quality practitioners.

1.11 QUALITY PRINCIPLES OF 'TOTAL QUALITY MANAGEMENT'

'Total quality management' works on some basic principles of quality management definition and implementation. These have evolved over a span of experimentation and deployment of quality culture in organisations.

1.11.1 DEVELOP CONSTANCY OF PURPOSE OF DEFINITION AND DEPLOYMENT OF VARIOUS INITIATIVES

Management must create constancy of purpose for products and processes, allocating resources adequately to provide for long term as well as short term needs rather than concentrating on short term profitability: suppliers and organisation must have an intent to become competitive in the world, to stay in business and to provide jobs to people and welfare of the society. The processes followed during entire lifecycle of product

development from requirement capturing till final delivery must be consistent with each other and must be followed over a larger horizon. Decisions taken by management at different instances must be consistent and based upon same standards, rules and regulations. Different initiatives by management must have relationships with each other and must be able to satisfy the vision of the organisation.

1.11.2 ADAPTING TO NEW PHILOSOPHY OF MANAGING PEOPLE/ STAKEHOLDERS BY BUILDING CONFIDENCE AND RELATIONSHIPS

Management must adapt to the new philosophies of doing work and getting the work done from its people and suppliers. We are in a new economic era where skills make an individual indispensable. The process started in Japan and is perceived as a model throughout the world for improving quality of working as well as products. We can no longer live with commonly accepted levels of delays, mistakes, defective materials, rejections and poor workmanship. Transformation of management style to total quality management is necessary to take the business and industry on the path of continued improvements.

1.11.3 DECLARE FREEDOM FROM MASS INSPECTION OF INCOMING/ PRODUCED OUTPUT

It was a common belief earlier that for improving quality of a product, one needs to have rigorous inspection program followed by huge rework, sorting, scrapping, etc. It was believed that one must check everything to ensure that no defect goes to customer. But there is a need for change in thinking of management and people as mass inspection results into huge cost overrun and product produced is of inferior quality. There must be an approach for elimination of mass inspection followed by cost of failure as the way to achieve quality of products. Improving quality of products requires setting up the right processes of development and measurement of process capabilities and statistical evidence of built-in quality in all departments and functions.

1.11.4 STOP AWARDING OF LOWEST PRICE TAG CONTRACTS TO SUPPLIERS

Organisations must end the practice of comparing unit purchase price as a criterion of awarding contracts. Vendor selection must be done on the basis of total cost including price, rejections, etc. Organisations must perform measurements of quality of supply along with price and do the source selection on the basis of final cost paid by it in terms of procurement, rework, maintenance, operations, etc. It must reduce the number of suppliers by eliminating those suppliers that do not qualify with statistical evidences of quality of their supply. This will automatically reduce variation and improve consistency. Aim of vendor selection is to minimise total cost, not merely initial cost of purchasing, by minimising variations in the vendor supplied product. This may be achieved by moving towards lesser/single supplier, on a long term relationship of loyalty and trust. Organisations must install statistical process control in development even at vendor site and reduce the variation in place of inspecting each and every piece.

1.11.5 IMPROVE EVERY PROCESS USED FOR DEVELOPMENT AND TESTING OF PRODUCT

Improve every process of planning, production and service to the customer and other support processes constantly. Processes have interrelationships with each other and one process improvement affects other processes positively. Search continually for problems in these processes in terms of variations in order to

improve every activity in the organisation—to improve quality and productivity and decrease the cost of production as well as cost of quality continuously. Institutionalise innovations and improvements of products and processes used to make them. It is management's job to work continually for optimising processes.

1.11.6 INSTITUTIONALISE TRAINING ACROSS THE ORGANISATION FOR ALL PEOPLE

An organisation must institute modern methods of training which may include on-the-job training, classroom training, self study, etc. for all people, including management, to make better use of their abilities. New skills are required to keep up with the changes in materials, methods, product and service design, infrastructure, techniques and service. Skill levels of people can be enhanced to make them suitable for better performance by planning different training programs. Training—technical as well as non-technical—should focus on improving present skills and acquiring new skills. Customer requirements may also change and people may need training to understand changes in customer expectations. Suppliers may be included in training programs to derive better output from them.

1.11.7 INSTITUTIONALISE LEADERSHIP THROUGHOUT ORGANISATION AT EACH LEVEL

An organisation must adopt and institute leadership at all levels' with the aim of helping people to do their jobs in a better way. Responsibility of managers and supervisors must be changed from controlling people to mentoring, guiding, and supporting them. Also, their focus should shift from number of work items to be produced to quality of output. Improvement in quality will automatically improve productivity by reducing scrap, inspection, rework, etc. Management must ensure that immediate actions are taken on reports of inherited defects, maintenance requirements, poor tools, fuzzy operational definitions and all conditions detrimental to quality of products and services offered to final users.

1.11.8 DRIVE OUT FEAR OF FAILURE FROM EMPLOYEES

An organisation must encourage effective two-way communication and other means to drive out fear of failure from the minds of all employees. Employees can work effectively and more productively to achieve better quality output when there is no fear of failure. People may not try new things if they are punished for failure. Management should not stop or discount feedback coming to them even if it is negative. Giving positive as well as negative feedback should be encouraged, and it must be used to perform SWOT analysis followed by actions. Fear is something which creates stress in minds of people, prohibiting them from working on new ways of doing things. Fear can cause disruption in decision-taking process which may result into excessive defense and also, major defects in the product. People may not be able to perform under stress.

1.11.9 BREAK DOWN BARRIERS BETWEEN FUNCTIONS/DEPARTMENTS

Physical as well as psychological breaking down of barriers between departments and staff areas may create a force of cohesion. People start performing as a team and there is synergy of group activities. People in different areas must work as a team to tackle problems that may be encountered with products and customer satisfaction. Ultimate aim of the organisation must be to satisfy customers. All people working together and helping each other to solve the problems faced by customers can help in achieving this ultimate aim.

1.11.10 ELIMINATE EXHORTATIONS BY NUMBERS, GOALS, TARGETS

Eliminate use of slogans, posters and exhortations of the work force, demanding 'Zero Defects' and new levels of productivity, without providing methods and guidance about how to achieve it. Such exhortations create adverse relationships between supervisors and workers. Main cause of low quality and productivity is in the processes used for production as the numbers to be achieved may be beyond the capability of the processes followed by the workers. This may induce undue frustration and stress and may lead to failures. The Organisation shall have methods to demonstrate that the targets can be achieved with smart work.

1.11.11 ELIMINATE ARBITRARY NUMERICAL TARGETS WHICH ARE NOT SUPPORTED BY PROCESSES

Eliminate work standards that prescribe quotas for the work force and numerical goals for managers to be achieved. Substitute the quotas with mentoring and support to people, and helpful leadership in order to achieve continual improvement in quality and productivity of the processes. Numerical goals should not become the definition of achievement/targets. There must be a methodology to define what achievement is and what must be considered as a stretched target.

1.11.12 PERMIT PRIDE OF WORKMANSHIP FOR EMPLOYEES

Remove the barriers that take away the pride of workmanship for workers and management. People must feel proud of the work they are doing, and know how they are contributing to organisational vision. This implies complete abolition of the annual appraisal of performance and of 'management by objective'. Responsibility of managers and supervisors must be changed from sheer numbers to quality of output. Management must understand 'managing by facts'.

1.11.13 ENCOURAGE EDUCATION OF NEW SKILLS AND TECHNIQUES

Institute a rigorous program of education and training for people working in different areas and encourage self-improvement programs for everyone. What an organisation needs is not just good people; it needs people who can improve themselves with education to accept new challenges. Advances in competitive position will have their roots in knowledge gained by people during such trainings.

1.11.14 TOP MANAGEMENT COMMITMENT AND ACTION TO IMPROVE CONTINUALLY

Clearly define top management's commitment to ever-improving quality and productivity and their obligation to implement quality principles throughout the organisation. It is not sufficient that the top management commits for quality and productivity but employees must also see and perceive their commitment. They must know what it is that they are committed to—i.e., what they must do in order to show their commitment.

1.12 QUALITY MANAGEMENT THROUGH STATISTICAL PROCESS CONTROL

Dr Joseph Juran is a pioneer of statistical quality control with a definition of improvement cycle through Define, Measure, Monitor, Control and Improve (DMMCI). One must understand the interrelationships among

customers, suppliers and processes used in development, testing, etc. and establish quality management based on metrics program. There are three parts of the approach, namely,

1.12.1 QUALITY PLANNING AT ALL LEVELS

Quality is not an accident but is a result of deliberate effort to achieve something which is defined in advance. An organisation must have a definition of what they wish to achieve. Quality improvement must be planned at all levels of organisation and then only it can be achieved. Quality planning happens at two levels viz. organisation level and individual department function project level.

- *Quality Planning at Organisation Level* Quality must be planned at organisation level first. It must be in the form of policy definition and strategic quality plans on the basis of vision, mission(s) and policies set by senior management. Planning process must attempt to discover who the customers are at present and who will be the customers in future, and what are and will be their needs and expectations from the organisation. The needs of the present or future customers must be expressed in numeric terms so that the actions can be planned and progress can be measured. The presence or absence of different attributes to the extent required shall define the quality level of the product or services.
- *Quality Planning at Unit Level* Quality planning at unit level must be done by the people responsible for managing the unit. Operational quality plans must be in sync with organisational policies and strategies. Project plan and quality plan at unit level must be consistent with the strategic quality plans at organisation level and must be derived from the organisation's vision and mission(s).

1.12.2 QUALITY CONTROL

Quality control process attempts to examine the present product at various levels with the defined standards so that an organisation may appraise the outcome of the processes. Removing defects in the processes to improve their capability can help to reach new levels of improved quality. (e.g., process improvement must be targeted towards lowering defects, reducing cost, and improving customer satisfaction.) It must measure the deviations with respect to the number of achievements planned in quality planning so that the organisation can initiate the actions to reduce the deviations to minimum level.

1.12.3 QUALITY IMPROVEMENT

Improvement process attempts to continuously improve the quality of the process used for producing products. Quality of the process is measured on the basis of the attributes of the products produced. There is no end to quality improvements and it needs to take newer challenges again and again. Finding deviations in the attributes of products and processes with respect to planned levels and permissible tolerances available shall guide the organisation to find the weak areas where actions may be prioritised.

1.13 QUALITY MANAGEMENT THROUGH CULTURAL CHANGES

Philip Crosby's approach to quality improvement is based on cultural change in an organisation towards total quality management. Quality management through cultural change defines quality improvements as a cultural change driven by management. It involves,

- Identifying areas in which quality can be improved depending upon process capability measurements and organisational priorities. An organisation must setup crossfunctional working groups (quality circles or

quality improvement teams) and try to improve awareness about the customer needs, quality and process measurements. The organisation may not be able to improve in all areas at a time and prioritisation may be essential depending upon some techniques such as Pareto analysis, Cost benefit analysis, etc. It must prioritise the improvements depending upon the resources available and efforts/investments required and the benefits derived from such improvements.

- Instituting teams representing different functions and areas for quality improvement can help in setting the change of culture. Improving quality of the processes of development, testing, managing, etc. is a team work led by management directives. A single person may not be able to institutionalise improvements across the organisation. Improvements in processes automatically improve the product and customer satisfaction.
- Setting measurable goals in each area of an organisation can help in improving processes at all levels. Goals may act as stretched targets with respect to what is currently achieved by the organisation. Goals may be set with reference to customer expectations or something which may give competitive advantage to the organisation in the market.
- Giving recognition to achievers of quality goals will boost their morale and set a positive competition among the teams leading to organisational improvements. This can lead to dramatic improvements in all areas. Management must demonstrate commitment to quality improvement, and recognition of achievements is a step in this direction.
- Repeating quality improvement cycle continuously by stretching goals further for next phase of improvements is required to maintain and improve the status further. The organisation must evaluate the goals to be achieved in short term, long term, and the combination of both to realise organisational vision.

1.14 CONTINUAL (CONTINUOUS) IMPROVEMENT CYCLE

Plan, Do, Check, and Act (PDCA) Cycle Continual (Continuous) improvement cycle is based on systematic sequence of Plan–Do–Check–Act activities representing a never ending cycle of improvements. PDCA cycle was initially implemented in agriculture. It was implemented later in the electronic industry. TQM has made the PDCA cycle famous in all industries. PDCA improvement cycle can be thought of as a wheel of improvement continually (continuously) rolling up the problem-solving hill and achieving better and better results for the organisation in each iteration. Stages of continual (Continuous) improvement through PDCA cycle are,

Plan An organisation must plan for improvements on the basis of its vision and mission definition. Planning includes answering all questions like who, when, where, why, what, how, etc. about various activities and setting expectations. Expected results must be defined in quantitative terms and actions must be planned to achieve answers to these questions. Quality planning at unit level must be in sync with quality planning at organisation level. Baseline studies are important for planning. Baseline studies define where one is standing and vision defines where one wishes to reach.

Do An organisation must work in the direction set by the plan devised in earlier phase for improvements. Plan is not everything but a roadmap. It sets the direction but execution is also important. Actual execution of a plan can determine whether the results as expected are achieved or not. Plan sets the tone while execution makes the plan work. ‘Do’ process need inputs like resources, hardware, software, training, etc. for execution of a plan.

Check An organisation must compare actual outcome of ‘Do’ stage with reference or expected results which are planned outcomes. It must be done periodically to assess whether the progress is in proper direction or not, and whether the plan is right or not. Expected and actual results must be in numerical terms, and compared at some periodicity as defined in the plan.

Act If any deviations (positive or negative) are observed in actual outcome with respect to planned results, the organisation may need to decide actions to correct the situation. The actions may include changing the plan, approach or expected outcome as the case may be. One may have to initiate corrective and/or preventive actions as per the outcome of 'Check'. When expected results and actuals match with given degree of variation, one may understand that the plan is going in the right direction. Running faster or slower than the plan will need action. Figure 1.4 shows diagrammatically PDCA cycle of continual improvement.

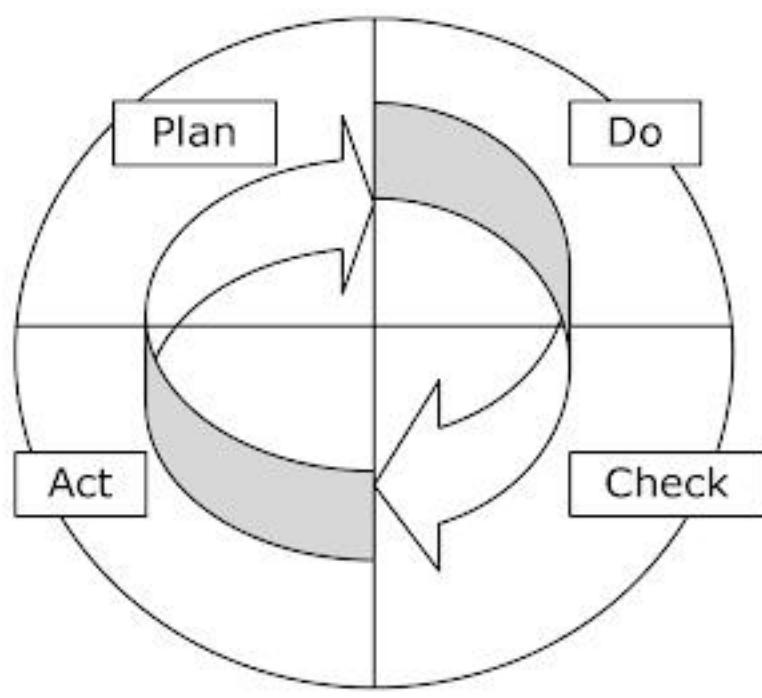


Fig. 1.4

Continual improvement cycle (PDCA cycle)

1.15 QUALITY IN DIFFERENT AREAS

Let us try to understand quality attributes of various products in different areas. Different domains need different quality factors. They may be derived from the customers/users of the domains. Here are few examples of some domains showing customer expectations in terms of quality for various products. These are generic expectations of customers in certain areas and may differ for some individual examples depending upon specific requirements. Definition of quality expectations will vary from instance to instance depending on the domain under consideration, type of product, type of customer, other competitive products, and their features. The table below lists different areas representing different domains and indicates some factors that might be considered related to quality in these areas. Table 1.2 shows some common expectations from customers.

Table 1.2

Products and expected attributes

Product/Service category	Expected attributes
Airlines Industry	On time arrival and departure, comfortable journey, low cost service, reliability and safety.
Health Care Industry	Correct diagnosis and treatment, minimum wait time, lower cost, safety and security.
Food Service Industry	Good product, good taste, fast delivery, good ambience, clean environment.
Consumer Products Industry	Properly made to suite individuals, defect free products, cost effective.
Military Services	Rapid deployment, decreased wages and cost, security.
Automotive Industry	Defect free product, less fuel consumption, more power, safe journey.
Communications Industry	Clear communications, faster access, cheaper service.

There are some common denominators in all these examples which may be considered as the quality factors. Although the terms used to explain each product in different domain areas vary to some extent, almost all areas can be explained in terms of few basic quality parameters. These are defined as the basic quality parameters by stalwarts of quality management.

- Cost of the product and value which the customer finds in it
- Service offered to the customers, in terms of support by the manufacturer
- Time required for the delivery of product
- Customer satisfaction derived from the attributes and functionalities of a product
- Number of defects in the product or frequency of failures faced by users

1.16 BENCHMARKING AND METRICS

Benchmarking is an important concept used in Quality Function Deployment (QFD). It is the concept of creating qualitative/quantitative metrics or measurable variables, which can be used to assess product quality on several scales against a benchmark. Typical variables of benchmarking may include price of a product paid by customer, time required to acquire it, customer satisfaction, defects or failures, attributes and features of product, etc. **Metrics** are defined for collecting information about the product capabilities, process variability and outcome of the process in terms of attributes of products. Metric is a relative measurement of some parameters of a product which are related to the product and processes used to make it. An organisation must develop consistent set of metrics derived from its strategic business plan and performance of benchmark partner.

1.17 PROBLEM SOLVING TECHNIQUES

Improving quality of products and services offered to customers requires methods and techniques of solving problems associated with development and processes used during their lifecycle. An organisation must use metrics approach of process improvement because it needs to make quantitative measurements. These measurements can be used for problem solving using quantitative techniques.

Problem solving can be accomplished by both qualitative and quantitative methods but problem definition becomes easier when we put them against some measures or comparators.

- Qualitative problem solving refers to understanding a problem solution using only qualitative indexes such as high, medium, low, etc. depending on whether something is improving or deteriorating from the present status and so forth. This is a typical scenario for low maturity organisations where the problems are much broader and can be classified in different bands very easily. For initial stages of improvement, qualitative problem solving is sufficient to get faster results. It saves time in defining and measuring data accurately and basic maturity can be achieved.
- Quantitative problem solving requires specification of exact measures in numerical terms such as 'the cost has increased 32.5% during the last quarter' or 'the time required to produce one product unit is reduced by 32 minutes'. For highly matured organisations, quantitative analysis is required for further improvements as basic improvements are already done. It must follow the cycle of Define, Measure, Monitor, Control and Improve. Measurement of processes and products may need good measuring instruments with high level of accuracy and repeatability.

Given quantitative data, one can use statistical techniques to characterise a process. Quantitative methodologies make it possible to analyse and visualise what is actually happening in a process. Process variations can be understood in a better way and actions can be initiated to reduce the variability.

1.18 PROBLEM SOLVING SOFTWARE TOOLS

While buying software for data management and statistical analysis, many organisations find it to be a big investment in terms of money, resources, etc. One must answer the question 'Why should one use software tools to solve problems about quality?' There are some advantages and disadvantages associated with usage of such tools for problem solving.

Advantages of Using Software Tools for Analysis and Decision Making

- Accuracy and speed of the tools is much higher compared to performing all transactions and calculations manually. Calculations can form the basis for making decisions and hence should be as accurate as possible.
- Decision support offered by the tool is independent of personal skills and there is least variation from instance to instance. Tools can support in some fixed range depending upon its logic. Some tools can learn things and use them as required.
- Tools can implement theoretical means of assessing metrics about quality as defined by business law. There is no manual variation.
- Tools alleviate the hard work required to perform hand or calculator driven computations and give more accurate and faster results.
- Tools can be integrated with other systems to provide a systematic and highly integrated means of solving problems

Disadvantages of Using Computer Tools for Analysis and Decision Making

- These programs and tools need training before they can be used. Training incurs cost as well as time. Some tools need specific trainings to understand them and use them.
- All softwares/hardwares are prone for defects and these tools are not exceptions to it. There can be some mistakes while building/using them. Sometimes these mistakes can affect the decisions drastically.
- Decision has to be taken by human being and not by the tool. Tools may define some options which may be used as guide. Some tools can take decision in the limited range.
- Tools may mean more cost and time to learn and implement. Every tool has a learning curve.

1.18.1 TOOLS

Tools are an organisations analytical asset that assist in understanding a problem through data and try to indicate possible solutions. Quality tools are more specific tools which can be applied to solving problems faced by projects and functional teams while improving quality in organisations. Tools may be hardware/software and physical/logical tools. We will learn more about quality tools in Chapter 16 on 'Qualitative and Quantitative Analysis'.

1.18.2 TECHNIQUES

Techniques indicate more about a process used in measurement, analysis and decision making during problem solving. Techniques are independent of tools but they drive tool usage. Techniques do not need tools for application while tools need techniques for their use. Same tool can be used for different purposes, if the techniques are differed. Table 1.3 gives a difference between tools & techniques.

Table 1.3**Difference between tools and techniques**

Tools	Techniques
Usage of tool is guided by the technique. Tool is of no use unless technique (to use it) is available.	Technique is independent of any tool.
Different techniques may use the same tool to achieve different results.	Same technique may use different tools to achieve the same result.
Tool improvement needs technological change.	Technique change can be effected through procedural change.
Contribution of tools in improvement is limited.	Contribution of techniques in improvement is important.



Quality tips

- Try to define quality perspective for the organisation and set of products and projects executed by it.
- Define customer expectations rather than going for system requirement specifications.
- Assess the cost spent by an organisation under various heads of quality. Testers have to play a significant role in reducing cost of quality and improving profitability of the organisation.
- Understand and improve every aspect of an organisation through approaches, techniques and tools to enhance customer satisfaction and goodwill for the organisation. This can help the organisation to prosper in the long term.

Summary



In this chapter, we have seen various definitions of quality as understood by different people and different stakeholders. It also covered the definitions by quality stalwarts and different international standards. Then, we studied the basic components to produce quality, and the views of customers and producer on quality. As a tester, one must understand different gaps like user gap, and producer gap, and how to close them to achieve customer satisfaction.

We have described various cost components like manufacturing cost and cost of quality. Cost of quality concepts with its three components viz. preventive cost of quality, appraisal cost of quality and failure cost of quality are described along with the importance of cost of prevention, and how it affects cost of quality and improves profitability.

We have seen different approaches to continually (continuously) improving quality. We have covered 'TQM principles of quality management', and 'DMMCI principles of continual improvement' through quality planning, quality control and quality improvement. We have also discussed a theory of 'Cultural change principles of quality management'.

Finally we have introduced the concept of problem solving through usage of tools and techniques. We have also briefly elucidated the concept of benchmarking.

- 1) Explain 'quality' in terms of the generic expectations from any product.
- 2) Differentiate between continuous improvement and continual improvement.
- 3) Define the stakeholders for successful projects at micro level and for successful organisations at macro level.
- 4) Define 'quality' as viewed by different stakeholders of software development and usage.
- 5) Explain customers view of quality.
- 6) Explain suppliers view of quality.
- 7) Define 'User's gap' and 'Producer's gap' and explain how these gaps can be closed effectively.
- 8) Describe various definitions of quality as per international standards.
- 9) Describe definition of quality as per Dr Deming, Dr Juran and Philip Crosby.
- 10) Describe 'Total Quality Management' principles of continual improvement.
- 11) Describe cultural change requirement for quality improvement.
- 12) Differentiate between tools and techniques.



CHAPTER

2

SOFTWARE QUALITY



OBJECTIVES

This chapter focusses on software quality parameters. It provides a basic understanding that quality expectation for different products is different and lays a foundation of quality management approach.



2.1 INTRODUCTION

In the previous chapter, we have discussed various definitions of quality and how they fit the perspective of different stakeholders. One of the definitions i.e., '**Conformance to explicitly stated and agreed functional and non-functional requirements**', may be termed as 'quality' for the

software product offered to customers/final users from their perspective. Customers may or may not be the final user and sometimes, developers have to understand requirements of final users in addition to the customer. If the customer is a distributor or retailer or facilitator who is paying for the product directly, then the final user's requirements may be interpreted by the customer to make a right product.

It is not feasible to put all requirements in requirement specifications. A large number of requirements remain undefined or implied as they may be basic requirements for the domain and the customer perceives them as basic things one must know. It gives importance to the documented and approved software and system requirement specifications on which quality of final deliverables can be decided. It shifts the responsibility of making a software specification document and getting it approved from customer to producer of a product. There are many constraints for achieving quality for software application being developed using such software requirement specifications.

2.2 CONSTRAINTS OF SOFTWARE PRODUCT QUALITY ASSESSMENT

Generally, requirement specifications are made by business analysts and system analysts. Testers may or may not have direct access to the customer and may get information through requirement statements, queries answered, etc. either from the customer or business system/analyst, etc. There are few limitations of product quality assessment in this scenario.

- Software is virtual in nature. Software products cannot be seen, touched or heard. Our normal senses and measuring instruments are not capable of measuring quality of software, which is possible in most of the other engineering products.
- There is a huge communication gap between users of software and developers/testers of the product. Generally 5–8 agencies are involved between these two extreme ends. If an average communication loss of 10% at each stage is considered, then huge distortion is expected from user's perspective of requirements and actual product.
- Software is a product which is unique in nature. Similarities between any two products are superficial ones. The finer requirements, designs foundation, architecture, actual code, etc. may be completely different for different products. Way of software design, coding, and reusability may differ significantly from product to product though requirements may look similar at a global level.
- All aspects of software cannot be tested fully as number of permutations and combinations for testing all possibilities tend to infinity. There are numerous possibilities and all of them may not be tried in the entire life cycle of a software product in testing or even in usage. Exhaustive testing is neither feasible nor justifiable with respect to cost.
- A software program executes in the same way every time when it is executing some instruction. An application with a problematic code executes wrongly every time it is executed. It makes a very small defect turn into a major one as the probability of defect occurrence is 100% when that part is executed.

Business analysts and system analysts must consider the following while capturing the requirements of a customer

2.3 CUSTOMER IS A KING

The customer is the most important person in any process of developing a product and using it—software development is not an exception. All software life cycle processes such as development, testing, maintenance, etc. are governed by customer requirements, whether implied or expressed.

An organisation must be dedicated to exceed customer satisfaction with the latter's consent. Exceeding customer satisfaction must not be received with surprise by the customer. He must be informed about anything that has been provided extra and must be in a position to accept it or reject it. Any surprise may be considered as defect.

A satisfied customer is an important achievement for an organisation and is considered as an investment which may pay back in short as well as long term (in terms of references, goodwill, repeat order, etc.). Satisfied customers may give references to others and come back with repeat orders. Customer references are very important for developing new accounts. Organisations should try to implement some of the following measures to achieve customer satisfaction.

2.3.1 FACTORS DETERMINING SUCCESS

To be a successful organisation, one must consider the following factors, entities, and their interactions with each other.

- **Internal Customer and Internal Supplier** ‘Internal customer satisfaction’ philosophy is guided by the principles of ‘Total Quality Management’. When an organisation is grouped into various functions/

departments, then one function/department acts as a supplier or a customer of another function/department. If every function/department identifies its customers and suppliers and tries to fulfill their requirements, in turn, external customer requirements get satisfied.

- **External Customer and External Supplier** External customers may be the final users, purchasers of software, etc. Software requirement specifications are prepared and documented by developing organisations to understand what customer/final user is looking for when he wishes to acquire a particular product. Customers are actually paying for getting their needs served and not for the implementation of the requirements as defined in the specifications document. External suppliers are the entities who are external to the organisation and who are supplying to the organisation.

2.4 QUALITY AND PRODUCTIVITY RELATIONSHIP

Many people feel that better quality of a product can be achieved by more inspection or testing, reworking, scrapping, sorting, etc. More inspection cycles mean finding more defects, fixing defects mean better quality (as it will expose maximum possible defects to be fixed by developers), and ultimately, the customer may get a better product. This directly means that there would be more rework/scrap and it will lead to more cost/price or less profit for such products as more effort and money is spent in improving quality. In such cases, time and effort required would be much higher.

In reality, quality improvement does not talk about product quality only but a process quality used for making such a product. If the processes of development and testing are good, a bad product will not be manufactured in the first place. It will reduce inspection, testing, rework, and cost/price. Thus quality must improve productivity by reducing wastage. It must target for doing 'first-time right.'

- **Improvement in Quality Directly Leads to Improved Productivity** Improved quality does not mean more inspection, testing, sorting and rejection but improving the processes related to product development. All products are the outcome of processes, and good processes must be capable of producing good product at the first instance. This approach reduces frustration, rejection, rework, inspection, and improves quality and customer satisfaction. As this hidden factory producing scrap and wastage stops working, productivity and efficiency improves. Thus, quality products must give more profitability to the supplier and is a cheaper option for the customer.

- **The Hidden Factory Producing Scrap, Rework, Sorting, Repair, and Customer Complaint is Closed** Customer does not intend to pay for scrap, rework, sorting, etc. to get a good product. Engineering industry has faced this problem of unwillingness of customer to pay even for first-time inspection/testing as they represent deficiencies in manufacturing processes. Customer complaints are mostly due to the problems associated with products and aligned services. Problems in products can be linked to faulty development processes. Either the defects are not found in the product during process of development or testing, or fixing of the defects found in these processes introduces some more defects in the product offered.

- **Effective Way to Improve Productivity is to Reduce Scrap and Rework by Improving Processes** Productivity improvement means improving number of good parts produced per unit time and not the parts produced per unit time. It is not hard work but smart and intelligent work which can

help an organisation in improving product quality, productivity and customer satisfaction by reducing re-work, scrap, etc. It necessitates that an organisation must incorporate and improve quality in the processes which lead to better product quality. As wastage of resources reduce with improvements in quality, productivity and efficiency improves as a direct result by improvements in processes.

- **Quality Improvements Lead to Cost Reduction** Quality improves productivity, efficiency and reduces scrap, rework, etc. Improvement in quality increases profit margins for producer by reducing cost of development, cost of quality and reduces sales price for customer. Thus quality implementation must reduce the cost and price without sacrificing profitability.

Employee Involvement in Quality Improvement Employee is the most important part of quality improvement program and crucial element for organisational performance improvement. Management leadership and employee contribution can make an organisation quality conscious while lack of either of the two can create major problems in terms of customer dissatisfaction, loss of profitability, loss of goodwill, etc. Employees are much nearer to problematic processes and know what is going wrong and how it can be improved. Employee involvement in quality implementation is essential as the employees facing problems in their work indicate the process problems. Management must include employees in quality improvement programs at all stages.

- **Proper Communication Between Management and Employee is Essential** Communication is a major problem in today's environment. One of the communication hurdles is that the chances of face-to-face communication are reducing due to technological improvements and distributed teams. Mostly, communication is by virtual appliances like emails, video conferencing, etc. where it is difficult to judge the person through body language. Either there is no communication or there is excessive communication leading to a problematic situation. There are huge losses in communication and distortions leading to miscommunication and wrong interpretation. The reasons for 'Producer's gap' and 'User's gap' are mainly attributed to communication problems. Different words and terms used during the message, tone, type of message, speaking skills, listening skills, etc. contribute to quality of communication.
- **Employees Participate and Contribute in Improvement Process** In quality improvement program design and implementation, employees perform an important part of identification of problems related to processes and giving suggestions for eliminating them. They are the people doing actual work and know what is wrong and what can be improved in those processes to eliminate problems. They must be closely associated with the organisation's goal of achieving customer satisfaction, profitability, name and fame in market, etc. Every employee needs to play a part in implementation of 'Total Quality Management' in respective areas of working. This can improve the processes by reducing any waste.

- **Employee Shares Responsibility for Innovation and Quality Improvement** Management provides support, guidance, leadership, etc. and employees contribute their part to convert organisations into performing teams. Everyday work can be improved significantly by establishing small teams for improvement which contribute to innovations. An organisation must not wait for inventions to happen for getting better products but perform small improvements in the processes everyday to achieve big targets in the long range. The theory of smart work in place of hard work helps employees to identify any type of waste in the process of development and eliminate it.

Table 2.1**Difference between inventions and innovation**

Invention	Innovation
Inventions may be accidental in nature. They are generally unplanned.	Innovation is a planned activity leading to changes.
Invention may or may not be acceptable to people doing the work immediately. Inventions are done by scientist and implementation and acceptance by people can be cumbersome as general level of acceptance is very low.	Innovation is done by people in a team, possibly cross-functional teams, involved in doing a work. There is higher acceptability by people as they are involved in it.
Inventions may not be directly applied to everyday work. It may need heavy customisation to make it suitable for normal usage.	Innovations can be applied to every day work easily. The existing methods and processes are improved to eliminate waste.
Breakthrough changes are possible due to inventions.	Changes in small steps are possible by innovation.
Invention may lead to major changes in technology, way of doing work, etc.	Innovation generally leads to administrative improvements, whereas technological or breakthrough improvements are not possible.
Invention may lead to scraping of old technologies, old skills and sometimes, it meets with heavy resistance.	Innovation may lead to rearrangement of things but there may not be a fundamental change. It generally works on elimination of waste.

Table 2.1 gives a difference between invention & innovation.

Many organisations have a separate 'Research and Development' function responsible for doing inventions. These functions are dedicated to make breakthrough changes by developing new technologies and techniques for accomplishing work. They are supposed to derive major changes in the approaches of development, implementation, testing, etc. 'Six sigma' improvements also talk about breakthrough improvements where processes may be redesigned/redefined. It may add new processes and eliminate old processes, if they are found to be problematic. But, an organisation should also create an environment which helps in innovation or rearrangement of the tasks to make small improvements everyday. Continuously identifying any type of waste in day-to-day activities and removing all nonessential things can refine the processes.

2.5 REQUIREMENTS OF A PRODUCT

Everything done in software development, testing and maintenance is driven by the requirements of a product to satisfy customer needs. There are basic requirements for building software, which will help customers conduct their businesses in a better way. Every product offered to a customer is intended to satisfy some requirements or needs of the customer. Requirements may be put in different categories. Some of these are,

- **Stated/Implied Requirements** Some requirements are specifically documented in software requirement specifications while few others are implied ones. When we build software, there are functional and non-functional requirements specified by a customer and/or business/system analyst. It is also intended not to violate some generally accepted requirements such as 'No spelling mistakes in user interfaces', 'Captions on the control must be readable', etc. These type of requirements may not be documented as

a part of requirement statement formally but generally considered as requirement, and are expected in a product. As a part of development team/test team, one must understand stated as well as intended or implied requirements from the users. It may be a responsibility of a developing organisation to convert as many implied requirements as possible into stated requirements. Though impossible, the target may be 100%.

- **General/Specific Requirements** Some requirements are generic in nature, which are generally accepted for a type of product and for a group of users while some others are very specific for the product under development. Addition of two numbers should be correct is a generic requirement while the accuracy of 8 digits after decimal and rounding may be a very specific requirement for the application under development. Usability is a generic requirement in software for the intended user while authentication and messaging to users may be driven by specific requirements of an application. Many times, the generic requirements are considered as implied ones and those may not be mentioned in requirement specifications while specific requirements are stated ones as those are present in requirement specifications.

- **Present/Future Requirements** Present requirements are essential when an application is used in present circumstances while future requirements are for future needs which may be required after some time span. For projects, present as well as future requirements may be specifically defined by a customer or business/system analyst. A product development organisation may have to do further research to identify or extrapolate future needs of users. For banking software, today's requirements may be 5000 saving accounts, and the application may be running in client-server environment. But a future need may be 50 lakh saving accounts and the application may be running as a web application. 'How much future?' must be guided by the customer's vision as this may influence product cost. Definition of future has direct relationship with usable life of an application. Some people may use a software for 3 years while some other may be planning to use it for 30 years. The future requirements may change as per the expected life span of the software.

On the basis of priority of implementation from user's perspective, requirements may be categorised in different ways as follows:

- **'Must' and 'Must not' Requirements or Primary Requirements** 'Must' requirements are primary requirements for which the customer is going to pay for while acquiring a product. These are essential requirements and the value of the product is defined on the basis of the accomplishment of 'must' requirements. Generally these requirements have the highest priority in implementation. Not meeting these requirements can cause customer dissatisfaction and rejection of a product. These requirements may be denoted by priority 'P1' indicating the highest priority. It also covers 'Must not' requirements which must be absent in the product.

- **'Should be' and 'Should not be' Requirements or Secondary Requirements** 'Should be' requirements are the requirements which may be appreciated by the customer if they are present/absent and may add some value to the product. Customer may pay little bit extra for the satisfaction of these requirements but price of the product is not governed by them. Generally, these requirements are lower priority requirements than 'Must' requirements. These requirements may give the customer delight, if present and little disappointment, if absent. These requirements may be denoted by priority 'P2'. It also covers 'Should not' requirements.

- **'Could be' and 'Could not be' Requirements or Tertiary Requirements** 'Could be' requirements are requirements which may add a competitive advantage to the product but may not add much value in terms of price paid by a customer. If two products have everything same, then 'could be' requirements may help in better appreciation of a product by the users. These are the lowest priority requirements. It also covers 'Could not' requirements. These requirements give a product identity in the market. These requirements may be denoted by priority 'P3'.

While talking about a product in view of a bigger market with large number of generic users, it may be very difficult to categorise the requirements as mentioned above. This is because 'must' requirement for one customer may be 'could be' requirement for somebody else. In such cases, an organisation may have to target the customer segment and define the priorities of the requirements accordingly. Customer must have the final authority to define the category of requirement.

2.6 ORGANISATION CULTURE

An organisation has a culture based on its philosophy for existence, management perception and employee involvement in defining future. Quality improvement programs are based on the ability of the organisation to bring about a change in culture. Philip Crosby has prescribed quality improvement for cultural change. Quality culture of an organisation is an understanding of the organisation's virtue about its people, customer, suppliers and all stakeholders. 'Q' organisations are more quality conscious organisations, while 'q' organisations are less quality conscious organisations. The difference between 'Q' organisations and 'q' organisation is enumerated as follows. Table 2.2 shows a difference between quality culture of 'Q' & 'q' organisations.

Table 2.2

Difference between 'Q' organisation and 'q' organisation

Quality culture is 'Q'	Quality culture is not 'q'
These organisations believe in listening to customers and determining their requirements.	These organisations assume that they know customer requirements.
These organisations concentrate on identifying cost of quality and focusing on it to reduce cost of failure which would reduce overall cost and price.	These organisations overlook cost of poor quality and hidden factory effect. They believe in more testing to improve product quality.
Doing things right for the first time and every time is the motto of success.	Doing things again and again to make them right is their way of working. Inspection, rework, scrap, etc. are essential.
They concentrate on continuous/continual process improvement to eliminate waste and get better output.	They work on the basis of finding and fixing the problem as and when it is found. Onetime fix for each problem after it occurs.
These organisations believe in taking ownership of processes and defects at all levels.	These organisations try to assign responsibility of defects to someone else.
They demonstrate leadership and commitment to quality and customer satisfaction.	They believe in assigning responsibility for quality to others.

2.6.1 SHIFT IN FOCUS FROM 'q' TO 'Q'

As the organisation grows from 'q' to 'Q', there is a cultural change in attitude of the management and employees towards quality and customer. In initial stages, at the level of higher side of 'q', a product is subjected to heavy inspection, rework, sorting, scrapping, etc. to ensure that no defects are present in final deliverable to the customer while the final stages of 'Q' organisation concentrate on defect prevention through

process improvements. It targets for first-time right. Figure 2.1 shows an improvement process where focus of quality changes gradually.



- **Quality Control Approach (Finding and Fixing Defects)** Quality control approach is the oldest approach in engineering when a product was subjected to rigorous inspection for finding and fixing defects to improve it. Organisations at the higher end of 'q' believe in finding and fixing defects to the extent possible as the way to improve quality of product before delivering it to customer and achieving customer satisfaction. It basically works on correction attitude involving defect fixing, scrap, rework, segregation, etc. It works

on the philosophy that a product is good unless a defect is found in it. There are huge testing teams, large investment in appraisal cost and defect fixing costs followed by retesting and regression testing.

- **Quality Assurance Approach (Creation of Process Framework)** Quality assurance is the next stage of improvement from quality control where the focus shifts from testing and fixing the defects to first-time right. An Organisation does investment in defining processes, policies, methods for handling various functions so that it can incorporate a process approach for doing various things. It becomes a learning organisation as it shifts its approach from 'quality control' to 'quality assurance'. The management approach shifts from corrections to corrective actions through root cause analysis. There are some actions on the basis of metrics program instituted by the organisation. It also starts working on preventive actions to some extent to avoid potential defects. Defects are considered as failures of processes and not of people, and actions are initiated to optimise the processes.

- **Quality Management Approach** There are three kinds of system in the universe, viz. completely closed systems, completely open systems and systems with semipermeable boundaries. Completely closed systems represent that nothing can enter inside the system and nothing can go out of the system. On the other hand, open system represents a direct influence of universe on system and vice-a-versa. Completely closed systems or completely open systems do not exist in reality. Systems with semipermeable boundaries are the realities, which allow the system to get impacted from external changes and also have some effect on external environment. Anything coming from outside may have an impact on the organisation but the level of impact may be controlled by taking some actions. Similarly anything going out can also affect the universe but impact is controlled.

Organisations try to assess the impact of the changes on the system and try to adapt to the changes in the environment to get the benefits. They are highly matured when they implement Quality Management as a management approach. There are virtually no defects in processes as they are optimised continuously, and products are delivered to customers without any deficiency. The organisation starts working on defect prevention mechanism and continuous improvement plans. The organisation defines methods, processes and techniques for future technologies and training programs for process improvements.

Management includes planning, organising, staffing, directing, coordinating, and controlling to get the desired output. It also involves mentoring, coaching, and guiding people to do better work to achieve organisational objectives.

2.7 CHARACTERISTICS OF SOFTWARE

There are many products available in the market which are intended to satisfy same or similar demands. There is a vast difference between software products and other products due to their nature.

- Software cannot be sensed by common methods of inspection or testing, as it is virtual in nature. The product is in the form of executable which cannot be checked by any natural method available to mankind like touch, smell, hearing, taste, etc. It cannot be measured by some measuring instruments commonly available like weighing balance, scales, etc. It needs testing in real environment but nobody can do exhaustive testing by trying all permutations and combinations.
- There are different kinds of software products and their performance, capabilities, etc. vary considerably from each other. There are no same products though there may be several similar ones or satisfying similar needs. Every product is different in characteristics, performance, etc. Software is always unique in nature.
- Every condition defined by the software program gets executed in the same way every time when it gets executed. But the number of conditions, and algorithm combinations may be very large tending to infinity and testing of all permutations/combinations is practically impossible.

2.8 SOFTWARE DEVELOPMENT PROCESS

Software development process defines how the software is being built. Some people also refer to SDLC as system development life cycle with a view that system is made of several components and software is one of these components. There are various approaches to build software. Every approach has some positive and some negative points. Let us talk about few basic approaches of developing software from requirements. It is also possible that different people may call the same or similar approach by different names.

- Waterfall development approach/model
- Iterative development approach/model
- Incremental development approach/model
- Spiral development approach/model
- Prototyping development approach/model
- Rapid application development approach/model
- Agile development approach/model

2.8.1 WATERFALL DEVELOPMENT APPROACH/MODEL

Waterfall model is the simplest software development model and is used extensively in development process study. There are many offshoots of waterfall model such as modified waterfall model, iterative waterfall model, etc. Though it is highly desirable to use waterfall model, it may not be always feasible to work with it. Still, waterfall model remains as a primary focus for study purpose. It is also termed as classical view of software development as it remains the basis or foundation of any development activity. Most of the other models of development are based upon the basic waterfall model as it represents a logical way of doing things.

Typical waterfall model is shown in Fig. 2.2.

Arrows in the waterfall model are unidirectional. It assumes that the developers shall get all requirements from a customer in a single go. The requirements are converted into high level as well as low level designs. Designs are implemented through coding. Code is integrated and executables are created. Executables are

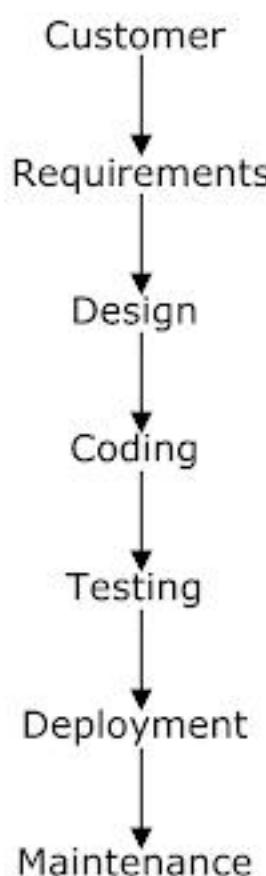


Fig. 2.2

Waterfall model

tested as per test plan. The final output in the form of an executable is deployed at customer premises. Future activities are handled through maintenance. If followed in reality, waterfall model is the shortest route model which can give highest efficiency, and productivity.

Waterfall models are used extensively in fixed price/fixed schedule projects where estimation is based on initial requirements. As the requirement changes, estimation is also revised.

Limitations of Waterfall Cycle There is no feedback loop available in waterfall model. It is assumed that requirements are stable and no problem is encountered during entire development life cycle. Also, no rework is involved in waterfall model.

2.8.2 ITERATIVE DEVELOPMENT APPROACH/MODEL

Iterative development process is more practical than the waterfall model. It does not assume that the customer gives all requirements in one go and there is complete stability of requirements. It assumes that changes may come from any phase of development to any previous phase and there are multiple permutations and combinations of changes.

Changes may have a cascading effect where one change may initiate a chain reaction of changes. Figure 2.3 shows a feedback loop which is the fundamental difference between waterfall model and iterative development model.

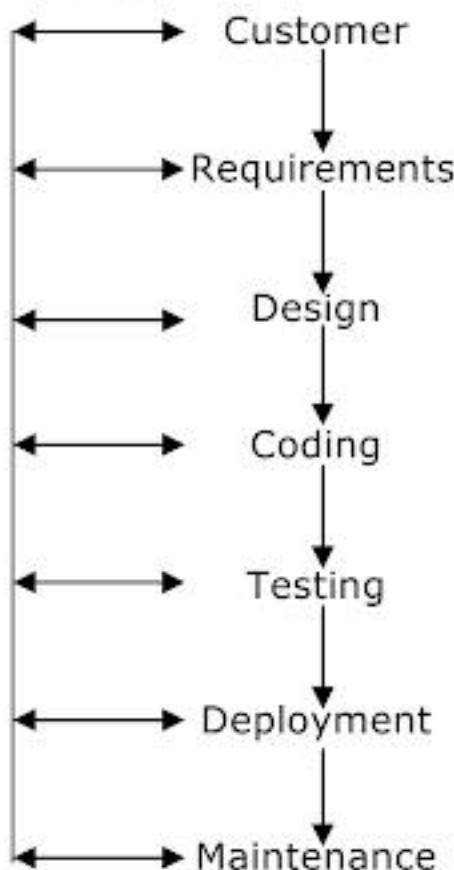


Fig. 2.3

Iterative development model

Limitations of Iterative Development Iterative development consists of many cycles of waterfall model. It gives problems in fixed price projects for estimation. Another problem faced by iterative development is that the product architecture and design becomes fragile due to many iterative changes.

2.8.3 INCREMENTAL DEVELOPMENT APPROACH/MODEL

Incremental development models are used in developing huge systems. These systems are made of several subsystems which in themselves are individual systems. Thus, incremental systems may be considered as a collection of several subsystems.

An individual subsystem may be developed by following waterfall methodology and iterative development. These subsystems may be connected to each other externally, either directly or indirectly. A directly interconnected system allows the subsystems

to talk with each other while indirectly interconnected system has some interconnecting application between two subsystems. Direct connectivity makes a system more robust but flexibility can be a major issue.

The incremental model gives flexibility to a customer. One system may be created and the customer may start using it. The customer can learn the lessons and use them while second part of the system is developed. Once those are integrated, third part may be developed and so on. The customer does not have to give all requirements at the start of development phase.

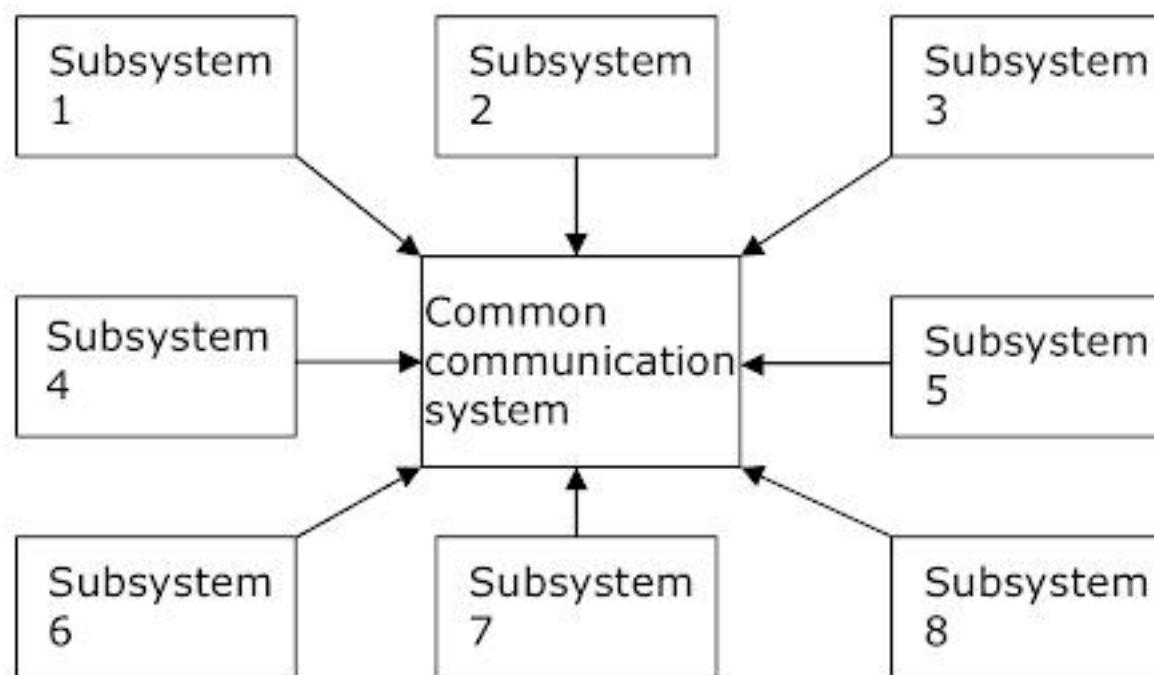


Fig. 2.4

Incremental model

The model in Fig. 2.4 shows a common communication system and incremental subsystems where different subsystems communicate through a common communication system.

Limitations of Incremental Development Incremental models with multivendor product integration are a major challenge as parameter passing between different systems may be difficult. Incremental models help in integration of big systems at the cost of loss of flexibility. When a system is incremented with new subsystems, it changes the architecture of that system. Increment in the system is followed by heavy regression testing to find that when multiple systems come together, can they work individually as well as collectively.

2.8.4 SPIRAL DEVELOPMENT APPROACH/MODEL

Spiral development process assumes that customer requirements are obtained in multiple iterations, and development also works in iterations. Many big software systems are built by spiral models of ever-increasing size. First some functionalities are added, then product is created and released to customer. After getting the benefits of first iteration of implementation, the customer may add another chunk of requirements to the existing one. Further addition of requirements increase the size of the software spirally. Sometimes, an individual part developed in stages represents a complete system, and it may communicate with the next developed system through some interfaces.

In many ERPs, initial development concentrated around material management part which later increased spirally to other parts such as purchasing, manufacturing, sales, warehousing, cash control, etc. Many banking softwares also followed a similar route. Figure 2.5 shows a spiral development model.

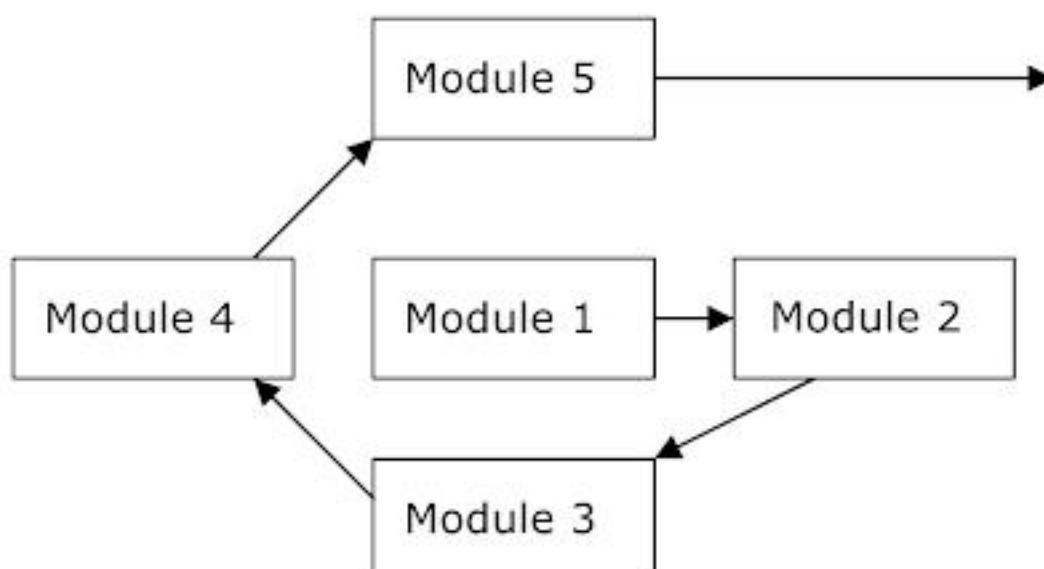


Fig. 2.5

Spiral development model

Spiral development is considered as a miniature form of the incremental model.

Limitations of Spiral Development Spiral models represent requirement elicitation as the software is being developed. Sometimes, it may lead to refactoring and changes in approach where initial structures become non-useable. Spiral development also needs huge regression testing cycles to find whether additions in the given system have affected overall system working or not.

2.8.5 PROTOTYPE DEVELOPMENT APPROACH/MODEL

Prototype development approach represents top to bottom reverse integration approach. Major problem of software development is procuring and understanding the customer requirements for the product. Prototyping is one of the solutions to help in this problem.

In prototyping, initially a prototype of the system is created—this is similar to cardboard model of a building. It helps the customer to understand what they can expect from the given set of requirements. It also helps the development team to understand the possible application's look and feel. Once the elicitation is done, the logic is built behind it to implement the elicited requirements.

Limitations of Prototype Development Though, one may get a feel of the system by looking at the prototype, one must understand that it is not the actual system but a model. The customer may get the feeling that the system is already ready and may pressurise development team to deliver it immediately. (Applications not having much graphical user interfaces are difficult to model.)

2.8.6 RAPID APPLICATION DEVELOPMENT APPROACH/MODEL

Rapid application development is not a rapid way of developing software as one may interpret from the name. It is one way to create usable software at a fast speed, and still give an opportunity to the user to understand the development and application being created.

It is a miniature form of spiral development. Development team may get very less number of requirements (let us say 5/6). They create a design, code it, test it and release it to customer. Once customer gets the delivery, he may have a better understanding of his expectations and development process by looking at

the product delivered. He may add another chunk of requirements and entire development cycle is followed. Thus, each iteration will give better understanding about a product being developed and may help in refining the requirements.

Limitations of Rapid Application Development Change in approach and refactoring are the major constraints in rapid application development. It also involves huge cycles of retesting and regression testing. Efforts of integration are huge.

2.8.7 AGILE DEVELOPMENT APPROACH/MODEL

Agile development methodologies are becoming popular due to their dynamic nature and easy adaptability to the situation. One of the surveys indicated that in case of waterfall model, many functionalities are added in requirement statement with a fear that changes in scope would not be appreciated by the development team. Some surveys show that many of the functionalities (about $\frac{3}{4}$ th) developed using waterfall or iterative model are never used by the users. Agile gives complete freedom to the user to add requirements at any stage of development, and development team has to accept these changes. Agile methodologies work on small chunk of work in each iteration and release working software at the end of iteration. The main thrust of Agile methodologies is complete adaptability to user environment and continuous integration of a product. It also gives importance to delivering working software rather than achieving requirements defined in requirement specifications. Agile represents a family of development methodologies and there are many methodologies under its umbrella. Some of them are as listed below.

- Scrum
- Extreme Programming
- Feature Driven Development
- Test Driven Development

Agile works on the following principles,

- Individuals and interactions are more important than formal sign-offs for requirements, designs, etc. It concentrates more on 'Fitness for use' and what the customer needs are.
- Working software is the outcome of each milestone rather than concentrating on deliverables as defined in the project plans. Success of software product is that it is working at each stage.
- Customer collaboration is required to get usable software rather than signing various documents for approvals. Requirement clarifications, requirement elicitation, and prototyping need customer involvement.
- Responding to changes required by the customer at any moment. There may be many changes suggested by the customer as he has better knowledge about what his business needs are.

2.8.8 MAINTENANCE DEVELOPMENT APPROACH/MODEL

Major cost of the software is in its maintenance phase. Every product including software has many defects which may create problems to its users in the long term. Every technology has a life span. New technologies may offer better services and options, and may replace existing technologies. Every now and then, technological updations are required for the software as well as system to perform better and in the most cost effective way. New functionalities may be required due to changing business needs. Maintenance activities of software may be put under 4 different groups namely,

- **Bug fixing** where the defects present in the given software are fixed. This may involve retesting and regression testing. During bug fixing, analysis of bug is an important consideration. There is always a possibility that while fixing a bug, new bugs may have been added in the product.
- **Enhancement** where new functionalities are added in the existing software. These functionalities may be required due to changes in the way business is done. Some functionalities may be introduced due to changes in user requirements.
- **Porting** where software is taken from older technologies to newer technologies. In porting, one is expected to port the functionalities and not the code. Whatever functionalities are available in the old technologies, all those are expected to be present in the new technology.
- **Reengineering** where there is some change in the logic or algorithm used due to changes in business environment.

2.9 TYPES OF PRODUCTS

Similar to software development methodologies, software products have some peculiarities defined as criticalities of software. Criticality of the software defines, how much important it is to a user/customer.

There are various schemes of grouping the products on the basis of criticality to the users. Few of them are listed below,

2.9.1 LIFE AFFECTING PRODUCTS

Products which directly/indirectly affect human life are considered as the most critical products in the world from user's perspective. Such products generally come under the purview of regulatory and safety requirements, in addition to normal customer requirements. The quality requirements are more stringent, and testing is very critical for such products as failures may result into loss of life or disablement of a user. This type of product may be further grouped into 5 different categories.

- Any product failure resulting into death of a person. These will be the most critical products as they can affect human life directly.
- Any product failure which may cause permanent disablement to a patient. These are second-level criticality products.
- Any product failure which may cause temporary disablement to a patient
- Any product failure which may cause minor injury and does not result into anything as defined above
- Other products which do not affect health or safety directly

Such software needs huge testing to try each and every conceivable fault in the product. It talks about very high level of confidence that application will not fail under normal and abnormal situations.

2.9.2 PRODUCT AFFECTING HUGE SUM OF MONEY

A product which has direct relationship with loss of huge sum of money is second in the list of criticality of the product. Such products may need large testing efforts and have many regulatory as well as statutory requirements. eCommerce and eBusiness softwares may be put in this category. Security, confidentiality, and accuracy are some of the important quality factors for such products.

These products also need very high confidence level and huge testing will represent the criticality. They need testing lesser than the products which directly/indirectly affect human life.

2.9.3 PRODUCTS WHICH CAN BE TESTED ONLY BY SIMULATORS

Products which cannot be tested in real-life scenario but need simulated environment for testing are third in the ranking of criticality. In this case, real life scenario is either impossible to create or may not be economically viable. Products used in aeronautics, space research, etc. may be put in this category.

Such products also need huge testing, although lesser than the earlier two types.

2.9.4 OTHER PRODUCTS

All other products which cannot be categorised in any of the above schemes may be put in this category.

Unfortunately, 'criticality' is not very easy to define. Let us consider an example of auto piloting software where we have three combinations of criticality together. It does affect the life of passengers traveling, cost of an aeroplane is huge and it cannot be tested in real environment. Thus, all the three criticalities coming together make a product most critical.

2.10 SOME OTHER SCHEMES OF CRITICALITY DEFINITIONS

There may be several other ways of classifying criticality of a product. It has direct relationship with business dependability and the extent of loss to the user organisation/person in case of failure.

2.10.1 FROM USER'S PERSPECTIVE

This classification mainly discusses dependency of a business on a system. The criticality may range from complete dependency to no/minimal dependency on the system.

- Product's failure which disrupts the entire business can be very critical from business point of view. There is no fallback arrangement available or possible in case of failure of a product which is completely dependent on the system.
- Product's failure which affects business partially as there may be some fallback arrangements is a type of criticality. Business may be affected temporarily, affecting profitability or service level but can be restored with some efforts.
- Product's failure which does not affect business at all is one of the options. If it fails, one may have another method to achieve the same result. Rearrangement may not have significant distortion of business process.

2.10.2 ANOTHER WAY OF DEFINING USER'S PERSPECTIVE

This classification considers the environment in which the product is operating. It may range from very complex user environment to very easy user environment.

- Products where user environment is very complex such as aeronautics, space research, etc. may be considered as very critical. Any failure of product can result into major problems as the environment where these products are working is not an easy environment. As the environment is very complex, system is already under stress, and failure of a product may add to the situation.
- Products where user environment is comparatively less complex (such as banks) may represent the second stage of complexity. Huge calculations may be affected, if the system collapses but there may be workaround available. People may find it inconvenient, but still the operations can be performed with some tolerable level of problems. For example, banks were running for so many years without computers and if centralised system fails, they may be able to withstand the pressure to some extent.

- Products where user environment is very simple, and product failure may not add to the consequences represents the lowest level of complexity. If there is any failure, it can be restored quite fast or some other arrangements can be used, and work can be continued.

2.10.3 CRITICALITY FROM DEVELOPER'S PERSPECTIVE

This classification defines the complexity of the system on the basis of development capabilities required. It may range from very complex systems to very simple systems.

- Form based software where user inputs are taken and stored in some database. As and when required, those inputs are manipulated and shown to a user on screen or in form of a report. There is not much manipulation of data and no heavy calculations/algorithms are involved.
- Algorithm based software, where huge calculations are involved and decisions are taken or prompted by the system on the basis of outcome of these calculations. Due to usage of different mathematical models, the system becomes complex and designing, developing and testing all combinations become problematic.
- Artificial intelligent systems which learn things and use them as per circumstances are very complex systems. An important consideration is that the 'learnings' acquired by the system must be stored and used when required. This makes the system very complicated.

There may be a possibility of combination of criticalities to various extends for different products at a time. A software used in aviation can affect human life, and also huge money at a time. It may not be feasible to test it in real-life scenario. It may involve some extent of artificial intelligence where system is expected to learn and use those 'learnings'. Thus, the combination increases severity of failure further.

2.11 PROBLEMATIC AREAS OF SOFTWARE DEVELOPMENT LIFE CYCLE

Let us discuss some problematic areas of software development life cycle.

2.11.1 PROBLEMS WITH REQUIREMENT PHASE

Requirement gathering and elicitation is the most important phase in software development life cycle. Many surveys indicate that the requirement phase introduces maximum defects in the product. Problems associated with requirement gathering are,

Requirements Are Not Easily Communicated Communication is a major problem in requirement statement creation, software development and implementation. Communication of requirements from customer to development team is marked by problems of listening to customer, understanding business domain, and usage of language including domain specific terms and terminologies. The types of requirements are,

Technical Requirements Technical requirements talk about platform, language, database, operating system, etc. required for the application to work. Many times, the customer may not understand the benefits of selecting a specific technology over the other options and problems of using these technically specified configurations. Selection of technology may be done as directed by the development team or as a fashion. Development organisation is mainly responsible for definition of technical requirements for software product under development on the basis of product usage. (Technical requirements also cover this type of system, whether a stand alone or client server or web application etc, tiers present in the system, processing options such as online, batch processing etc). It also talks about configuration of machines, routers, printers, operating systems, databases, etc.

Economical Requirements Economics of software system is dependent on its technical and system requirements. The technical as well as system requirements may be governed by the money that the customer is ready to put in software development, implementation and use. It is governed by cost-benefit analysis. These requirements are defined by development team along with the customer. The customer must understand the benefits and problems associated with different approaches, and select the approach on the basis of some decision-analysis process. The consequences of any specific selection may be a responsibility of the customer but development organisations must share their knowledge and experience to help and support the customer in making such a selection.

Legal Requirements There are many statutory and regulatory requirements for software product usage. For any software application, there may be some rules and regulations by government, regulatory bodies, etc. applicable to the business. There may be some rules which keep on changing as per decisions made by government, regulatory authorities, and statutory authorities from time to time. There may be numerous business rules which are defined by customers or users for doing business. Development team must understand the rules and regulations applicable for a particular product and business.

Operational Requirements Mostly operational requirements are defined by customers or users on the basis of business needs. These may be functional as well as non-functional requirements. They tell the development team, what the intended software must do/must not do when used by the normal user. Operational requirements are derived from the business requirements. This may include non-functional requirements like security, performance, user interface, etc.

System Requirements System requirements including physical/logical security requirements are defined by a customer with the help of a development team. These include requirements for hardware, machine configurations, types of backup, restoration, physical access control, etc. These requirements are defined by customer's management and it affects economics of the system. There may be some specific security requirements such as strong password, encryption, and privilege definitions, which are also declared by the customer.

Requirements Change Very Frequently Requirements are very dynamic in nature. There are many complaints by development teams that requirement change is very frequent. Many times, development teams get confused because customer requirements change continuously. '**Customer does not know what he wants**' is a very common complaint made by development teams. As the product is being built and shown to customer, lot of new ideas are suggested. Some ideas may have significant effect on cost, schedule, and effort while some other may change the architecture, basic approach, and design of software. The time gap between requirement definition and actual product delivery also plays a major role in changing requirements. Top-down approach, rapid application development, and joint application development are some of the techniques used to develop applications by accommodating changes suggested by customers.

2.11.2 GENERALLY A UNIQUE PRODUCT IS DEVELOPED EACH TIME

No two things in the world are same, though they might appear to be similar. In case of software, no two applications are same. Even in case of simple porting (desktop to client-server to web application), software application changes significantly. The same implementation done by two different developers may differ from each other. Even the same program written by the same developer at two different instances may not match

exactly. Thus a software produced may be unique for that instance. One more fact about software product maintenance is that designers find it difficult to understand original design or approach and developers find it difficult to read the code written earlier.

2.11.3 INTANGIBLE NATURE OF PRODUCT, INTELLECTUAL APPROACH THROUGHOUT DEVELOPMENT

Software products cannot be felt by normal senses. Its existence can be felt only by disc space it occupies. There are multiple options or approaches (for example, in architecture or design) possible for implementation of the same set of requirements. Some may feel that one approach is better than the other for different reasons. The capabilities of individuals and organisations vary significantly in design and development, and each may have a good justification why a certain approach is selected with respect to other.

2.11.4 INSPECTION CAN BE EXHAUSTIVE/IMPOSSIBLE

While defining exhaustive inspection, one may tend to include infinite permutations and combinations of testing. Testing of complete software product is practically impossible. It may need huge money and long time to test all possibilities, and still one may not be sure that everything is covered in testing. Testing uses a sampling theory to find the defects in the product and processes used. Testing tries to find out the lacunae in development methodology and processes used.

2.11.5 EFFECT OF BAD QUALITY IS NOT KNOWN IMMEDIATELY

Any level of exhaustive testing is not capable of testing each and every algorithm, branch, condition and combination thoroughly. There are some areas which remain untested even after the application is used over extended periods. Any problem in such areas may be discovered only when the particular situation arises. The effect of this kind of problem and situation during usage may not be known beforehand while deploying the software in use.

2.11.6 QUALITY IS INBUILT IN PRODUCT

Quality of a software product cannot be improved by testing it again and again and finding and fixing the defects. It needs to be built in the product while development using good processes and methods. Any amount of testing cannot certify that a product is defect-free. Good processes and procedures can make good software. No software can be considered as defect-free even if no defect is found in the test iteration defined for it. We can only say that no defect has been discovered till that point of time using those many test cases.

2.11.7 QUALITY OBJECTIVES VARY FROM PRODUCT TO PRODUCT/ CUSTOMER TO CUSTOMER

Quality objectives define the expectations of customer/user and the acceptance level of various parameters which must be present in a given product for accepting/using it. Quality objectives are product dependent, time dependent and are mainly driven by customers or final users. There may be a possibility of trade-off between these factors. Some people define them as test objectives as they define the priority of testing. In a small computer game for kids, cost may be more important than accuracy. On the contrary, applications developed for aeronautics need to be more accurate while cost factor may not be that important. Quality objectives are defined on the basis of factors of quality which are 'must', 'should be', and 'could be' for the

application. Degree of importance changes from product to product, customer to customer, and situation to situation. Some of the quality factors are listed below.

Compliance Every system is designed in accordance with organisational and user policies, procedures and standards. If software meets these standards, it is said to be complying with the specifications. In addition to customer defined standards, there may be few standards for different domains like aviation, medical, and automotive. Software must follow these standards when it is used by particular type of people or for a particular domain. Some regulations and laws may be imposed by the regulatory and statutory bodies.

Generally, these requirements are categorised as legal requirements. These requirements may be put in non-functional requirements.

Correctness Data entered, processed and results obtained must be accurate as per requirements of customers and/or users. Definition of correctness may change from customer to customer, application to application, and time to time. Generally, accuracy refers to mathematical accuracy, but it is not a rule. For a shopkeeper, accuracy of 0.01 may be sufficient as nothing below 1 paisa is calculated while a scientist may need an accuracy of 256 digits after decimal point as rounding off errors can cause a major problem in research work.

Ease of Use Efforts required to learn, operate, prepare input for and interpret output from the system define ease of use for an application. The normal users who are expected to use the software must be comfortable while using it. Ease of use reduces training cost for the new users dramatically. If a software application can be learned without any external help, such software is considered as the best from this perspective. If there is a requirement of training or hand holding before the software can be used, people may find it inconvenient.

Ease of use is a very important factor when large number of users are expected (for example, emailing software and mobile phones), and providing them training is a difficult task.

Maintainability Efforts required to locate and fix errors in an operational system must be as less as possible to improve its ability for maintenance. There may be some possibilities of enhancements and reengineering where good maintainable software has least cost associated with such activities. Software may need maintenance activity some time or the other to improve its current level of working. Ability of software to facilitate maintenance is termed as maintainability. Good documentation, well commented code, and requirement traceability matrix improve maintainability of a product.

Portability Efforts required in transferring software from one hardware configuration and/or software system environment to another environment defines portability of a system. It may be essential to install same software in different environments and configurations as per business needs. If the efforts required are less, then the system may be considered as highly portable. On the other hand, if the system cannot be put in a different environment, it may limit the market.

Coupling Coupling talks about the efforts required in interconnecting components within an application and interconnection of system as a whole with all other applications in a production environment. Software may have to communicate with operating system, database, and other applications when a common user is working with it. Good coupling ensures better use of environment and good communication, while bad coupling creates limitation on software usage.

Performance Amount of resources required to perform stated functions define the performance of a system. For better and faster performance requirements, more and more system resources and/or optimised

design may be required. Better performance improves customer satisfaction through better experience for users. Performance attribute may cover performance, stress and volume. Details about these will be discussed in the latter chapters of this book.

Ease of Operations Effort required in integrating the system into operating environment may define ease of operations. Ease of operations also talks about the help available to a user for doing any operation on the system (for example, online help, user manuals, operations manual). ‘Ease of operations’ is different from ‘Ease of use’ where the former considers user experience while using the system, while the latter considers how fast the system working knowledge can be acquired.

Reliability Reliability means that the system will perform its intended functions correctly over an extended time. Consistent results are produced again and again, and data losses are as less as possible in a reliable system. Reliability testing may be a base of ‘Build Verification Testing (BVT)’ where test manager tries to analyse whether system generates consistent results again and again.

Authorisation The data is processed in accordance with the intents of the user management. The authorisation may be required for highly secured processes which deal with huge sum of money or which have classified information. Applications dealing with classified information may need authorisation as the sensitivity of information is very important.

File Integrity Integrity means that data will remain unaltered in the system and whatever goes inside the system will be reproduced back in the same way. Accepting data in correct format, storing it in the same way, processing it in a way so that data does not get altered and reproducing it again and again are covered under file integrity. Data communication within and from one system to another may also be considered under the scope of file integrity.

Audit Trail Audit trail talks about the capability of software to substantiate the processing that has occurred. Retention of evidential information about the activities done by users for further reference may be maintained.

Continuity of Processing Availability of necessary procedures, methods and backup information to recover operations, system, data, etc. when integrity of the system is lost due to problems in the system or the environment define continuity of processing. Timeliness of recovery operations must be defined by the customer and implemented by developing organisations.

Service Levels Service levels mean that the desired results must be available within the time frame acceptable to the user, accuracy of the information must be reliable, and processing completeness must be achieved. For some applications in eBusiness, service level definition may be a legal requirement. Service level may have direct relationship with performance.

Access Control The application system resources will be protected against accidental or intentional modification, destruction, misuse or disclosure by authorised as well as unauthorised people. Access control generally talks about logical access control as well as physical access control for information, assets, etc.

2.12 SOFTWARE QUALITY MANAGEMENT

Quality management approaches talk about managing quality of a product or service using systematic ways and methods of development and maintenance. It is much above achieving quality factors as defined in software requirement specifications. Quality management involves management of all inputs and processing to the processes defined so that the output from the process is as per defined quality criteria. It talks about three levels of handling problems, namely,

Correction Correction talks about the condition where defects found in the product or service are immediately sorted and fixed. This is a natural phenomenon which occurs when a tester defines any problem found during testing. Many organisations stop at fixing the defect though it may be defined as corrective action by them. Responsibility of finding and fixing defects may be given to a line function. This is mainly a quality control approach.

Corrective Actions Every defect needs an analysis to find the root causes for introduction of a defect in the system. Situation where the root cause analysis of the defects is done and actions are initiated to remove the root causes so that the same defect does not recur in future is termed as corrective action. Corrective action identification and implementation is a responsibility of operations management group. Generally, project leads are given the responsibilities of initiating corrective actions. This is a quality assurance approach where process-related problems are found and resolved to avoid recurrence of similar problems again and again.

Preventive Actions On the basis of root causes of the problems, other potential weak areas are identified. Preventive action means that there are potential weak areas where defect has not been found till that point, but there exists a probability of finding the defect. In this situation, similar scenarios are checked and actions are initiated so that other potential defects can be eliminated before they occur. Generally identification and initiation of preventive actions are a responsibility of senior management. Project managers are responsible for initiating preventive actions for the projects. This is a quality management approach where an organisation takes preventive action so that there is no defect in the first place.

Quality management is a set of planned and systematic activities which ensures that the software processes and products conform to requirements, standards and processes defined by management, customer, and regulatory authorities. The output of the process must match the expectations of the users.

2.13 WHY SOFTWARE HAS DEFECTS?

One very important question about a product is, ‘Why there are defects in the product at all?’ There is no single answer to this question. After taking so much precaution of defining and implementing the processes, doing verification and validation of each artifact during SDLC, yet nobody can claim that the product is free of any defects. In case of software development and usage, there are many factors responsible for its success/failure. Few of them are,

- There are huge communication losses between different entities as requirements get converted into the actual product. Understanding of requirements is a major issue and majority of the defects can be attributed to this.
- Development people are more confident about their technical capabilities and do not consider that they can make mistakes. Sometimes self review and/or peer review does not yield any defects.

- Requirement changes are very dynamic. As the traceability matrix is not available, impact analysis of changing requirements becomes heuristic.
- Technologies are responsible for introducing few defects. There are many defects introduced due to browsers, platforms, databases, etc. People do not read and understand release notes, and consequences of failure are attributed to technologies.
- Customer may not be aware of all requirements, and the ideas develop as the product is used. Prototyping is used for clarifying requirements to overcome this problem to some extent.

2.14 PROCESSES RELATED TO SOFTWARE QUALITY

Quality environment in an organisation is established by the management. Quality management is a temple built by pillars of quality. Culture of an organisation lays the foundation for quality temple. Every organisation has different number of tiers of quality management system definition. Figure 2.6 shows a relationship between vision, mission(s), policy(ies), goal(s), objective(s) strategy(ies) & values of organisation.

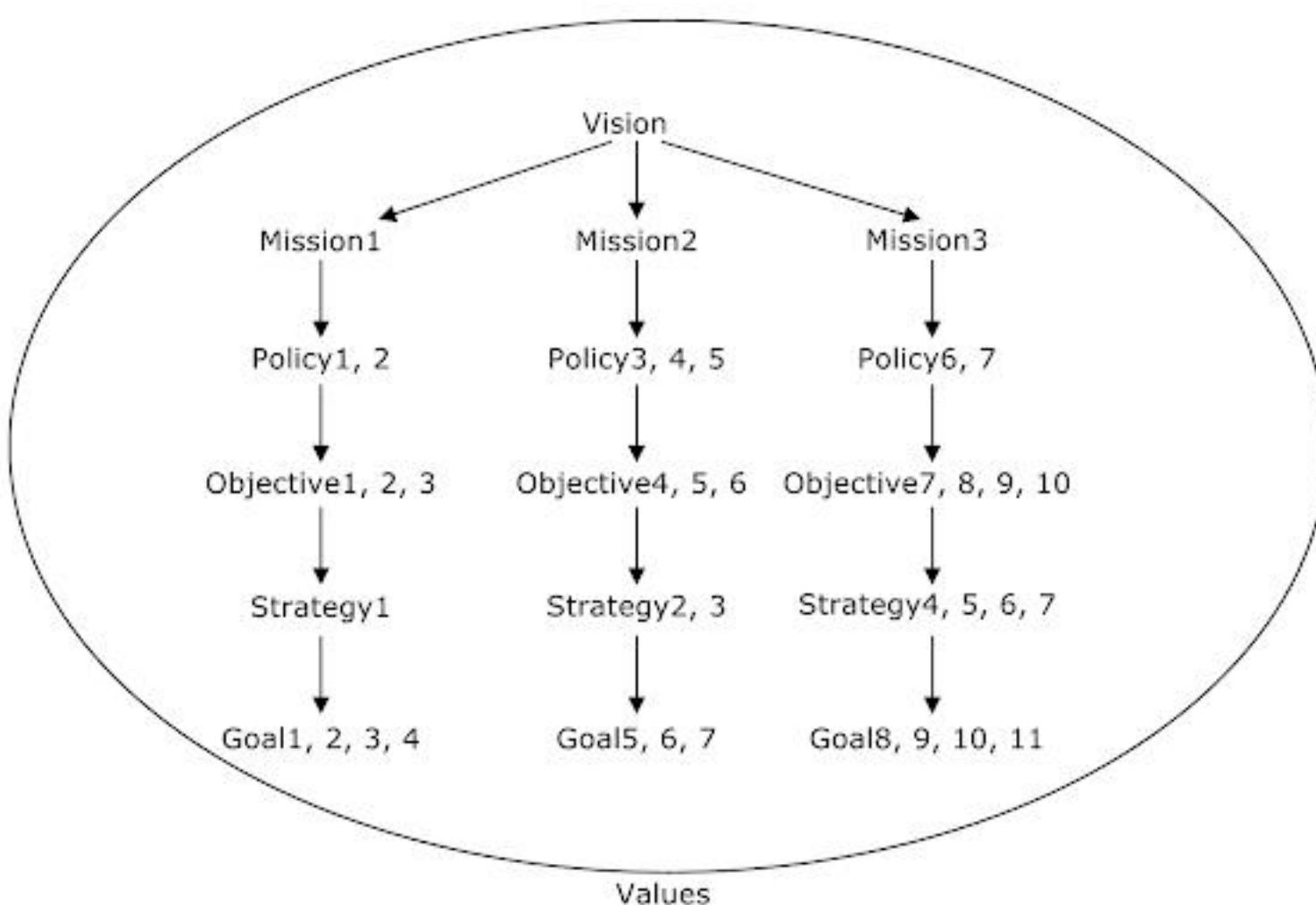


Fig. 2.6

Relationship between Vision, Mission(s), Policy(ies), Objective(s), Strategy(ies), Goal(s) and Values

2.14.1 VISION

The vision of an organisation is established by the policy management. Vision defines in brief about what the organisation wishes to achieve in the given time horizon. ‘To become a billion-dollar company within 3 years’ can be a vision for some organisations. Every organisation must have a vision statement, clearly defining the ultimate aim it wishes to achieve with respect to time span.

2.14.2 MISSION

In an organisation, there are several initiatives defined as missions which will eventually help the organisation realise its vision. Success of all these missions is essential for achieving the organisation's vision. The missions are expected to support each other to achieve the overall vision put by management. Missions may have different lifespans and completion dates.

2.14.3 POLICY

Policy statement talks about a way of doing business as defined by senior management. This statement helps employees, suppliers and customers to understand the thinking and intent of management. There may be several policies in an organisation which define a way of achieving missions. Examples of policies may be security policy, quality policy, and human resource development policy.

2.14.4 OBJECTIVES

Objectives define quantitatively what is meant by a successful mission. It defines an expectation from each mission and can be used to measure the success/failure of it. The objectives must be expressed in numerals along with the time period defined for achieving them. Every mission must have minimum one objective.

2.14.5 STRATEGY

Strategy defines the way of achieving a particular mission. It talks about the actions required to realise the mission and way of doing things. Policy is converted into actions through strategy. Strategy must have a time frame and objectives along with goals associated with it. There may be an action owner to lead the strategy.

2.14.6 GOALS

Goals define the milestones to be achieved to make the mission successful. For a mission to be declared as successful/failure at the end of the defined time frame in terms of whether the objectives are achieved or not, one needs a milestone review to understand whether the progress is in proper direction or not. Goals provide these milestone definitions.

2.14.7 VALUES

Values can be defined as the principles, or way of doing a business as perceived by the management. 'Treating customer with courtesy' can be a value for an organisation. The manner in which the organisation and management think and behave, is governed by the values it believes in.

2.15 QUALITY MANAGEMENT SYSTEM STRUCTURE

Every organisation has a different quality management structure depending upon its need and circumstances. Generic view of quality management is defined below. Figure 2.7 shows a structure of quality management system in general.

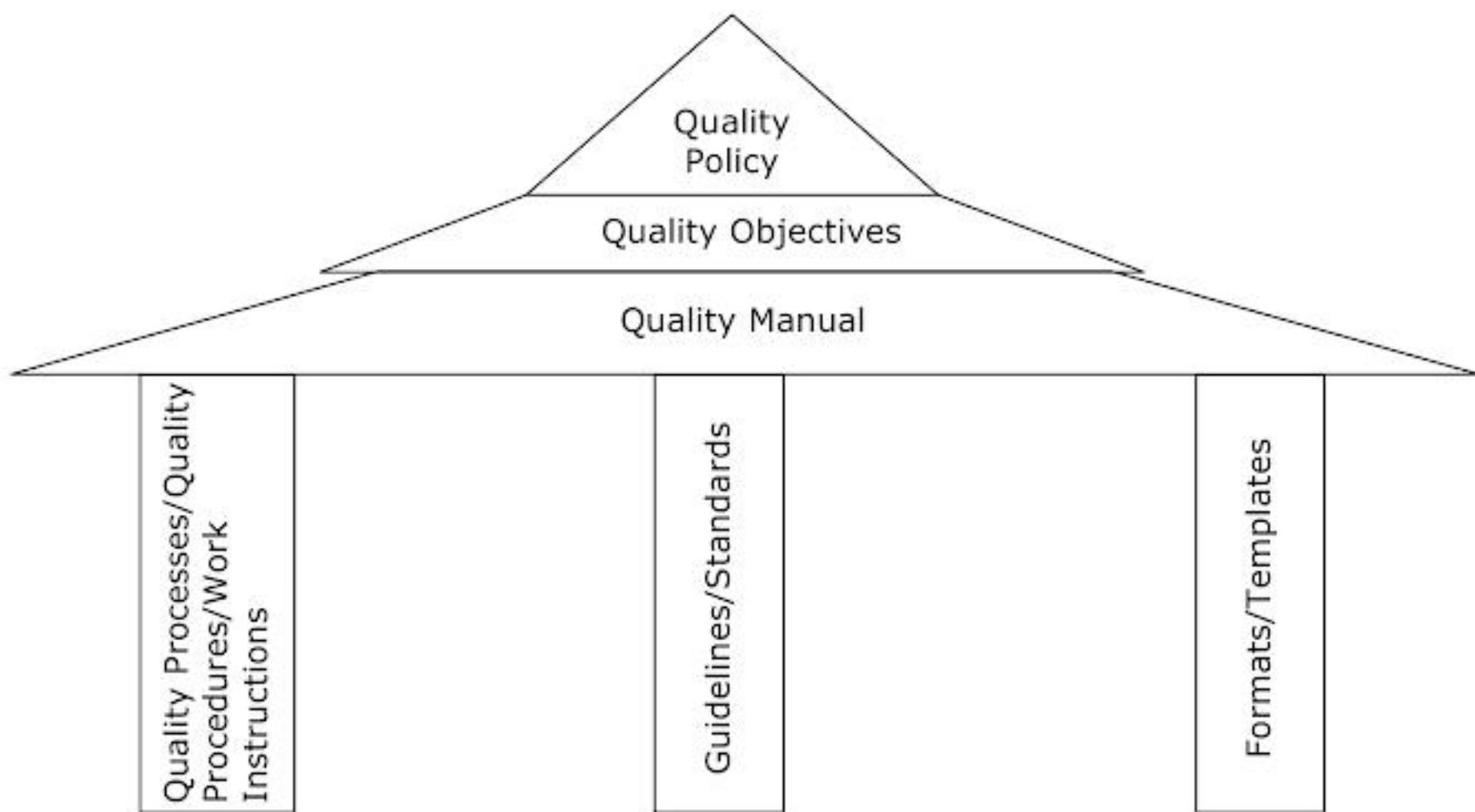


Fig. 2.7

Quality Management System of a typical organisation

2.15.1 1ST TIER—QUALITY POLICY

Quality policy sets the wish, intent and direction by the management about how activities will be conducted by the organisation. Since management is the strongest driving force in an organisation, its intents are most important. It is a basic framework on which the quality temple rests.

2.15.2 2ND TIER—QUALITY OBJECTIVES

Quality objectives are the measurements established by the management to define progress and achievements in a numerical way. An improvement in quality must be demonstrated by improvement in achievements of quality factors(test factors) in numerical terms as expected by the management. The achievements of these objectives must be compared with planned levels expected and results and deviations must be acted upon.

2.15.3 3RD TIER—QUALITY MANUAL

Quality manual, also termed as policy manual is established and published by the management of the organisation. It sets a framework for other process definitions, and is a foundation of quality planning at organisational level.

2.16 PILLARS OF QUALITY MANAGEMENT SYSTEM

Top part of the quality temple is build upon the foundation of following pillars.

2.16.1 QUALITY PROCESSES/QUALITY PROCEDURES/WORK INSTRUCTIONS

Quality processes, quality procedures, work instructions, methods, etc. are defined at an organisation level by the functional area experts, and at project and function level by the experts in those areas separately. Organisation level processes act as an umbrella, whereas project and function level processes are in the purview of these top-level process definitions. Organisation level set of processes may differ from the process definition for different projects and functions. It is also defined as quality planning at project level. Quality procedures must be in sync with the tone established by quality manual at an organisation level.

2.16.2 GUIDELINES AND STANDARDS

Guidelines and standards are used by an organisation's project team for achieving quality goals for the products and services delivered to customers. Many a times, guidelines defined by customers are termed as standards for the project, as the project team takes the recommendations by customers as mandatory. Difference between a guideline and a standard is defined as, shown in Table 2.3.

Table 2.3

Difference between guidelines and standards

Guidelines	Standards
Guidelines are suggested ways of doing things. They are made by experts in individual fields.	Standards are mandatory ways of doing things. These are also described by experts in respective fields.
Guidelines may be overruled and there is no issue if somebody does not follow it.	Overruling of standards is a punishable offence. It may lead to nonconformance during reviews and audits.
Guidelines may or may not be written. Generally it is recommended that one must write the guidelines to capture the tacit knowledge.	Standards must be written to avoid any misunderstanding or loss of communication.
Guidelines and standards may need revision from time to time. Revisions must be done to maintain suitability over a time period.	

2.16.3 FORMATS AND TEMPLATES

Common formats and templates are used for tracking a project, function, and department information within an organisation. It creates same understanding across the board where outputs can be compared for the projects and functions. This also acts as a checklist to maintain consistency across the projects in the organisation. Formats and templates, if made compulsory, are considered as standards whereas if they are indicative or suggestive, they are considered as guidelines. Generally templates are mandatory while formats are suggestive in nature.

2.17 IMPORTANT ASPECTS OF QUALITY MANAGEMENT

Quality improvement is not an accident but a planned activity. An organisation must plan for improvement under the leadership of management and with employee participation.

2.17.1 QUALITY PLANNING AT ORGANISATION LEVEL

An organisation creates quality plan at the organisation level for achieving quality objectives, goals, its vision and missions. Quality planning includes establishing missions, policies and strategies at organisation

level along with objectives and goals to achieve the vision. It must set a framework for definition and implementation of good processes, practices, recruiting people, infrastructures, hardware and software. There should be an appraisal of quality achieved as against expected results at planned intervals, and actions must be initiated in case of any deviation.

2.17.2 QUALITY PLANNING AT PROJECT LEVEL

Projects should plan for quality at project level. These are generally strategic-level quality plans with details of responsibilities and actions. Project plan must define all aspects of quality plan at project level, and may have a relation with the organisation's quality planning. The quality objectives of the project may be inherited from organisation level objectives or may be defined separately for the project. Project level objectives must be in sync with organisation level objectives.

2.17.3 RESOURCE MANAGEMENT

An organisation should use good inputs as required by quality planning so that the output of the processes match with the organisation's business plans. It includes people, machines, materials, and methods as the basic resources. Good processes and good technology need good people to perform the work and achieve planned results.

2.17.4 WORK ENVIRONMENT

Working environment is an important input for a good product and to achieve the organisation vision and missions. A good environment can help an organisation to build on its strength while a bad environment is a roadblock to achieving objectives, and can create problems in its mission of customer satisfaction. Many organisations employ special techniques of 'Working Climate Analysis' to understand its environment, and take actions for correcting it.

Work environment has two components, viz. external environment and internal environment. External environment is mainly a physical environment while internal environment is built in the heart and brain of individuals. Good team spirit and loyalty can be major factors contributing to organisational success.

2.17.5 CUSTOMER-RELATED PROCESSES

Customer-related processes must be analysed for their capability in servicing customers and achieving customer satisfaction. Requirement analysis, designing, project processes, product delivery and other processes related to the customer must be analysed for their capabilities, and corrective/preventive actions must be initiated if those are found to be inadequate. Only capable processes can yield results in a consistent way.

2.17.6 QUALITY MANAGEMENT SYSTEM DOCUMENT AND DATA CONTROL

Many organisations define quality management system on the basis of some quality standards/models. There may be some specific customer requirements for different standards and models which may help an organisation in defining its own quality management system and following customer directives. Statistical process control and data management are essential for continuous improvement of processes.

2.17.7 VERIFICATION AND VALIDATION

Verification and validation are performed by an organisation at each level of development and for each activity. Verification includes management reviews and technical reviews (such as code review and project plan

review) whereas validation involves different kinds of testing (such as unit testing and system testing) to ensure that the work product meets the predefined acceptance criteria.

Verification Verification defines the processes used to build the product correctly. Verification is successful, if the processes and procedures are followed correctly as defined by the process framework and also, they are capable of giving results. Verification cannot directly ensure that the right product has been built but checks if it has been built in the right way.

Validation Validation ensures that the right product has been built. It involves testing a software product against requirement specifications, design specifications, and customer needs. The method followed for development may or may not be correct or capable, but the final output should be as per customer requirements.

2.17.8 SOFTWARE PROJECT MANAGEMENT

Project management is a specific skill required in leaders of projects (for example, project manager). Project management involves planning, organising, staffing, directing, coordinating and controlling the project to satisfy customers by delivering the right product, on time, in the budgeted cost. Nowadays project managers have to perform different tasks like mentoring, guiding, and supporting rather than supervising people.

2.17.9 SOFTWARE CONFIGURATION MANAGEMENT

The work products are built and tested again and again. The defects found during verification and validation are corrected, and the work product undergoes further updatings, integration and testing. Software configuration management involves creating work products, maintaining them, reviewing them by related stakeholders and updating them as and when required. Baseline work products are released for further development process.

2.17.10 SOFTWARE METRICS AND MEASUREMENT

New methods of project management approach stress a need for measuring the product attributes and process capabilities to achieve the quality of final deliverables to customer. Metrics and measurement programs are established by an organisation to capture metrics data/process measurement to ensure that processes are followed correctly and are capable of giving desired outputs. The lacunae found in the processes as well as products can be taken as an input to initiate corrective actions.

2.17.11 SOFTWARE QUALITY AUDITS

Audit is defined in dictionaries as an examination of accounts. Quality audits of software products and processes must be performed to analyse the quality of the products as well as processes used to make them. These can help in analysis of the situation and taking corrective/preventive actions at proper levels. Software quality audits are performed by qualified auditors at predefined levels. Audits can be categorised, as per the following:

The Auditing Agency Involved

- Internal audits are conducted by the people internal to the organisation. It is also called as first-party audits.

- Customer audits are conducted by the customer or customer representatives. It is also called as second-party audits.
- Certification audits are conducted by third-party certification bodies.

The Phase When the Audit is Conducted

- Kick-off audits are conducted at the start of the activity, say at a project start.
- Phase-end audits at the end of a phase.
- Pre-delivery audits are conducted before giving any deliverable to a customer.
- Postmortem audits are conducted at the end of the activity, say a project closure.

Subject of the Audit

- Product audits are conducted to ensure that planned arrangements for quality are achieved or not.
- Process audits are conducted to ensure that processes defined are followed or not.

2.17.12 SUBCONTRACT MANAGEMENT

Suppliers are the stakeholders for the organisation and projects. An organisation must build a strong bond of relationship with its suppliers. It must analyse the inputs from suppliers to make sure that the organisation gets proper inputs so that outputs can be managed as planned. There must be a methodology supported by values which stresses on developing a long-term relationship with the suppliers and avoids least purchase cost bids to total cost-benefit analysis. Suppliers may be supported to do statistical process control to reduce cost and delay in supply or service problems.

2.17.13 INFORMATION SECURITY MANAGEMENT

Information is one of the biggest assets of the organisation. An organisation must protect the information assets available in various databases and all information that has been developed, captured, and used. It must be able to use that information for its continuous/continual improvement. Tacit knowledge is very important from security of information. Information security is associated with three buzz words viz. confidentiality, integrity and availability.

2.17.14 MANAGEMENT REVIEW

Management must periodically review the status of different projects and functions to understand progress which in turn will help in achieving the organisation's vision. Management must plan for corrective and preventive actions if required as indicated by metrics. It must decide upon future business plans for improvements. Management reviews must be systematic and planned. Inputs, processes and outputs of management reviews must be defined.

Software quality tips

It is essential for an organisation to know whether it is achieving the target quality or not. There are some simple tips which can define the organisation's success in terms of achieving quality. Some of them are as follows,

- **Aim at Customer Satisfaction** Everything done by an organisation is for achieving customer satisfaction. Management must devise a process of collecting customer feedback and periodically measure and monitor customer satisfaction. It must initiate actions where the customer feedback is negative.
- **Have Measurable Objectives** Measurable objectives are essential to track the progress made by the organisation. Qualitative objectives may or may not be sufficient to ensure its achievement as the organisation achieves maturity, and the organisation should try to put quantitative objectives, atleast for critical processes. Organisations must have a definition of goals along with objectives for continuous measurements.
- **Understand Requirements Accurately** Understanding and defining customer requirements is the most challenging work for business analyst, system analyst and management. It needs to ensure that a developer must understand the requirements correctly and interpret the requirements into correct product. Requirement losses in terms of misinterpretation must be reduced. Implied requirements must be converted into defined requirements.
- **Implement P-D-C-A Cycle in Each Phase** Plan–Do–Check–Act cycle of continual (continuous) improvement must be followed to ensure improvement in product and process quality. An organisation must plan for future, do the things as planned, check the actual results with the planned ones and take actions on deviations.
- **Detect and Remove Defects as Early as Possible, Prevention is Better Than Cure** In software development, longer the defect remains in a system, more costly and more difficult it is to remove it. It would be always advantageous to detect the defect through review and testing process as early as possible. All defects must lead to process improvements so that defect recurrence is avoided.
- **Systematic Change Control and Version Control** Any change in work product must go through the stages of draft, review, approve and baseline. Version control and labeling is used effectively during development process to identify work products. Many tools are available for managing changes, though it can also be done manually. Configuration management is very important to give the right product to the customer.
- **Follow Easy to Use Standards/Conventions for Naming, Commenting, Coding and Documentation** An organisation must define standards and guidelines which are very useful for normal developers and testers. Common standards and guidelines show the best way to do things. It spreads common understanding across the teams and reduces the chances of misinterpretation. People do not have to invent the wheel again and again but can use the experience of others by referring to these guidelines and standards. They must be very simple to understand and use.
- **Start with Compiling and Analysing Simple Metrics** An organisation must define simple metrics at the start which are useful for planning improvements and tracking them. The main purpose of metrics is to define improvement actions needed and measuring how much has been achieved.

Summary

This chapter is based upon the foundation of the earlier chapter where we have seen quality perspectives. In this chapter, we have studied different constraints faced while building and testing a software product. It tries to link the relationship between quality improvement and productivity improvement. We have seen the cultural difference between quality conscious organisations 'Q' and less quality conscious organisations 'q'.

This chapter elucidates various development models such as waterfall, iterative, incremental, and prototyping. New development methodologies like agile have also been introduced. We have also seen the criticality definitions of different systems from the perspective of different stakeholders and how a tester must understand these system criticalities before devising testing.

We have learnt different types of requirements and problems faced while defining these requirements. We have listed all the quality factors (test factors) applicable for a system and how it affects testing.

Finally, we have dealt with quality management system as a generic model and seen that prevention is required rather than finding and fixing the defects.

- 1) What are the constraints of software requirement specifications?
- 2) Explain relationship between quality and productivity.
- 3) Explain the concept of 'q' organisations and 'Q' organisations
- 4) Explain different development models.
- 5) How products are classified depending upon their criticality?
- 6) What are different types of requirements?
- 7) What problems are posed by the requirement stage?
- 8) What are the characteristics of good requirements?
- 9) Explain difference between expressed and implied requirements.
- 10) Explain difference between present and future requirements.
- 11) Explain difference between generic and specific requirements.
- 12) List and explain quality objectives (test objectives) applicable to software development and usage.
- 13) Explain generic quality management system structure for an organisation.



FUNDAMENTALS OF SOFTWARE TESTING



OBJECTIVES

This chapter aims to provide a basic knowledge of testing. It clearly highlights the difference between 'Total Quality Management' and 'Big Bang' approaches to testing. It also defines different methodologies used in testing such as 'Black Box Testing', 'White Box Testing' and 'Gray Box Testing'. The chapter concludes with test processes including process of defining test policy, test strategy, and test plan.



3.1 INTRODUCTION

Software development activities during a life cycle have corresponding verification and validation activities at each stage of software development. Software verification involves comparing a work product with processes, standards, and guidelines. Software validation activities are associated with

checking the outcome of developed product and the processes used with respect to standards and expectations of a customer. It is considered as a subset of software quality assurance activities though there is a huge difference between quality assurance and quality control. Cost of software verification as well as validation comes under appraisal cost, when one is doing it for the first time. When repeat verification/validation (such as retesting or regression testing) is done, it is defined as cost of failure. Software testing involves verification as well as validation activities such as checking the compliance of the artifacts and activities with respect to defined processes and standards, and executing the software program to ensure that it performs correctly as desired by the customer and expressed in requirement specification agreed between development team and customer. Testing involves finding the difference between actual behaviors with respect to the expected behaviors of an application. There are many stages of software testing as per software development life cycle. It begins with feasibility testing at the start of the project, followed by contract testing and requirements testing, then goes through design testing and coding testing till final acceptance testing, which is performed by customer/user.

3.2 HISTORICAL PERSPECTIVE OF TESTING

The concept of independent testing did not prevail during the initial days of software development. It was believed that whatever the developers were doing was the best way of producing the product and the customer was expected to use it as it is. If there were any problems reported by customer/users, they would be fixed by the developers, and the application would be given again to customer/users. The primary responsibility of testing was with customer/users (and not with developers) during the production phase.

Glenford Myer introduced software testing as a separate phase in software development life cycle. According to him, software testers were expected to test software with all the possible combinations. The main intention was to create a software which would never fail in production. Testers were expected to have an attitude to break the software so that it would be eventually corrected and would never fail during use. This approach separated debugging from testing, and an independent testing community was created. Different phases of software testing evolution as a separate discipline in software development activities were followed in a cycle.

3.2.1 DEBUGGING-ORIENTED TESTING

During the initial phase, software testing was considered as a part of software development. Developers were expected to perform debugging on the application, which they were building. Tests were not documented, and were mainly done in a heuristic way. Generally, testing was completely 'positive testing' to see whether the implementation was working correctly or not.

3.2.2 DEMONSTRATION-ORIENTED TESTING

In this phase, there was an introduction of software testers independent of development activity. Their main aim was to show to customer/users that the software really works. Although, this phase was much advanced than the initial phase of debugging, yet the approach was still positive testing only. The approach was oriented towards demonstration that the software could do something which was expected by the customer/users. Test cases were generally derived from the requirement statements which were oriented towards successful demonstration.

3.2.3 DESTRUCTION-ORIENTED TESTING

This approach was the basis of Glenford Myer's theory of software testing. As per this approach, it was not sufficient to only test the software positively but users must also be protected from any conceivable failure of application. The tester's responsibility changed from 'proving that software works under normal conditions' to 'proving that software does not fail at some abnormal instances'. Often, the testers were too imaginative in breaking the software, and defects for which there were no feasibility of happening were reported as defects. This phase was quiet frustrating to software developers—testers were considered as demons and testing was considered as a hurdle to be passed before delivering the application to the customer.

3.2.4 EVALUATION-ORIENTED TESTING

Evaluation-oriented testing is executed nowadays at many places of software development where the product as well as process of software development is evaluated. It corresponds to the testing process definition where software is evaluated against some fixed parameters derived from quality factors (test factors). Quality factors/test factors are introduced in this phase as a part of customer requirements. It is believed that there

is no possibility of software without any defect, but some level of defect may be acceptable to the customer. This approach also refers to level of confidence given to customer that application will work as expected by the user. The confidence level is determined and application is evaluated against it. Confidence-level expectations are linked with cost of testing.

3.2.5 PREVENTION-ORIENTED TESTING

Prevention-based testing is done in some highly matured organisations while for many others, the concept is still utopia. Testing is considered as a prevention activity where process problems are used to improve it so that defect-free products can be produced. Every defect found in testing is considered as process lacunae, and efforts are initiated to improve the processes of development. This reduces dependency on testing as a way to improve the quality of software. This helps in reducing the cost by producing right product at the first time.

3.3 DEFINITION OF TESTING

Testing is defined as ‘execution of a work product with intent to find a defect’. The primary role of software testing is not to demonstrate the correctness of software product, but to expose hidden defects so that they can be fixed. Testing is done to protect the common users from any failure of system during usage.

This approach is based on the assumption that any amount of testing cannot show that software product is defect free. If there is no defect found during testing, it can only show that the scenario and test cases used for testing did not discover any defect. From user’s point of view, it is not sufficient to demonstrate that software is doing what it is supposed to do. This is already done by system architects in system architecture design and testing, and by developers in code reviews and unit testing. Testers are involved mainly to ensure that the system is not doing what it is not supposed to do. Their work includes assurance that the system will not be exposed to any major risks of failure when a normal user is using it. Some people call this approach as negative approach of testing. This negative approach is built upon few assumptions and risks for the software being developed and tested. These assumptions and risks must be documented in the test plan while deciding test strategy or test approach.

3.3.1 WHY TESTING IS NECESSARY?

One may challenge testing activities by asking this question—‘If any level of testing cannot declare that there is no defect in the product, then why is it required at all?’ In normal life, we find highly qualified and experienced people involved in each stage of development from requirement gathering till acceptance of software. Finding a defect in software is sometimes considered as challenging the capabilities of these people involved in development phases. Testing is necessary due to the following reasons.

- Understanding of customer requirements may differ from person to person. One must challenge the understanding at each stage of development, and there must be some analysis of customer expectations. Approach-related problems may not be found when there is no detail analysis by another person not involved emotionally with development. Everything is considered as ‘OK’ unless there is an independent view of a system.
- Development people assume that whatever they have developed is as per customer requirements and will always work. But, it is imperative to create real-life scenario and undertake actual execution of a product at each level of software building (including system level) to assess whether it really works or not.

- Different entities are involved in different phases of software development. Their work may not be matching exactly with each other or with the requirement statements. Gaps between requirements, design, and coding may not be traceable unless testing is performed in relation to requirements.
- Developers may have excellent skills of coding but integration issues can be present when different units do not work together, even though they work independently. One must bring individual units together and make the final product, as some defects may be possible when the sources are developed by people sitting at different places.
- There is a possibility of blindfold and somebody has to work as the devil's representative. Every person feels that what he/she has done is perfect and there is no chance of improvement. Testers have to challenge each assumption and decision taken during development.

3.4 APPROACHES TO TESTING

There are many approaches to software testing defined by the experts in software quality and testing. The approaches may differ significantly as per customer requirements, type of the system being developed as well as management thinking about software development life cycle followed by software, type of project, type of customer, and maturity of development team. These approaches form the part of testing strategy. Few of them are discussed below.

3.4.1 BIG BANG APPROACH OF TESTING

Characteristics of 'Big bang' approach involve testing software system after development work is completed. This is also termed 'system testing' or final testing done before releasing software to the customer for acceptance testing. This testing is the last part of software development as per waterfall methodology. Big bang approach has main thrust on black box testing of software to ensure that the requirements as defined and documented in requirement specifications and design specifications are met successfully. Testing done at the end of development cycle may show the defects pertaining to any phase of development such as requirements, design, and coding. Roughly saying, the phase-wise defect origination follows the trend shown in Table 3.1.

Table 3.1

Phase-wise defect distribution

Development phases	Percentage of defects
Requirements	58
Design	35
Coding	5
Other	2

In case of big bang approach, software is tested before delivery using the executable or final product. It may not be able to detect all defects as all permutations and combinations cannot be tested in system testing due to various constraints like time. In such type of testing, one may find a cascading effect of camouflage effect, and all defects may not be detected. It may discover the failures but cannot find the problems effectively. Sometimes, defects found may not be fixed correctly as analysis and defect fixing can be a problem.

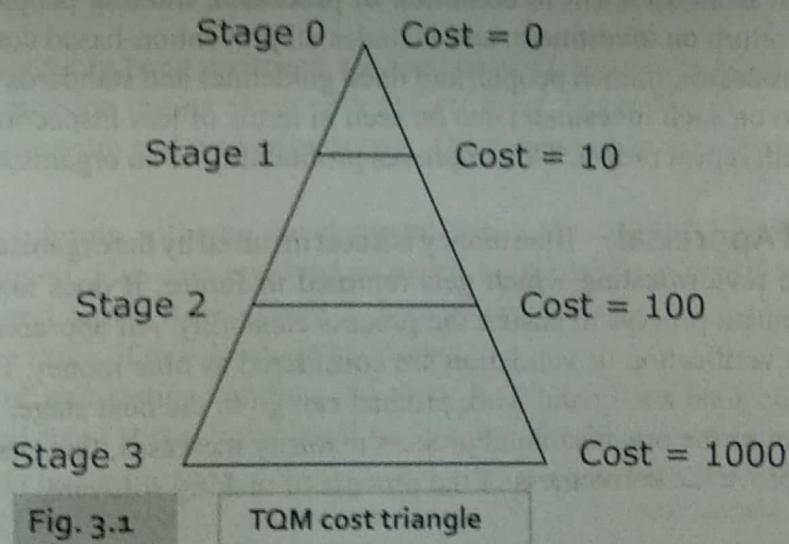
3.4.2 TOTAL QUALITY MANAGEMENT APPROACH

If there is a process definition for testing software, and these processes are optimised and capable, no (less) defects are produced and no (few) undetected defects are left in the software when it is delivered to the customer. Defect removal costs are approximately 10 times more after coding than before coding. This is a cost associated with fixing the problems belonging to requirements, design, coding, etc. If the defect is not detected earlier but found in acceptance testing or further down the line during warranty, it may be much more costly. Defect removal cost would be 100 times more during production (at user site) than before coding. This may involve deploying people at customer site, loss of goodwill, etc, which is a part of failure cost.

3.4.3 TOTAL QUALITY MANAGEMENT (TQM) AS AGAINST BIG BANG APPROACH

Figure 3.1 is a very famous cost triangle defined by TQM. If the organisation has very good processes defined which are optimised and capable, and can produce consistent results again and again, then it gives advantage in productivity and effectiveness in development. It can reduce cost of production significantly.

Stage 0 There may be a stage of maturity in an organisation where no verification/validation is required to certify the product quality. The cost involved is 'zero' or the benefits derived by investment in process definition, optimisation and deployment make quality free. There is a famous saying 'quality is free'. Theoretically, if the processes are optimised, there is no need of verification/validation as the defects are not produced at all.



Stage 1 Even if some defects are produced during any stage of development in such quality environment, then an organisation may have very good verification processes which may detect the defects at the earliest possible stage and prevent defect percolation. There will be a small cost of verification and fixing, but stage contamination can be saved which helps in finding and fixing the defects very fast. This is an appraisal cost represented by '10'. This is the cost which reduces the profitability of the organisation due to scrap, rework and reverification.

Stage 2 If some defects escape the verification process, still there are capable validation processes for filtering the defects before the product goes to a customer. Cost of validation and subsequent defect fixing is much higher than verification. This cost is represented by '100'. One may have to go to the stage where defect was introduced in the product and correct all the stages from that point onward till the defect-detection point. The cost is much higher, but till that point of time the defect has not reached the customer, it may not affect customers feelings or goodwill.

Stage 3 At the bottom of the pyramid, there is the highest cost associated with the defects found by customer during production or acceptance testing. This is represented by '1000' showing that cost paid for fixing such defect is huge. There may be customer complaints, selling under concession, sending people onsite for fixing defects in front of the customer, loss of goodwill, etc. This may result into premature closure of relationship and bad advertisement by the customer.

3.4.4 TQM IN COST PERSPECTIVE

Total quality management (TQM) aims at reducing the cost of development and cost of quality through continual improvement. Often, it is termed 'Quality is free'. It means that the cost of quality must repay much more than what has been invested. TQM defines the cost incurred in development and quality into three parts as follows.

Green Money/Cost of Prevention Green money is considered as an investment by the organisation in doing quality work. It is a cost spent in definition of processes, training people, developing foundation for quality, etc. It gives return on investment, and includes all prevention-based costs. If an organisation has defined and optimised processes, trained people, and fixed guidelines and standards for doing work which are followed, then the return on such investment can be seen in terms of less inspection and testing, and higher customer satisfaction with repeat orders. This improves profitability of an organisation.

Blue Money/Cost of Appraisal Blue money is a cost incurred by the organisation during development, in the form of first-time review/testing which gets returned in future. It does not earn profit, but it is an essential part of development process to ensure the process capability. All appraisal techniques used during SDLC such as first-time verification or validation are considered as blue money. First-time testing helps in certifying that nothing has gone wrong and work product can go to the next stage. In initial phases, the cost of appraisal increases, but as the organisational process maturity increases, this cost must go down as fewer samples are required to prove the correctness of the process of making software.

Red Money/Cost of Failure Red money is a pure loss for the organisation. It involves money lost in scrap, rework, sorting, etc. It also represents loss due to various wastes produced during development life cycle, and directly reduces the profit for the organisation and the customer may not pay for it. As the investment or green money increases, failure cost must go down. All cost incurred in reinspection, retesting, and regression testing represent cost of failure.

3.4.5 CHARACTERISTICS OF BIG BANG APPROACH

Big bang approach talks about testing as the last phase of development. All the defects are found in the last phase and cost of rework can be huge.

- Testing is the last phase of development life cycle when everything is finalised. Heavy costs and efforts of testing are seen at the end of software development life cycle representing 'Big bang' approach. Most of the testing is concentrated in this phase only, as there are no previous verification or validation activities spread during the development phases.
- Big bang approach is characterised by huge rework, retesting, scrap, sorting of software components, and programs after the complete software is built. There may be special teams created to fix defects found in final testing. Big bang works only on correction, and there are no corrective and preventive actions and process improvements arising from these defects. The processes remain immature and every time, defect fixing becomes an invention of the wheel.
- Regression testing reveals many issues, as correction may not be correct and may introduce some defects in the product. There are many interdependencies between various entities while building software product, and these may get affected adversely. When some areas in the software units are touched for correction or fixing of defect, dependent components may get affected in a negative way. These may be termed 'regression defects'.
- All requirements and designs cannot be covered in testing. As the schedule of delivery is fixed and development generally lags behind the schedule, thus huge adhoc, exploratory, monkey, and random testing is done in this part of testing. Testing is done in a hurry, and many iterations of defect fixing and testing are done in the shortest possible time. Some defects may flow to customer as 'known defects' or sometimes they are not declared at all.
- The major part of software build never gets tested as coverage cannot be guaranteed in random testing. Software is generally tested by adhoc methods and intuition of testers. Generally, positive testing is done to prove that software is correct which represents second level of maturity.

Organisations following big bang approach are less matured and pay a huge cost of failure because of several retesting and regression testing along with defect-fixing cycles. Success is completely dependent on good development, testing team and type of customer. The following can be observed.

- Verification activities during software development life cycle can find out about two-third of the total number of defects found. The cost involved in fixing such defects is much less than any other way of finding the defects and fixing them. There is less stage contamination as defects are prevented from progressing from one stage of development to another.
- Validation in terms of unit testing can find out about three-fourth of the remaining defects. Defects are found in the units and fixed at that point itself so that they do not occur further down the line. It reduces the chances of defects being found in system testing. Unit testing validates an individual unit.
- Validation in terms of system testing can find out about 10% of the total number of defects. System testing must be intended to validate system-level requirements along with some aspects of design. Some people term system testing as certification testing while some people term acceptance testing as certification testing. If the exit criteria of system testing (acceptance criteria by customer) are met, system may be released to the customer.

Remaining defects (about 5–10%) go to the customer, unless the organisation makes some deliberate efforts to prevent them from leaking to production phase. Big bang approach may not be useful in preventing defects from going to the customer, as it can find only 5% of the total defects present in the product. In other terms, theoretically, to achieve the effectiveness of life-cycle testing, one may need about 18 cycles of system testing. This can prove to be a costly affair.

3.5 POPULAR DEFINITIONS OF TESTING

Let us try to define "software testing" keeping the background of Big bang approach in mind. All definitions of testing indicate that testing is a life-cycle operation, and not the activity at the end of development phase. No definition of testing can really cover all aspects of testing. Hence, no definition is complete but indicates a part of what software testing is.

3.5.1 TRADITIONAL DEFINITION OF TESTING

There can be many definitions of testing pertaining to different instances. Few of them are as follows.

- Testing is done to establish confidence that the program does what it is supposed to do. It generally covers the functionalities and features expected in the software under testing. It covers only positive testing.
- Testing is considered as any activity aimed at evaluating an attribute or capability of a program or system with respect to user requirements. To some extent, this definition may be considered as correct, as number of defects found in testing is directly proportional to number of defects remaining in the system.
- Testing is a process of demonstrating that errors are not present in the product. This approach is used in acceptance testing where if the application meets acceptance criteria, then it must be accepted by the customer.
- Testing gives number of defects present which indirectly gives a measurement of software quality. More number of defects indicate bad software and bad processes of development.
- Testing is done to evaluate the program or system used for making software. As we consider that defects are introduced due to incapable processes, testing may be used to measure process capability to some extent.
- Testing is used to confirm that a program performs its intended functions correctly. Intended functionality may be defined from requirement specifications.

If testing is defined as a process, then it is designed to,

- Prove that the program is error free or there is no defect present
- Establish that the software performs its functions correctly and is fit for use
- Establish that all expectations of functionalities are available

Testing may not be any of these certification activities. If the goal of testing is to prove that an application works correctly, then the tester should subconsciously work towards this goal, choosing test data that would prove that the system is working correctly. The reverse would be true if the goal of testing is to locate defects so that eventually these would be corrected. Test data should be selected with an eye towards providing the test cases that are likely to cause product failure.

3.5.2 WHAT IS TESTING?

Let us try to define what is meant by testing with this background. Testing is completely guided by software requirements specifications and design specifications, and supported by test strategy and test approach depending on assumptions and risks of development, testing and usage. Testing process may include the

- An activity of identification of the differences between expected results and actual results produced, during execution of software application. Difference between these two results suggests that there is a possibility of defect in the process and/or work product. One must note that this may or may not be a defect.
- Process of executing a program with the intention of finding defects. It is expected that these defects may be fixed by the development team during correction, and the root causes of the defects are also found and closed during corrective actions. This can improve development process.
- Detecting specification-related errors and deviations of working application with respect to the specifications. Requirement mismatches and misinterpretation must be detected by testing.
- Establish confidence that a program does what it is supposed to do. This defines the confidence level imparted by software testing to a customer that the software will work under normal conditions. The expectation of confidence level is a function of depth and width of software testing.
- Any activity aimed at evaluating an attribute of a program or software. Acceptance testing is an activity defining whether the software has been accepted or not.
- Measurement of software quality in terms of coverage of testing (in terms of requirements, functionality, features, and number of defects found) can give information about confidence level imparted to a customer.
- Process of evaluating processes used in software development. Every failure/defect indicates a process failure. This can be used to improve development processes.
- Verifying that the system satisfies its specified requirements as defined, and is fit for normal use. Requirements may be elicited with the help of the customer.
- Confirming that program performs its intended functions correctly
- Testing is the process of operating a system or component under specified conditions, observing and recording the results of such processing, and evaluating some aspect of system or component on the basis of testing
- Software testing is the process of analysing a software item to detect the difference between existing and required conditions, and to evaluate the feature of the software item.

We have previously discussed about different stakeholders and their interests in software development and testing. Let us try to analyse the expectations or views of different stakeholders about testing.

3.5.3 MANAGER'S VIEW OF SOFTWARE TESTING

The senior management from development organisation and customer organisation have the following views about testing the software product being developed.

- The product must be safe and reliable during use, and must work under normal as well as adverse conditions when it is actually used by the intended users.
- The product must exactly meet the user's requirements. These may include implied as well as defined requirements.
- The processes used for development and testing must be capable of finding defects, and must impart the required confidence to the customer.

3.5.4 TESTER'S VIEW OF SOFTWARE TESTING

Testers have different definitions about software testing, as mentioned below.

- The purpose of testing is to discover defects in the product and the process related to development and testing. This may be used to improve the product and processes used to make it.

- Testing is a process of trying to discover every conceivable fault or weakness in a work product so that they will be corrected eventually. Random testing sometimes become too imaginative, and unconceivable defects may be found.

3.5.5 CUSTOMER'S VIEW OF SOFTWARE TESTING

Customer is the person or entity who will be receiving/using the product and will be paying for it. Testers are considered as the representatives of the customer in system development.

- Testing must be able to find all possible defects in the software, alongwith related documentation so that these defects can be removed. Customer must be given a product which does not have defects (or has minimum defects).
- Testing must give a confidence that software users are protected from any unreasonable failure of a product. Mean time between failures must be very large so that failures will not occur, or will occur very rarely.
- Testing must ensure that any legal or regulatory requirements are complied during development.

Testing is an activity which is expected to reduce the risk of software's failure in production. All stakeholders have many expectations from testing. Let us try to analyse the meaning of a successful tester.

3.5.6 OBJECTIVES OF TESTING

To satisfy the definition of testing given earlier, testing must accomplish the following things.

- Find a scenario where the product does not do what it is supposed to do. This is deviation from requirement specifications
- Find a scenario where the product does things it is not supposed to do. This includes risk

The first part refers to specifications which were not satisfied by the product while the second part refers to unwanted side effects while using the product.

3.5.7 BASIC PRINCIPLES OF TESTING

The basic principles on which testing is based are given below.

- Define the expected output or result for each test case executed, to understand if expected and actual output matches or not. Mismatches may indicate possible defects. Defects may be in product or test cases or test plan.
- Developers must not test their own programs. No defects would be found in such kind of testing as approach-related defects will be difficult to find. Development teams must not test their own products. Blindfolds cannot be removed in self testing.
- Inspect the results of each test completely and carefully. It would help in root cause analysis and can be used to find weak processes. This will help in building processes rightly and improving their capability.
- Include test cases for invalid or unexpected conditions which are feasible during production. Testers need to protect the users from any unreasonable failure so that one can ensure that the system works properly.
- Test the program to see if it does what it is not supposed to do as well as what it is supposed to do.
- Avoid disposable test cases unless the program itself is disposable. Reusability of test case is important for regression. Test cases must be used repetitively so that they remain applicable. Test data may be changed in different iterations.

- Do not plan tests assuming that no errors will be found. There must be targeted number of defects for testing. Testing process must be capable of finding the targeted number of defects.
- The probability of locating more errors in any one module is directly proportional to the number of errors already found in that module.

3.5.8 SUCCESSFUL TESTERS

The definition by testers about testing talks about finding defects as the main intention of testing. Testers who find more and more number of defects are considered as successful. This gives some individuality to testing process which talks about ability of a tester to find a defect. There is a difference between executing a test case and finding the defect. This needs an ability to look for detailing, problem areas, and selection of test data accordingly.

- Testers must give confidence about the coverage of requirements and functionalities as defined in test plan.
- Testers must ensure that user risks are identified before deploying the software in production.
- Testers must conduct SWOT analysis of the software and processes used to make it. This can help in strengthening the weaker areas and the processes responsible for defects so that the same problems do not recur.

3.5.9 SUCCESSFUL TEST CASE

Testing is a big investment and justify its existence, if it catches a defect before going to the customer. Every defect caught before delivery means the probability of finding a defect by a customer is reduced. If testing does not catch any defect, it is a failure of testing and a waste for the organisation as well as customer.

Testing involved in software development life cycle starts from requirements, goes through design, coding, and testing till the application is formally accepted by user/customer.

3.6 TESTING DURING DEVELOPMENT LIFE CYCLE

Let us discuss life cycle phase and testing associated with it. This discussion is based on the consideration that development methodology follows waterfall cycle/model.

Requirement Testing Requirement testing involves mock running of future application using the requirement statements to ensure that requirements meet their acceptance criteria. This type of testing is used to evaluate whether all requirements are covered in requirement statement or not.

This type of testing is similar to building use cases from the requirement statement. If the use case can be built without making any assumption about the requirements, by referring to the requirements defined and documented in requirement specification documents, they are considered to be good. The gaps in the requirements may generate queries or assumptions which may possibly lead to risks that the application may not perform correctly. Gaps also indicate something as an implied requirement where the customer may be contacted to get the insight into business processes. It is a responsibility of business analyst to convert (as many as possible), implied requirements to expressed requirements. Target is 100%, though it is difficult to achieve.

Requirement testing differs from verification of requirements. Verification talks about review of the statement containing requirements for using some standards and guidelines, while testing talks about dummy

execution of requirements to find the consistency between them, i.e., achievement of expected results must be possible by requirements without any assumption. Verification of requirements may talk about the compliance of an output with defined standards or guidelines.

The characteristics of requirements verification or review may include the following.

- Completeness of requirement statement as per organisation standards and formats. It must cover all standards like performance and user interface expected by customer organisation.
- Clarity about what is expected by the users at each step of working while using an application. It must include the expected output by the customer. It may be in the form of error messaging, screen outputs, printer outputs, etc.
- Measurability of expected results, possibly in numerals, so that these results can be tested. Test case will have expected results which must satisfy measurement criteria defined. 'User friendliness' or 'fairly fast' are words which can create confusion about requirements.
- Testability of the scenario defined in requirement statement is must. Some requirements like application must work 24×7 for 10 years may not be directly testable.
- Traceability of requirements further down the development life cycle must be ensured. Requirement traceability starts at requirement phase and gets populated as one goes down the life cycle.

Theoretically, each statement in requirement document must give atleast one functional/non-functional test scenario which may result into test cases. Requirements must be prioritised as 'must', 'should be' and 'could be' requirements. The customer is an entity to confirm requirement priority.

Requirement validation must define end-to-end scenario completely so that there is no gap. It must talk about various actors, transactions involved and information transfer from one system to another.

Design Testing Design testing involves testing of high-level design (system architecture) as well as low-level design (detail design). High-level design testing covers mock running of future application with other prerequisites, as if it is being executed by the targeted user in production environment. This testing is similar to developing flow diagrams from the designs, where flow of information is tracked from start to finish. When the flow is complete, the design may be considered as good. Wherever the flow is not defined, or not clear about where it will lead to, there are defects with design which must be corrected. For low-level design, system requirements and technical requirements are mapped with the entities created in design to ensure adequacy of detail design.

Design verification talks about reviewing the design, generally by the experts who may be termed as subject-matter experts. It involves usage of standards, templates, and guidelines defined for creating these designs. Design verification ensures that designs meet their exit criteria.

- Completeness of design, in terms of covering all possible outcomes of processing and handling of various controls as defined by requirements
- Clarity of flow of data within an application and between different applications which are supposed to work together in production environment
- Testability of a design which talks about software structure and structural testing
- Traceability with requirements
- Design must cover all requirements

Code Testing Code files, Tables, Stored procedures etc are written by developers as per guidelines, standards, and detail design specifications. In reality, developers do not implement requirements directly but

they implement detail design as defined by the designer. Code testing (unit testing) is done by using stubs/drivers as required. Code review is done to ensure that code files written are,

- Readable and maintainable in future. There are adequate comments available.
- Testable in unit testing.
- Traceable with requirements and designs. Anything extra as well as anything missing can be considered as a defect.
- Testable in integration and system testing.
- Optimised to ensure better working of software. Reusability creates a lighter system.

Test Scenario and Test Case Testing Test scenarios are written by testers to address testing needs of a software application. Test cases are derived from test scenarios which are related to requirements and designs. Test scenarios can be functional as well as structural, depending upon the type of requirement and design they are addressing.

- Test scenario should be clear and complete, representing end-to-end relationship of what is going to happen and also, the possible outcomes of such processing.
- Test scenarios should cover all requirements. Test scenarios may be prioritised as per requirement priorities.
- Scenarios should be feasible so that they can be constructed during testing.
- Test cases should cover all scenarios completely.
- Test scenarios and test cases must be prioritised so that in case of less time availability, the major part of the system (where priority is higher) is tested.

3.7 REQUIREMENT TRACEABILITY MATRIX

Some quality management models and standards prescribe complete traceability of a software application from requirements through designs and code files upto test scenario, test data, test cases and test results. Requirement traceability matrix is one way of doing the complete mapping for the software. One can expect a blueprint of an entire application using requirement traceability matrix.

Typical requirement traceability matrix is as shown in Table 3.2.

Table 3.2

Requirement traceability matrix

Requirements	High-level Design	Low-level design	Code files/ Stored Proce- dures/TBLs	Test scenario	Test cases	Test results

3.7.1 ADVANTAGES OF REQUIREMENT TRACEABILITY MATRIX

As discussed earlier, requirement traceability matrix is a blueprint of software under development. All the agencies concerned with software can use it to understand the software in a better way. It may answer questions about what is being developed and how it will be implemented. It helps in tracing if any software requirement is not implemented, or if there is a gap between requirements and design further down the line. It also helps to understand if any redundancy has been created in the application.

- Entire software development can be tracked completely through requirement traceability matrix.
- Any test case failure can be tracked through requirements, designs, coding, etc.
- Any changes in requirements can be affected through entire work product upto test cases and vis-à-vis any test case failure can be traced back to requirements.
- The application becomes maintainable as one has complete relationship from requirement till test results available.

3.7.2 PROBLEMS WITH REQUIREMENT TRACEABILITY MATRIX

Theoretically, all softwares must have requirement traceability matrix, but in reality, most of the softwares do not have it. The reasons are numerous; some of the prominent ones are listed below.

- Number of requirements is huge. It is very difficult to create requirement traceability matrix manually. For using some tools, one needs to invest money. Also, people may need to be trained for using tools.
- There may be one-to-many, many-to-one and many-to-many relationships between various elements of traceability matrix, when we are trying to connect columns and rows of traceability matrix, and maintaining these relationships need huge efforts.
- Requirements change frequently, and one needs to update the requirement traceability matrix whenever there is a change. Similarly designs, code and test cases may also change which will affect traceability matrix.
- Developing teams may not understand the importance of requirement traceability matrix, if development follows waterfall model, and during maintenance, it may be too late to create it. Incremental and iterative developments are the major challenges for maintaining traceability.
- A customer may not find value in it and may not pay for it

3.7.3 HORIZONTAL TRACEABILITY

When an application can be traced from requirement through design and coding till test scenario and test cases upto test results, it is termed as horizontal traceability. On failure of any test case, we must be able to find which requirements have not been met. Any design which does not have requirement, introduces an extra feature which may be considered as defect. Similarly, when any requirement is not traceable to design, that requirement is not implemented at all. Same thing can happen in the relationship between design and coding, that is, coding and test scenario, and also, test scenario and test case. When any entity can't be traced in forward direction, horizontal traceability is lost.

3.7.4 BIDIRECTIONAL TRACEABILITY

One must be able to go from requirements, designs, coding, and testing to reach the test result. Reverse must also be possible, where one may start from the result and go to requirements. One must be able to go in any

direction from any point in traceability matrix. This is referred to as 'bidirectional traceability'. CMMI model mandates bidirectional traceability for all products.

3.7.5 VERTICAL TRACEABILITY

Traceability explained above is called 'horizontal traceability' as it goes in horizontal direction, either forward or backward. Traceability may exist in individual column as the requirements may have some interdependencies between them, or these may be child and parent relationships. For achieving a requirement, the other child requirement must be achieved. One requirement may have several child requirements, while some child requirements may have several parent requirements. If these requirements are traced completely, it ensures vertical traceability. Similarly design, coding, and testing may have a vertical traceability where there may be parent-child relationship and interdependence on different parts. Designs may have parent-child relationships, and coding may have 'called functions' and 'calling functions' traceability relationships.

3.7.6 RISK TRACEABILITY

Some application development organisations also add references about the risks of failure faced by the application in Failure Mode Effect Analysis (FMEA). The risks are traced to requirements and mainly with design which defines control mechanism to reduce probability or impact or improves detection ability of a risk. This helps the customer to identify which accident-prone zones are in the application and where the user is completely/partially protected from failures. It also helps in identifying various types of controls that are designed and used. Typical risk traceability matrix is as shown in Table 3.3.

Table 3.3

Risk traceability matrix

Risk	High-level Design/Control	Low-level design/Control	Code files/ Stored procedures/TBLs	Test scenario	Test cases	Test results

3.8 ESSENTIALS OF SOFTWARE TESTING

Software testing is a disciplined approach. It executes software work products and finds defects in it. The intention of software testing is to find all possible failures, so that eventually these are eliminated, and a good product is given to the customer. It intends to find all possible defects and/or identify risks which final user may face in real life while using the software. It works on the principle that no software is defect free, but less risky software is better and more acceptable to users. The tester's job is to find out defects so that they will be eventually fixed by developers before the product goes to a customer. Completion of testing must yield number of defects which can be analysed to find the weaker areas in the process of software development. No amount of testing can show that a product is defect free as nobody can test all permutations and combinations possible in the given software. Software testing is also viewed as an exercise of doing a SWOT analysis of software product where we can build the software on the basis of strengths of the process of development and testing, and overcome weakness in the processes to the maximum extent possible.

Strengths Some areas of software are very strong, and no (very less) defects are found during testing of such areas. The areas may be in terms of some modules, screens, and algorithms, or processes like requirement definition, designs, coding, and testing. This represents strong processes present in these areas supporting development of a good product. We can always rely on these processes and try to deploy them in other areas.

Weakness The areas of software where requirement compliance is on the verge of failure may represent weak areas. It may not be a failure at that moment, but it may be on the boundary condition of compliance, and if something goes wrong in production environment, it will result into defect or failure of software product. The processes in these areas represent some problems. An organisation needs to analyse such processes and define the root causes of problems leading to these possible failures. It may be attributed to some aspects in the organisation such as training, communication, etc.

Opportunity Some areas of the software which satisfy requirements as defined by the customer, or implied requirements but with enough space available for improving it further. This improvement can lead to customer delight (it must not surprise the customer). These improvements represent ability of the developing organisation to help the customer and give competitive advantage. It decides the capability of the developing organisation to provide expert advice and help to the customer for doing something better.

Threats Threats are the problems or defects with the software which result into failures. They represent the problems associated with some processes in the organisation such as requirement clarity, knowledge base and expertise. An organisation must invest in making these processes stronger. Threats clearly indicate the failure of an application, and eventually may lead to customer dissatisfaction.

3.9 WORKBENCH

Workbench is a term derived from the engineering set-up of mass production. Every workbench has a distinct identity as it takes part in the entire development life cycle. It receives something as an input from previous workbench, and gives output to the next workbench. This can be viewed as a huge conveyor belt where people are working their part while the belt is moving forward. The complete production and testing process is defined as set of interrelated activities where input of one is obtained from the output of previous activity, and output of that activity acts as an input to the next. Each activity represents a workbench. A workbench comprises some procedures defined for doing a work, and some procedures defined to check the outcome of the work done. The work may be anything during software development life cycle such as collecting the requirements, making designs, coding, testing, etc. Organisational process database refers to the methods, procedures, processes, standards, and guidelines to be followed for doing work in the workbench as well as for checking whether the processes are effective and capable of satisfying what customer is looking for in the outcome. There are standards and tools available for doing the work and checking the work in the workbench. While checking/testing a work product in a workbench, if one finds deviations between expected result and actual result, it may be considered as defect, and the work product and the process used for development needs to be reworked.

3.9.1 TESTER'S WORKBENCH

Tester's workbench is made of testing process, standards, guidelines and tools used for conducting tests, and checking whether the test processes applied are effective or not. For every workbench, there should be

a definition of entry criteria, process of doing/checking the work, and exit criteria. For testers, there must be a definition of all things that enter the testers workbench. These may be defined in a test plan. Let us discuss with an example of test-case execution as one activity represented by a workbench.

Examples of Tester's Workbench Considering a typical system testing life cycle for a product/project, the different work benches for a tester may be defined as follows. Kindly note that it is not an exhaustive list but a representative one. As one goes into finer details, there may be many more workbenches in each of these defined below.

- Workbench for creating test strategy
- Workbench for creating a test plan
- Work bench for writing test scenario
- Workbench for writing test cases
- Workbench for test execution
- Workbench for defect management
- Workbench for retesting
- Workbench for regression testing

The following is a typical workbench described for system testing execution.

Inputs to Tester's Workbench Inputs may be test scenario, test cases, work products, documentation associated with a work product, test environment, or test plan depending upon the location of workbench in life cycle. The software work product is delivered to the tester as described in delivery note supplied by development team. Delivery note must contain any known issue which tester needs to know before performing testing.

Do Process The software undergoes testing as per defined test case and test procedure. This may be guided by organisational process database defining testing process. 'Do process' must guide the normal tester while doing the process.

Check Process Evaluation of testing process to compare the achievements as defined in test objectives is done by 'check process'. Check process helps in finding whether 'do processes' have worked correctly or not.

Output Output must be available as required in form of test report and test log from the test process. Output of the tester's workbench needs to have an exit criteria definition.

Standards and Tools During testing, the tester may have to use several standards and tools. Standards may include how to install the application, which steps are to be followed while doing testing, how to capture defects, etc. There may be several tools used in testing like defect management tools, configuration management tools, regression testing tools, etc.

Rework If 'check processes' find that 'do processes' are not able to achieve the objectives defined for them, it must follow the route of rework. This is a rework of 'do process' and not of work product under testing. This ensures that all incapable processes are captured so that these can be taken for improvement.

There may be two more criteria in the work bench, viz. suspension criteria for the workbench, and resumption criteria for the workbench guided by organisational policies and standards. Suspension criteria

defines when the testing process needs to be suspended or halted, whereas resumption criteria defines when it can be restarted after such halt or suspension. If there are some major problems in inputs or standards and tools required by the workbench, 'do/check process' may be suspended. When such problems are resolved, the processes may be restarted.

Testing process may be defined as a process used to verify and validate that the system structurally and functionally behaves correctly as defined by expected result. Components of testing process may include the following.

- Giving inputs (program code) to tester from previous work bench
- Performing work (execute testing) using tools and standards
- Following a process of doing and checking whether test process is capable or not
- Produce output (test results and test log) which may act as an input to the next work bench

Check process must work to ensure that results meet specifications and standards, and also that the test process is followed correctly. If no problem is found in the test process, one can release the output in terms of test results. If problems are found in test process, it may need rework. Figure 3.2 shows a schematic diagram of a workbench.

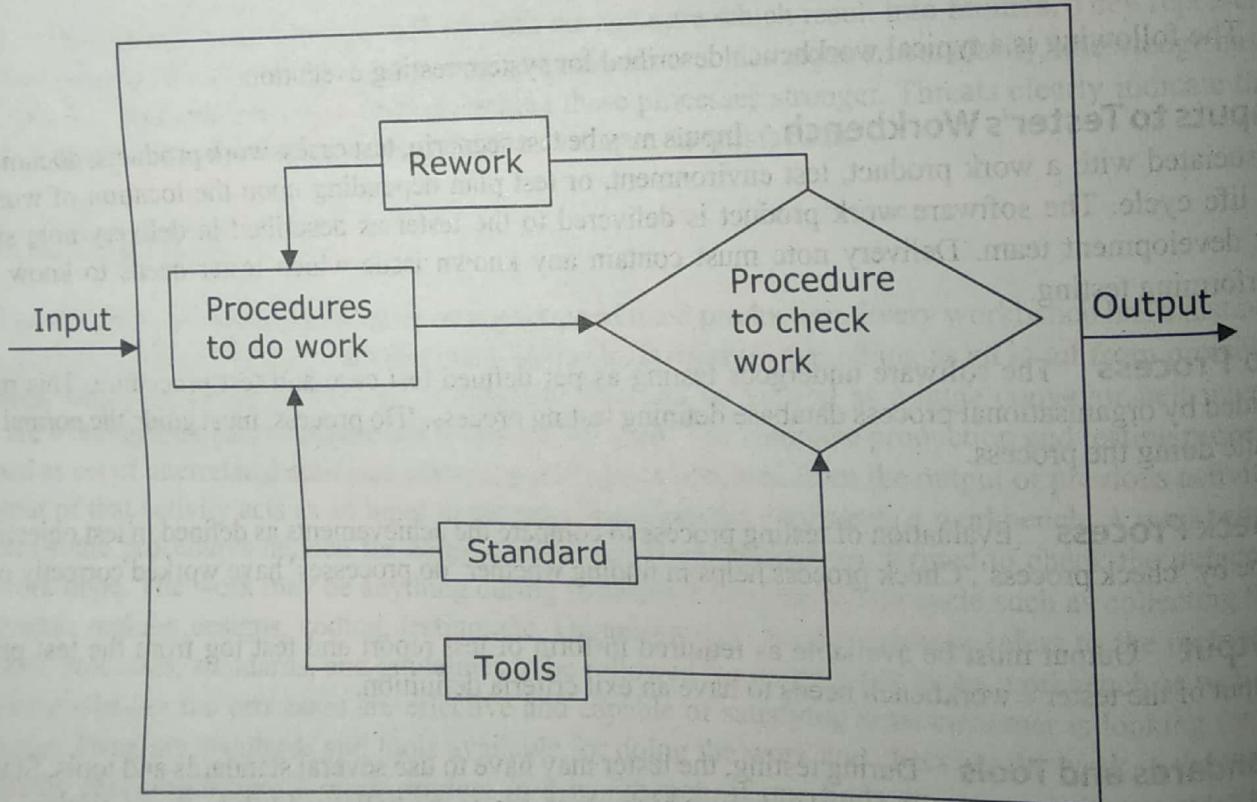


Fig. 3.2

Typical workbench

3.10 IMPORTANT FEATURES OF TESTING PROCESS

Testing is characterised by some special features, as given below.

Testing is a Destructive Process, But it is Constructive Destruction Testing involves a systematic destruction of a product with the intent to find the defects so that these would be fixed before the

product is given to the customer. While executing tests, a tester goes about testing an application using some valid/invalid inputs to find the response of software to each of these conditions. The ability of the tester to break the application can make him successful. For devising negative testing, one needs to build the scenario with destructive mentality.

Testing Needs a Sadistic Approach with a Consideration That There is a Defect A tester cannot certify that a work product is defect free. He needs to go through the software work product and hunt for defects. If the defect is not found, it directly means that there is some problem with testing process. Defect-free product does not exist. Testers are expected to identify the risks to the final users and test the product accordingly.

If the Test Does Not Detect a Defect Present in the System, it is an Unsuccessful Test Success of testing lies in its ability to find defects or threats and weaker areas of software work product and the processes supporting these areas. Testers who cannot find defects are unsuccessful testers. Testing is considered as an investment only when it reduces the probability that software may fail at customer end.

A Test That Detects a Defect is a Valuable Investment for Development As Well As Customer, it Helps in Improving a Product The root cause analysis of defects can show where the application and process can be and needs to be improved. It also helps in identifying the weaker areas of the processes used for developing software. Weaker areas are analysed to find the process lacunae and take actions on these areas and strengthen them. Thus, defect finding helps in improving processes which can result in increasing customer satisfaction. Testing with an intention to find and fix the problems in processes can be considered as an investment by customer/organisation.

Some organisations use final black box testing as an acceptance testing for the application. If no defect is found in system testing, the software is delivered to the customer. If the sole purpose of testing is to validate specifications implemented then,

- Testing is an unnecessary/unproductive activity as it does not consider invalid scenarios. No amount of testing can certify that there is no defect in the product. There must be a probability of finding the defect in testing.
- Testing is designed to compensate for ineffective software development process, it cannot give results. The defect indicates a process deficiency and it must be fixed by improving the processes. Testing cannot improve the quality of product. Defect is a symptom of something failing and one must try to fix the root cause and not only the symptom.
- If development methodology designed and implemented by a team is not correct, testing cannot compensate for it. Testing is not meant to certify the work products but to find the defects.
- Testing is a separate discipline and may get affected by as well as affects software development processes. Better processes of development and testing can reduce the chances of failure of product at customer place and subsequent customer complaints.

It is Risky to Develop Software and Not to Test it Before Delivery Not providing sufficient resources, time and support for testing activities is a common scenario across the industry. There is a belief that less-tested software means less defects, less rework, less scrap, and less-corrective actions which means higher profits. But this may result into customer dissatisfaction as the defects will get exposed at customer site. Reducing the coverage of testing is another risk associated with software. Software testing must give a desired level of confidence to users that system will not fail. Less testing reduces this confidence level and increases a probability of failure at customer site.

With High Pressure to Deliver Software As Quickly As Possible, Test Process Must Provide Maximum Value in Shortest Timeframe This approach is adopted in test strategy designing where the efficiency and effectiveness of testing is defined. Generally, test cases are categorised into installation testing, smoke testing, and sanity testing before going into further detailed testing. Test cases which represent scenarios that may occur with higher probability can help in reducing probabilities of failure in production environment. An organisation may define such test cases with probability aspect such as high, medium, and low or allocate the numbers indicating priority of execution. Probability of occurrence of a defect may not have any relationship with severity as severity talks about type of failures.

Testing is no longer an after-programming evaluation to certify that the software works, but supposed to indicate the confidence level that product will work at customer site. Testing can give the SWOT analysis of development process. Thus, testing is adjunct to software development life cycle.

Testing starts at the proposal level and ends only when software application is finally accepted by user/customer. Every stage of development and every work product must go through stages of review and formal approval to ensure that it can go to the next stage. But it is a key to ensure quality at each work product and each phase of software development life cycle.

Highest Payback Comes from Detecting Defect Early in Software Development Life Cycle and Preventing Defect Leakage/Defect Migration from One Phase to Another

Defects are the problems or something wrong happening in software as well as the development process used for making software. Every defect indicates failure of the process at some place or another. It is always economical to fix the defects as and when they appear, and conduct an analysis to find the root causes of the defects rather than waiting till it hits the software product and user, again and again. It is always beneficial to do a root cause analysis and fix the problem areas at the earliest possible time. The investment in testing can be worthwhile only if it is capable of finding defects as soon as it is introduced in the work product and also can prevent any potential defect from getting introduced. Defect, if not corrected in the phase where it is introduced, leaks to the next stage and creates a larger problem. This is termed 'phase contamination'. Defect keeps on migrating from one phase to next phase, till it hits back the users at some point of time.

Organisation's Aim Must Be Defect Prevention Rather Than Finding and Fixing a Defect The major misconception about testing is that it is considered as a fault-finding mission. Instead, it must be viewed as defect-prevention mission to avoid critical problems in software development process by initiating actions to prevent any recurrence of problems. Finding and fixing the problem is not a good approach as the basic cause of defect is never addressed. It needs analysis of root causes, and defect prevention mechanism—that is installed and operational—to prevent recurrence as well as removal of potential problems.

3.11 MISCONCEPTIONS ABOUT TESTING

At many places, software testing is termed 'quality assurance (QA)' activity. In reality testing is a quality control (QC) activity. There are many other misconceptions about software testing, as listed below.

Anyone Can Do Testing, and No Special Skills Are Required for Testing Many organisations have an approach that anyone can be put in testing. They give the task of testing to developers on the bench or people asking for 'light duty'. Many people involved in testing do not have any experience in testing or in the domain in which testing is done. Test planning, test case writing, and test data definition using

different methodologies may not be possible with unskilled people. If the organisation considers investment in special skills as a waste of time and money, it may result in disaster for customer/user.

Testers Can Test Quality of Product at the End of Development Process This is a typical approach where system testing or acceptance testing is considered as qualification testing for software. Few test cases out of infinite set of possibilities are used for certifying whether the software application works or not. The customer may be dissatisfied as the application does not perform well as per his expectations. Sometimes, the defects remain hidden for an entire life cycle of software without anybody knowing that there was a defect.

Defects Found in Testing Are Blamed on Developers Another common misconception regarding defects found in testing is blaming developers for defects. Though two-third of defects are due to wrong requirements, yet developers are mostly blamed for defects in software development. Also, some surveys indicate that most of the defects can be attributed to faulty development processes. Developers are responsible for converting the design into code by using the standards or guidelines available. Ensuring good inputs to developer's workbench is a responsibility of the management.

Defects Found By Customer Are Blamed on Tester Testers perform testing by executing few test cases and try to cover some part of software program to check whether the program performs as intended or not. No one can say that testing can ensure 100% coverage. If no defect is found during testing, it does not indicate that the software program is defect free. There may be few defects left in the product which can be found only in real life. One must do a root cause analysis of the defects found, and try to learn from the experiences to ensure that a better product is produced and similar defects do not recur next time. But no one can blame testers for defects reported by customer.

3.12 PRINCIPLES OF SOFTWARE TESTING

Testing needs to be performed according to processes defined for it. It needs skilled and trained people to break the application and demonstrate the problems or defects in the software product. Some key points in software testing are as follows.

Programmers/Team Must Avoid Testing Their Own Work Products Everybody is in love with the work product he/she has made. Also, the approach of an individual remains the same and hence, approach-related defects cannot be found in self review or self testing. A second opinion is essential which can add value to a work product. Though self review is a good tool for retrospection, yet it has many limitations.

Thoroughly Inspect Results of Each Test to Find Potential Improvements Test results show possibilities of weaker areas in the work product and the problems associated with the processes used for developing a work product. The defects found in test log do not form an exclusive list of all problems with the application, but indicate the areas where development team and management must perform a root cause analysis. Corrective actions are to be planned and executed to prevent any possible recurrence of similar defect and make software better. Defects indicate process failures.

Initiate Actions for Correction, Corrective Action and Preventive Actions Defect identification, fixing and initiation of action to prevent further problems are the natural ways of making better

products and improve processes. Corrections of the defect are done by the developers. But one must ensure that corrective and preventive actions are initiated for making better products again and again.

Establishing that a program does what it is supposed to do, is not even half of the battle and rather easier one than establishing that program does not do what it is not supposed to do. This is negative testing driven by risk assessment for the final users. Roughly, testing may involve 5% positive testing and 95% negative testing.

3.13 SALIENT FEATURES OF GOOD TESTING

Defects indicate the quality of software under testing, development and test processes used for making it. Testing is a life-cycle activity where the testers take part in testing right from proposal stage till the application is finally accepted by the customer/user. Good software testing involves testing of the following.

Capturing User Requirements The requirements defined by the users or customer as well as some implied requirements (which are intended by the users but not put in words) represent the foundation on which software is built. Intended requirements are to be analysed and documented by testers so that they can write the test scenario and test cases for these requirements. User requirements involve technical, economical, legal, operational, and system requirements. Generally, a user is able to specify only functional and non-functional requirements which form a part of operational requirements and legal requirements but definition of other requirements is a responsibility of development organisation.

Capturing User Needs User needs may be different from user requirements specified in software requirement specifications. User needs may include present and future requirements, and other requirements which may include process requirements (including definition of deliverables) and implied requirements. Elicitation of requirements is to be done by the development organisation to understand and interpret the requirements.

Design Objectives Design objectives state why a particular approach has been selected for building software. The selection process indicates the reasons and criteria framework used for development and testing. How an applications functional requirements, user interface requirements, performance requirements and other requirements are interpreted in design, and how they can be achieved in further development must be defined in an approach document.

User Interfaces User interfaces are the ways in which the user interacts with the system. This includes screens and other-ways of communication with the system as well as displays and reports generated by the system. User interfaces should be simple, so that the user can understand what he is supposed to do and what the system is doing. Users ability to interact with the system, receive error messages, and act according to instructions given is defined in the user interfaces.

Internal Structures Internal structures are mainly guided by software designs and guidelines or standards used for designing and development. Internal structures may be defined by development organisation or sometimes defined by customer. It may talk about reusability, nesting, etc. to analyse the software product as per standards or guidelines. It may include commenting standards to be used for better maintenance. Every approach may have some advantages/disadvantages, and one needs to weigh the benefits and costs associated with them to get a better solution.

Execution of Code Testing is execution of a work product to ensure that it works as intended by customer or user, and is prevented from any probable misuse or risk of failure. Execution can only prove that application module, and program work correctly as defined in requirements and interpreted in design. Negative testing shows that application does not do anything which is detrimental to the usage of a software product.

3.14 TEST POLICY

Test policy is generally defined by the senior management covering all aspects of testing. It decides the framework of testing and its status in overall mission of achieving customer satisfaction. For project organisations, test policy may be defined by the client while for product organisation, it is decided by the senior management.

3.15 TEST STRATEGY OR TEST APPROACH

Test strategy defines the action part of test policy. It defines the ways and means to achieve the test policy. Generally, there is a single test policy at organisation level for product organisations while test strategy may differ from product to product, customer to customer and time to time. Some of the examples of test strategy may be as follows.

- Definition of coverage like requirement coverage or functional coverage or feature coverage defined for particular product, project and customer.
- Level of testing, starting from requirements and going upto acceptance phases of the product.
- How much testing would be done manually and what can be automated?
- Number of developers to testers.

3.16 TEST PLANNING

Test planning is the first activity of test team. If one does not plan for testing, then he/she is planning for failure. Test plans are intended to plan for testing throughout software development life cycle. Test plans are defined in the framework created by test strategy and established by test policy. Test plans are made for execution which involves various stages of software testing associated with software development life cycle. Test plan should be realistic and talk about the limitations and constraints of testing. It should talk about the risks and assumptions done during testing.

Plan Testing Efforts Adequately with an Assumption That Defects Are There All software products have defects. Test planning should know the number of defects it is intending to find by executing the given test plan. Test plan should cover the number of iterations required for software testing to give adequate confidence required by customer (to show that software will be usable). Defect found in testing is an investment in terms of process improvement opportunity. Test plan is successful if intended number of defects are found.

If Defects Are Not Found, It Is Failure of Testing Activity There are many defects in software. If no (less) defects are found in testing, then it does not mean that there are no (less) defects in the product. It may mean that the test cases are not complete or adequate, or the test data is not effective in locating defects in the software product. Testing is intended to find defects. If defects are not found, the testing process may be considered as defective. Every defect found is an investment as it reduces a probability of any defect which customer may find. If more number of defects are found, it means the development process is problematic.

Successful Tester Is Not One Who Appreciates Development But One Who Finds Defect in the Product Success of testing is in finding a defect and not certifying that application or development process is good. Successful testers can find more defects with higher probabilities of occurrences

and higher severities of failure. Tester should find defects, which have a probability of affecting common users and thus contribute to a successful application.

Testing Is Not a Formality to Be Completed at the End of Development Cycle Testing is not a certifying process. It is a life-cycle activity and should not be the last part of a development life cycle before giving the application to customer/user. Acceptance testing and system testing are the integral parts of software development where the certification of application is done by the customer but complete test cycle is much more than black box testing.

Software testing includes the following.

- Verification or checking whether a right process is followed or not during development life cycle.
- Validation or checking whether a right product is made or not as per customer's need.

Some differences between verification and validation are shown in Table 3.4.

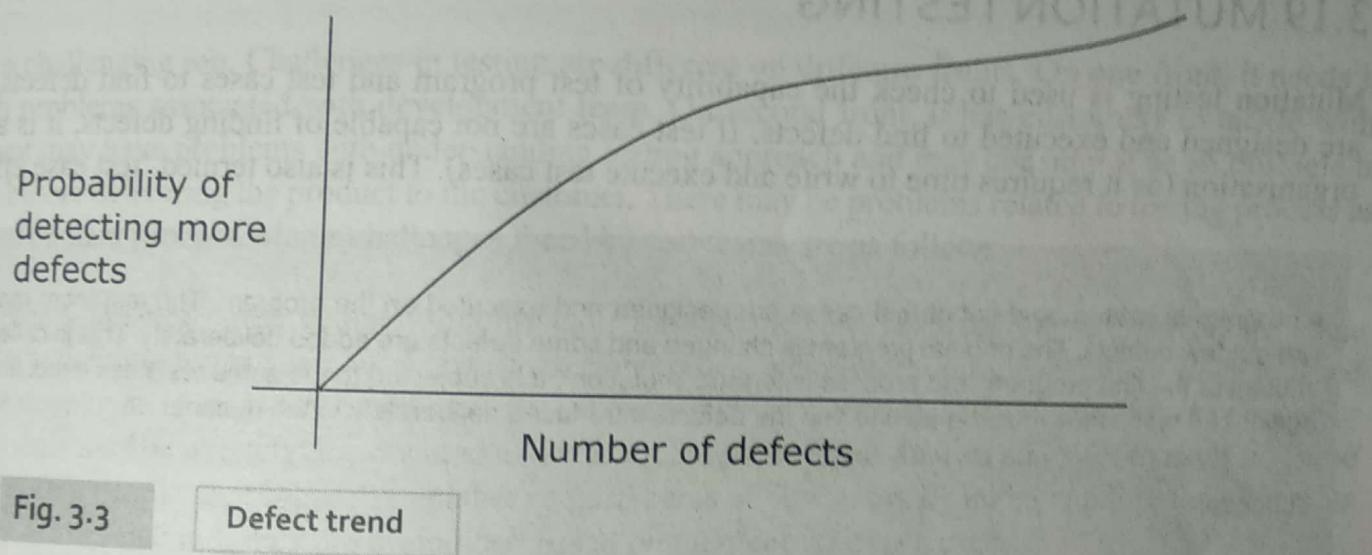
Table 3.4

Difference between Verification and Validation

Verification	Validation
Verification is an activity where we check the work products with reference to standards, guidelines, and procedures.	Validation is an activity to find whether the software achieves whatever is defined by requirements.
Verification is prevention based. It tries to check the process adherence.	Validation is detection based. It checks the product attributes.
Verification talks about a process, standard and guidelines.	Validation talks about the product.
Verification is also termed 'white box testing' or 'static testing' as the work product undergoes a review.	Validation is also termed 'black box testing' or 'dynamic testing' as work product is executed.
Verification may be based on opinion of reviewer and may change from person to person.	Validation is based on facts and is generally independent of a person.
Verification can find about 60% of the defects.	Validation can find about 30% of the defects.
Verification involves the following.	Validation involves all kinds of testing.
<ul style="list-style-type: none"> • reviews • walkthroughs • inspection • audits 	<ul style="list-style-type: none"> • system testing • user interface testing • stress testing
Verification can give the following.	Validation can give the following.
<ul style="list-style-type: none"> • statement coverage • decision coverage • path coverage 	<ul style="list-style-type: none"> • requirement coverage • feature coverage • functionality coverage
Some parts of software can undergo verification only, as given below.	Some characteristics of software can be proven by validation only, as given below.
<ul style="list-style-type: none"> • coding guidelines • nesting • commenting • declaration of variables 	<ul style="list-style-type: none"> • stress testing • performance testing • volume testing • security testing

3.17 TESTING PROCESS AND NUMBER OF DEFECTS FOUND IN TESTING

Testing is intended to find more number of defects. Generally, it is believed that there are fixed number of defects in a product and as testing finds more defects, chances of the customer finding the defect will reduce. Actually the scenario is reverse. As we find more and more defects in a product, there is a probability of finding some more defects. This is based on the principle that every application has defects and every test team has some efficiency of finding defects. It is governed by the test team's defect-finding ability. Let us say, the organisational statistics shows that after considerable testing and use of application by a user, number of defects found is three per KLOC; test planning must intend to find three defects per KLOC for the program under testing. The number of defects found after considerable testing will always indicate possibilities of existing number of defects. Figure 3.3 shows a relationship between number of defects found and probability of finding more defects.



3.18 TEST TEAM EFFICIENCY

Test team efficiency is a very important aspect for development team and management. If test team is very efficient in finding defects, less iterations of testing are required. On the other hand, if development team is less efficient in fixing defects, more iterations of testing and defect fixing may be required.

The test team has some level of efficiency of finding defects. Suppose the application has 100 defects and the test team has an efficiency of 90%, then it will be able to find 90 defects. Thus, if the test team finds 180 defects (considering same efficiency), it means that there are 200 defects in the software product.

Every test manager must be aware of the efficiency of a test team that he/she is working with. Often, test managers and project managers try to assess the test team efficiency at some frequency. The process may be as defined below.

Ideally, test team efficiency must be 100% but in reality, it may not be possible to have test teams with efficiency of 100%. It must be very close to 100% in order to represent a good test team. As it deviates away from 100%, the test team becomes more and more unreliable. Test team efficiency is dependent on organisation culture and may not be improved easily unless organisation makes some deliberate efforts.

The development team introduces some defects in a software product and gives it to the test team. The test team completes testing iterations as planned and gives the number of defects found. The development team then analyses the defects found by the test team to understand how many defects have been found by the test team. The ratio gives test team efficiency.

Suppose,

Defects deliberately introduced by development = X

Total defects found by testing team = Y

Defects found by testing team but not belonging to defects deliberately introduced by development = Z

The ratio of $(Y - Z)/X$ will give the test team efficiency.

Solved Example 3.1

3.19 MUTATION TESTING

Mutation testing is used to check the capability of test program and test cases to find defects. Test cases are designed and executed to find defects. If test cases are not capable of finding defects, it is a loss for an organisation (as it requires time to write and execute test cases). This is also termed 'test case efficiency'.

A program is written, and set of test cases are designed and executed on the program. The test team may find out few defects. The original program is changed and some defects are added deliberately. This is called 'mutant of the first program' and process is termed 'mutation'. It is subjected to the same test case execution again. The test cases must be able to find the defects introduced deliberately in the mutant.

Suppose,

Defects deliberately introduced by development = X

Defects found by test cases in original program = Y

Defects found by test cases in mutant = Z

The ratio of $(Z - Y)/X$ will give the test case efficiency. Theoretically, it must be 100%. But it may not be exactly 100% due to the following reasons.

Solved Example 3.2

3.19.1 REASONS FOR DEVIATION OF TEST TEAM EFFICIENCY FROM 100% FOR TEST TEAM AS WELL AS MUTATION ANALYSIS

Though desirable, it is very difficult to get a test team with 100% efficiency of finding defects and test cases with 100% efficiency of finding defects. Some of the reasons for deviation are listed below.

- *Camouflage Effect* It may be possible that one defect may camouflage another defect, and the tester may not be able to see that defect, or test case may not be able to locate the hidden defect. It is called 'camouflage effect' or 'compensating defects' as two defects compensate each other. Thus, defect introduced by developer may not be seen by the tester while executing a test case.

- **Cascading Effect** It may be possible that due to existence of a certain defect, few more defects are introduced or seen by the tester. Though there is no problem in the modification, defects are seen due to cascading effect of one defect. Thus, defects not introduced by developer may be seen by tester while executing a test case.
- **Coverage Effect** It is understood that nobody can test 100%, and there may be few lines of code or few combinations which are not tested at all due to some reasons. If defect is introduced in such a part which is not executed by given set of test cases, then tester may not be able to find the defect.
- **Redundant Code** There may be parts of code, which may not get executed under any condition, as the conditions may be impossible to occur, or some other conditions may take precedence over it. If developer introduces a defect in such parts, testers will not be able to find the defect as that part of code will never get executed.

3.20 CHALLENGES IN TESTING

Testing is a challenging job. Challenges in testing are different on different fronts. On one front, it needs to tackle with problems associated with development team. On second front, it has customers to tackle with. Management may have problems with understanding testing approach and may consider it as an obstacle to be crossed before delivering the product to the customer. There may be problems related to testing process as well as development process. Major challenges faced by test teams are as follows.

- Requirements are not clear, complete, consistent, measurable and testable. These may create some problems in defining test scenario and test cases. Sometimes, a configuration management issue is faced when the development team makes changes in requirements but test team is not aware of these changes.
- Requirements may be wrongly documented and interpreted by business analyst and system analyst. These knowledgeable people are supposed to gather requirements of customers by understanding their business workflow. But sometimes, they are prejudiced based on their earlier experiences.
- Code logic may be difficult to capture. Often, testers are not able to understand the code due to lack of technical knowledge. On the other hand, sometimes, testers do not have access to code files.
- Error handling may be difficult to capture. There are many combinations of errors, and various error messages and controls are required such as detective controls, corrective controls, suggestive controls, and preventive controls.

3.20.1 OTHER CHALLENGES IN TESTING

More bugs found in software introduce additional iterations of fixing defects, retesting, and regression testing of a product. It means more efforts, delayed shipment to customer and late payment receipt by organisation. Often, testers are blamed for delays in delivery.

- Badly written code introduces many defects. Code may not be readable, maintainable, optimisable, and may create problems in future. Defects may not be fixed correctly, and fixing of defects may introduce more defects called 'regression defects'.
- Bad architecture of software cannot implement good requirement statement. What developers do in reality is that they implement the design and not the requirements. Bad architecture creates complex code and adds many defects to the software product.

- Testing is considered as a negative activity. Often, testers need to reject builds if problems are found in it which does not satisfy exit criteria. It is a difficult situation as organisational fund flow may be depending upon successful delivery of system, and it gets affected due to such rejection.
- Testers find themselves in lose-lose situation in testing. If more defects are found, application delivery is delayed and testers are blamed for such delay. On the other hand, if testing is not done properly, customer complaints are possible and again testers are held responsible for it.

3.21 TEST TEAM APPROACH

Type of the organisation and type of the product being developed define a test team. There may or may not be a separate team doing testing if management does not recognise its importance, or the application under development demands this scenario. There are four approaches of software testing team.

3.21.1 LOCATION OF TEST TEAMS IN AN ORGANISATION

Generally, test team is located in an organisation as per testing policy. It may vary from organisation to organisation, project to project and customer to customer. Following are some of the approaches used for locating test team organisationally.

Independent Test Team Independent test team may not be reporting to development group at all, and are independent of development activities. They may be reporting independently to senior management or customer. Presence of test manager is essential to lead the test team. Such test teams may be shown as in Fig. 3.4.

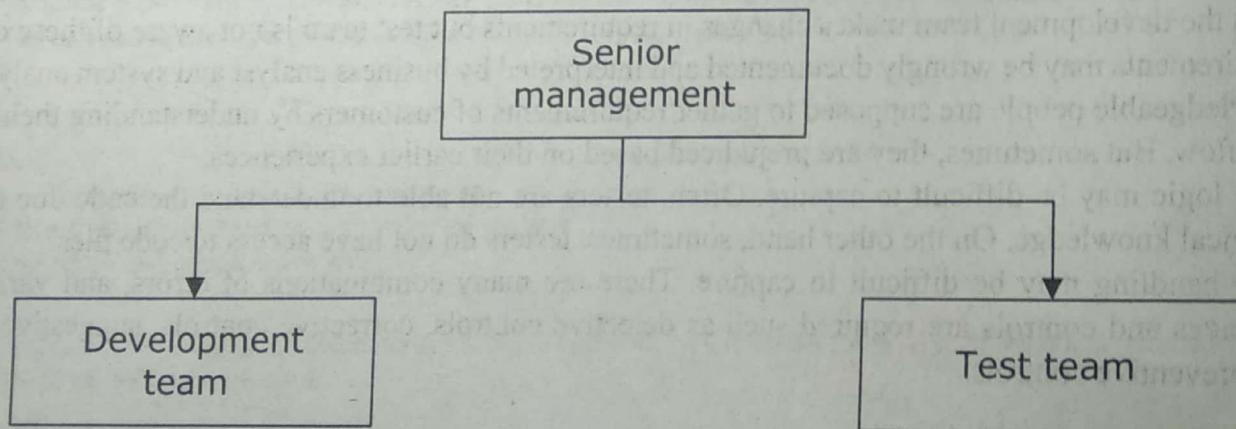


Fig. 3.4

Organisational structure of test team independent of development team

Advantages of Independent Test Team

- The test team is not under delivery pressure. They can take sufficient time to execute complete testing as per definitions of iterations, coverage, etc
- Test team is not under pressure of 'not finding' a defect. They are considered as the certifiers of a product and must be able to find every conceivable fault in the product before delivery.
- Independent view about a product is obtained as thought process of developers and testers may be completely different.

- Expert guidance and mentoring required by test team for doing effective testing may be available in the form of a test manager.

Disadvantages of Independent Test Team

- There is always 'us' vs 'them' mentality between development team and test team. Team synergy can be lost as developers take pride in what they develop while testers try to break the system.
- Testers may not get a good understanding of development process as development team tries to hide the process lacunae from them. Testers are treated as outsiders.
- Sometimes, management may be inclined excessively towards development team or test team, and the other team may feel that they have no value in an organisation.

Test Team Reporting to Development Manager

If the test team is reporting to development manager, then they can be involved from the start of project till the project is finally closed. Such test team may be as shown in Fig. 3.5.

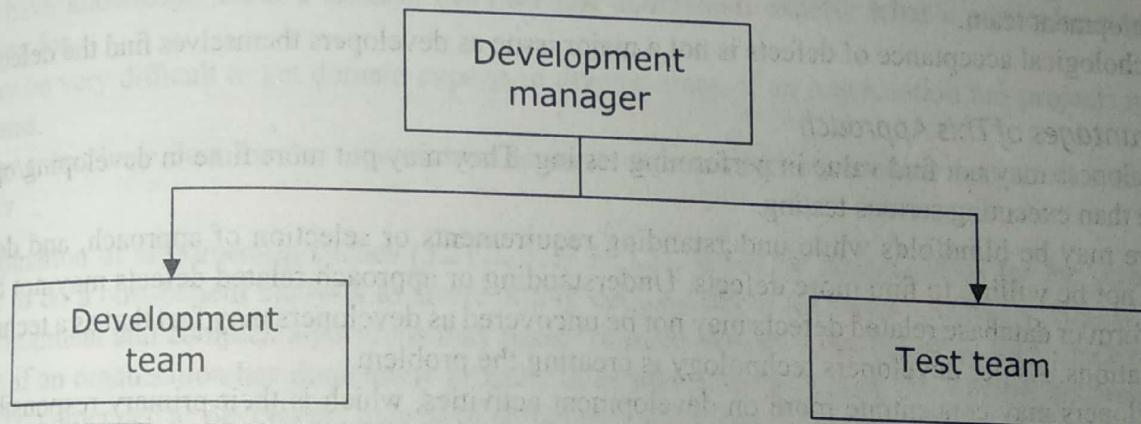


Fig. 3.5

Organisational structure of test team reporting to development manager

Advantages of Test Team Reporting to Development Manager

- There is a better cooperation between development team and test team as both are part of the same team.
- Test team can be involved in development and verification/validation activities from the start of the project. It gives them good understanding of requirements.
- Testers may get a good understanding of development process and can help in process improvement.

Disadvantages of Test Team Reporting to Development Manager

- Expert advice in the form of test manager may not be available to testers. In case testers need some guidance or mentoring, they may have to rely on an external person.
- Sometimes, development managers are more inclined towards development team. Defects found by test team are considered as hurdles in delivery process.
- Often, testers start perceiving the product from developers angle and their defect finding ability reduces.

Matrix Organisation In case of matrix organisation, an effort is made to achieve the advantages of both approaches, and get rid of disadvantages of both approaches. Test team may be reporting functionally to the development manager while administratively it reports to the test manager.

3.21.2 DEVELOPERS BECOMING TESTERS

Sometimes, those who work as developers in initial stages of development life cycle take the role of testers when the latter stages of life cycle are executed. This is a common practice while working with simple software which does not need very structured testing. Developers becoming testers can be suitable when the application is technologically heavy.

Advantages of This Approach

- Developers do not need another knowledge transfer while working as a tester. The knowledge transfer that they received at initial stages of development can be used by them in both roles.
- Developers have better understanding of detail design and coding, and can test the application easily.
- For automation, some amount of development skill is required in writing the automation scripts. Developers can adapt themselves in automation testing better as they have the ability to create code.
- It is less costly as there is no separate test team. It provides staff balancing to some extent. Initially, the development team is large but as SDLC comes to an end, the test team becomes larger than the development team.
- Psychological acceptance of defects is not a major issue as developers themselves find the defects.

Disadvantages of This Approach

- Developers may not find value in performing testing. They may put more time in developing/optimising code than executing serious testing.
- There may be blindfolds while understanding requirements or selection of approach, and developers may not be willing to find more defects. Understanding or approach related defects may not be found. Platform or database related defects may not be uncovered as developers may feel that as a technological limitations. As per developers technology is creating the problem.
- Developers may concentrate more on development activities, which is their primary responsibility and may neglect testing activities.
- Development needs more of a creation skill while testing needs more of a destruction skill. It is difficult to have a team having both skills at a time.

3.21.3 INDEPENDENT TESTING TEAM

An organisation may create a separate testing team with independent responsibility of testing. The team would have people having sufficient knowledge and ability to test the software.

Advantages of This Approach

- Separate test team is supposed to concentrate more on test planning, test strategies and approach, creating test artifacts, etc.
- There is independent view about the work products derived from requirement statement.
- Special skills required for doing special tests may be available in such independent teams.
- 'Testers working for customer' can be seen in such environment.

Disadvantages of This Approach

- Separate team means additional cost for an organisation.
- Test team needs ramping up and knowledge transfer, similar to a development team.
- An organisation may have to check for rivalries between development team and test team.

3.21.4 DOMAIN EXPERTS DOING SOFTWARE TESTING

An organisation may employ domain experts for doing testing. Generally, this approach is very successful in system testing and acceptance testing where domain specific testing is required. Domain experts may use their expertise on the subject matter for performing such type of testing.

Advantages of This Approach

- 'Fitness for use' can be tested in this approach where actual user's perspective may be obtained. Domain experts will be testing software from user's perspective.
- Domain experts may provide facilitation to developers about defects and customer expectations, and may be able to interpret requirements in the correct context.
- Domain experts understand the scenario faced by actual users and hence, their testing is realistic.

Disadvantages of This Approach

- Domain experts may have prejudices about the domain which may reflect in testing. Domain experts may have knowledge about a domain but may not understand exactly what a particular customer is looking for.
- It may be very difficult to get domain experts in diverse areas, if an organisation has projects in diverse domains.
- It may mean huge cost for the organisation as these experts cost much more than normal developers/testers.

Combination of all three approaches (3.21.2, 3.21.3, 3.21.4) can be advantageous for the organisation. One has to do a cost-benefit analysis to arrive at any decision about test team formation. Highly complex user environment and complex algorithms may make 'domain experts doing testing' more effective. On the contrary, if an organisation has done many projects in similar domain in past, 'developers becoming tester' may be recommended as developers may have sufficient knowledge about the subject.

In addition to a test team, there are many other agencies involved in software testing as per phases of software development.

Customer/User Customer or users generally do acceptance testing to declare formal acceptance/rejection/changes in requirements for the product. Customer perspective is most important in software acceptance. Organisations creating prototype may rely on customer approval of prototype.

Developers Developers do unit testing before the units are integrated. Generally, units require stubs and drivers for testing, and developers can create the same. Sometimes, integration testing is also done by developers if it needs stubs and drivers.

Tester Testers perform module, integration, and system testing as independent testing. They may oversee acceptance testing. Tester's view of system testing is very close to user's view while accepting software.

Information System Management Information system management may do testing related to security and operability of system. They would provide the specialised skills needed for this type of testing.

Senior Management/Auditors Senior management or auditors appointed by senior management such as Software Quality Assurance (SQA) perform predelivery audit, smoke testing, and sample testing to ensure that proper product is delivered to the customer.

3.22 PROCESS PROBLEMS FACED BY TESTING

'Q' organisations consider that defects in the product are due to incorrect processes. In general, it is believed that incorrect processes cause majority (about 90%) of the working problems. Defects are introduced in software due to incapable processes of development and testing. Software testing is also a process, and prone to introduce defects in the system. If the process of software testing is faulty, it gives problems in terms of defects not found during testing but found by customer, or wrong defects found which are either 'not a defect', 'duplicate', 'cannot be reproduced' or 'out of scope' type of defects. The basic constituents of processes are people, material, machines and methods.

People Many people are involved in software development and testing, such as customer/user specifying requirements; business analysts/system analysts documenting requirements; test managers or test leads defining test plans and test artifacts; and testers defining test scenarios, test cases, and test data. There is a possibility that at few instances some personal attributes and capabilities may create problems in development and testing. Proper skill sets such as domain knowledge and knowledge about development and testing process may not be available.

Material Testers need requirement documents, development standards and test standards, guidelines, and other material which add to their knowledge about a prospective system. These documents may not be available, or may not be clear and complete. Similarly, other documents which act as a framework for testing such as test plans, project plan, and organisational process documents may be faulty. Test tools and defect tracking tools may not be available. All of these may be responsible for introducing defects in the product.

Machines Testers try to build real-life scenarios using various machines, simulators and environmental factors. These may include computers, hardware, software, and printers. The scenarios may or may not represent real-life conditions. There may be problems induced due to wrong environmental configurations, usage of wrong tool, etc.

Methods Methods for doing test planning, risk analysis, defining test scenarios, test cases, and test data may not be proper. These methods undergo revisions and updatations as the organisation matures.

Economics of Testing As one progresses in testing, more and more defects are uncovered, and probability of customer facing a problem reduces while the cost of testing goes up. The cost of customer dissatisfaction is inversely proportional to testing efforts. It means more investment in testing efforts reduces the cost of customer unhappiness. On the other hand, the cost of testing increases exponentially. If we plot both curves, then at some point, the two curves intersect each other. This point shows optimum testing point. Area before this point represents an area under testing where defective product goes to customer and customer dissatisfaction cost is higher than cost of testing, while area after this point represents an area of over testing where cost of customer dissatisfaction is less than cost of testing.

Cost of testing curve is guided by the following.

- Defect finding ability of testing team (test team efficiency)
- Defect fixing ability of development team (defect fixing efficiency)
- Defect introduction index of development team which talks about regression defects getting introduced due to fixation of some defects discovered during testing

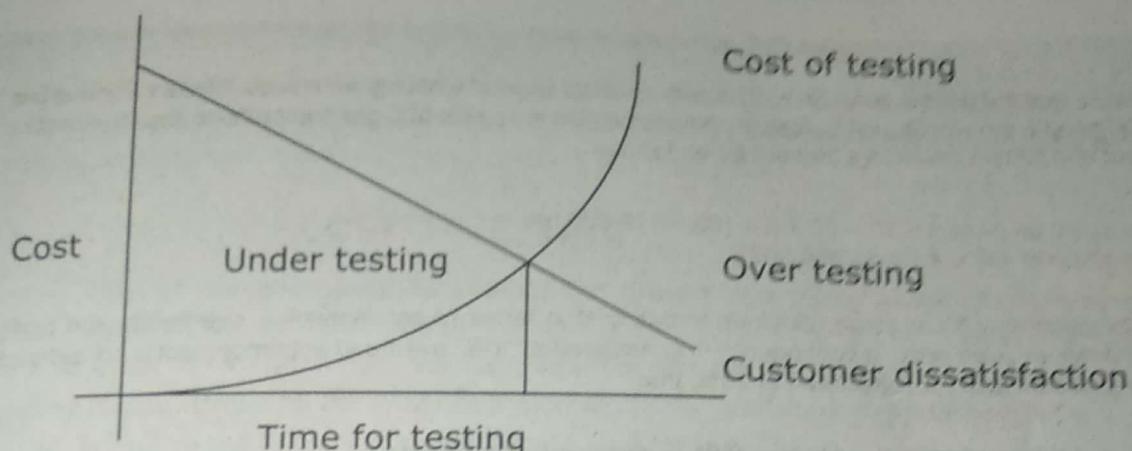


Fig. 3.6

Cost of testing and cost of customer dissatisfaction

- If test team efficiency and defect fixing efficiency are 100%, and defect introduction index is zero, we need only two iterations of testing. As it goes away from ideal numbers, number of iterations increase exponentially.

Cost of customer dissatisfaction is guided by the following aspects.

- Cost of customer dissatisfaction mainly depends upon customer-supplier relationship. If an organisation has done several projects which were very successful, then the customer may not mind if some defects are there in the current project. Customer dissatisfaction curve tends to be parallel to 'Y' axis. On the other hand, if the customer is very finicky about defects and past performance of an organisation is not good, it may tend to be parallel to 'X' axis.
- Cost of customer dissatisfaction also depends on the type of product and its mission criticality to the customer. Customer may not like any problem in high mission-critical software while he may accept certain problems in other types of software.

3.23 COST ASPECT OF TESTING

As seen earlier, cost of quality includes cost of prevention, cost of appraisal and cost of failure. Testing may take some portion of each of these costs. It is believed that cost of quality is about 50% of the cost of product. Testing is a costly affair and an organisation must try to reduce the cost of testing to the maximum extent possible.

There is a famous concept of efforts conversion into cost in case of software development, as effort is the major component of total cost. This may be done by standard costing method or marginal costing method as per organisation's process definition. Efforts spent by the organisation in developing and testing of an application are converted at some predefined rates to arrive at the total cost of a product. Sometimes, the cost of resource varies as per the role played by a person. For example, a project manager may get more rate than a developer, or an architect may get more billing than a tester.



Solved Example 3.3

Let us suppose that the project duration is 10 months with 22 days of working per month, 8 hours working per day and 100 people are working on it. Also, if conversion rate is say Rs 500 per person hour, then the cost of development and testing taken together will be as follows.

$$\text{Total efforts spent on project} = 10 \times 22 \times 8 \times 100 = 176,000 \text{ hrs}$$

$$\text{Total cost} = 176,000 \times 500 = \text{Rs } 88,000,000$$

An organisation may have some additions to this cost in terms of contingencies, overheads and profit expected to arrive at sales price. If contingency is considered at 10%, overhead apportionment is considered at 10% and expected profit is considered at 20%, then

$$\text{Sales price would be} = 8,800,000 \times 110\% \times 110\% \times 120\% = \text{Rs } 127,776,000$$

It is a very rare scenario that a project will have 100 people working for all 10 months for the development project. Generally, development projects never have the same number of resources throughout the life cycle. Initially, it may need less number of people and as one passes through different phases of development, number of resources required increases exponentially. Once the peak activities are over, number of resources required goes down.

The same cycle is followed by testing resource requirements. As the testing phases progress, number of resources required increases exponentially. Once the peak activities are over, number of test resources goes down. Thus, for development project, costing is always dynamic.

In case of maintenance or production support type of work, number of resources remains fairly constant over a long-time horizon. Figure 3.7 indicates resources required for a development project.

Number of resources for a development project

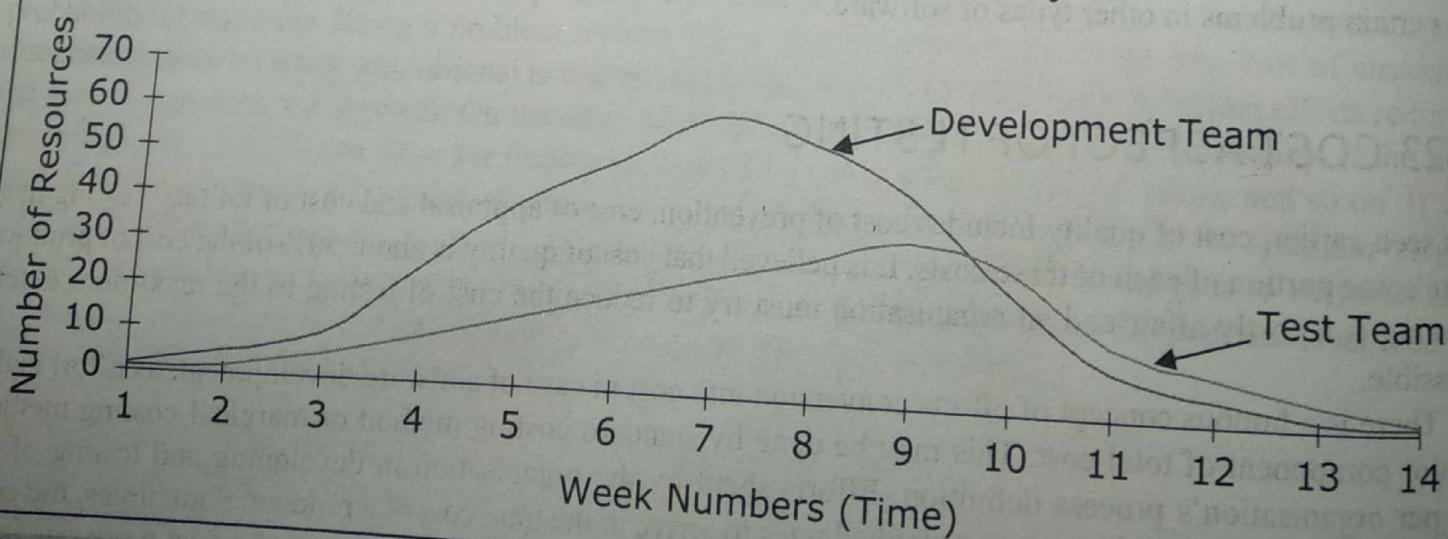


Fig. 3.7

Number of resources for development project

Cost of development/manufacturing includes the cost spent in the following.

- Capturing the requirements, conducting analysis, asking queries, and elicitation of requirements
- Cost spent in designing the application including high-level designing and low-level designing
- Cost spent in writing code, integrating and creating the final product

3.23.1 ASSESSMENT OF COST OF TESTING

Cost of testing may be considered as the cost of the project activities on and above normal phases of development. Various phases of development are associated with phases of verification and validation. Cost of testing is a function of two processes, viz. development process maturity and test process maturity. It has also a very close relationship with the type of application being produced, methodology of development and testing, domain, technical parameters of development, and testing.

Type of Application and Cost of Testing Depth and breadth of testing has direct relationship with the size and importance of an application to a user, and test efforts that the customer is ready to pay for also varies accordingly. As far as size of an application is concerned, testing follows 'Rayleigh Putnam' curve. As the size of application increases, the testing efforts increase exponentially. An organisation must evaluate its own curve on the basis of historical data, or may adapt to the existing baselines in initial phases when historical data is not available.

$$\text{Cost of testing} = k f(x)$$

where 'k' is a constant and $f(x)$ is a function of size of an application while 'x' indicates the size of the application.

An application may be defined as per its importance to the user. If user is completely dependant upon the application, he may want more confidence and thus, invest more in testing. On the other hand, if software does not have business criticality, the customer may not be ready to pay that much cost.

Development and Test Methodology Development and test methodologies also affect testing efforts and costs. It is generally believed that waterfall development methodology can produce robust software and testing efforts required are much lesser. Agile methodologies may need huge testing because there is more stress on regression testing, as each iteration of development is over. Iterative development has more testing costs as there is a change in requirements, design, coding, etc.

Test methodologies have an impact on test efforts and costs. An organisation conducting phase-wise testing has lesser cost than the organisation facing testing at the last phase of development. Life cycle testing is very cheap in comparison to testing at the last phase of development. Usage of regression testing tools may increase the cost of testing in initial phases while in the long term, overall cost of testing is reduced due to automation.

Domain and Technological Aspects Application domain plays an important role, along with criticality of the application, to the users. If the application is going to affect human life, there may be many regulations coming into the picture. The customer will be more cautious about testing and test results. There may be more legal implications if sufficient and adequate testing is not done. Some other types of software affecting huge sum of money are also regulated by the rules, regulations and laws of the place where it is used.

It is believed that technology also plays an important role in deciding the extent of testing. Object-oriented development faces different challenges in testing compared to normal development without any object usage.

There may be more thrust on integration testing rather than unit testing. Old development languages and databases sometimes make testing difficult.

Maturity of Development and Testing Processes Development and testing process maturity is an important issue in deciding the cost of testing. Theoretically, testing is not at all required if development can achieve the defined output. If development process is highly matured and can achieve the expected output, then testing may be considered as wastage. Many engineering organisations have reached the phase of 'zero' defect and 'zero' inspection. But, software application development may not have achieved a level of maturity to produce defect-free products. Some amount of testing is required to find all conceivable faults so that development team can fix them.

Many organisations have a development phase followed by one complete iteration of testing. It must find all defects so that these are eventually fixed in rework phase. Second iteration of testing must achieve the required level of confidence that application will not fail.

Testing involves all three components of cost of quality mentioned below.

3.23.2 COST OF PREVENTION IN TESTING

Cost of prevention is the cost incurred in preventing defects from entering into a system. In various phases of verification and validation, cost of prevention is distributed. Some of these are discussed below.

Cost Spent in Creation of Verification and Validation Artifacts This part of cost is spent when the project team and test team create various artifacts related to verification and validation at different phases of software development. Cost of planning includes creating test plans and quality plans so that quality can be appraised correctly. This phase may also include the time and effort spent in creation of guidelines for testing, reviews, creation of checklists, writing of test cases and test data along with test scenarios. In case of test automation, cost of test-script creation may be a part of prevention cost.

Cost Spent in Training Testers may need training on the domain under testing. They may also need training on test process and various tools used for testing. This may include organisation-level training as well as project-specific training.

Cost of prevention is calculated as follows.

- Cost spent in creating various plans related to software verification and validation. An organisation must have a baseline definition of the effort required to create plans.
- Cost spent in writing test scenarios, test cases, creating guidelines for verification and validation, and creating checklists for doing verification and validation activities. Organisation baselines must define the time and effort required for this activity.
- Training requirements may be separately assessed depending upon the competency of test teams, type of application, and level of tool usage for testing. There may not be baseline data available in this case as it may change from project to project.

3.23.3 COST OF APPRAISAL IN TESTING

Cost of appraisal includes cost spent in actually doing verification and validation activities. Generally, first-time verification/validation is considered under cost of appraisal. Test artifacts also need reviews and testing to confirm that they are correct. Checklists needed for the reviews must also be reviewed before they are used.

Irrespective of whether time and efforts are spent by developers or testers, all efforts spent on conducting reviews, walkthroughs, inspection, unit testing, integration testing, and system testing are considered under cost of appraisal.

Cost of appraisal is calculated as follows.

- Cost required for conducting first-time verification and validation activities can be assessed from the size of an application and organisation baseline data available.
- Time and effort required to conduct the given number of test cases may be derived from the productivity numbers, and number of test cases required can be derived from the size of an application.

3.23.4 COST OF FAILURE IN TESTING

Cost of failure in testing accounts for all retesting, regression testing, and rereviews conducted as the defects are found in earlier iterations. Any cost spent on and above first-time verification and validation makes a cost of failure for testing. The organisation must have a target to reduce the cost of failure continuously, as this directly affects its profitability.

Cost of failure is calculated as follows.

- On the basis of historical data available in baseline studies, one may plan number of iterations required to achieve predetermined exit criteria.
- Number of test cases per iteration and number of iterations required can give the efforts required to perform retesting and regression testing.

3.24 ESTABLISHING TESTING POLICY

Good testing is a deliberate planned effort by the organisation. It does not happen on its own, but detailed planning is required. Testing efforts need to be driven by test policy, test strategy or approach, test planning, etc. Test policy is an intent of test management about how an organisation perceives testing and customer satisfaction. It should define test objectives and test deliverables.

Test strategy or approach must define what steps are required for performing an effective testing. How the test environment will be created, what tools will be used for testing, defect capturing, defect reporting, and number of test cycles required will be a part of test strategy. It must talk about the depth and breadth of testing to ensure adequate confidence levels for users.

Test objectives define what testing will be targeting to achieve. It is better to have test objectives expressed in numbers in place of qualitative definitions. Some of the test objectives may be about code coverage, scenario coverage, and requirement coverage, whereas others may define the targeted number of defects.

Testing must be planned and implemented as per plan. Test plan should contain test objectives and methods applied for defining test scenario, test cases, and test data. It should also explain how the results will be declared and how retesting will be done. It is a general expectation that automation can solve all problems regarding testing process. One thing to be noted is that—automation can increase the speed and repeatability of testing but test planning cannot be done automatically. Rather, it needs involvement of people doing testing.

3.25 METHODS

Generally, methods applied for testing efforts are defined at organisational levels. They are generic in nature and hence, need customisation. They are customised into a test plan, and any tailoring required

to suite a specific project may be done. Management directives establish methods applied for testing. It includes what part will be tested/not tested, and how it will be tested. Which tools will be used for testing, defect-tracking mechanism, communication methods and if there is any decision of automation, how it will be undertaken—all this must be covered by these processes. Management directives are defined in test strategy.

Testing strategy may be discussed with users/customer to get their views/buy-in about testing. It may be accomplished through meetings and memorandums. User/customer must be made aware of cost of finding and fixing defects. All stakeholders for the project must be made aware that 'zero defects' is an impossible condition and acceptance criteria for the project must be defined well in advance (possibly at the time of contract). Methods of using data or inputs provided by a customer must be analysed for sufficiency and correctness.

3.26 STRUCTURED APPROACH TO TESTING

Testing that is concentrated to a single phase at the end of development cycle, just before deployment, is costly. Testing is a life cycle activity and must be a part of entire software development life cycle. If testing is done only in the last phase before delivery to customer, the results obtained may not be accurate and defect fixing may be very costly. Here, it cannot show development process problems and similar defects can be found again and again. Four components of wastes involved in this type of testing are given below.

Waste in Wrong Development Wrong specifications used for development or testing will result into a wrong product and wrong testing. Even if specifications are correct, it may be wrongly interpreted in design, code, documentation, etc. The defects not found during reviews or white box testing will be discovered only at the customer's end. This may lead to high customer dissatisfaction, huge rework, retesting, etc.

Waste in Testing to Detect Defects If testing is intended to find all defects in product, then cost of testing will be very high. Effective reviews can reduce the cost of software testing and development. If entire responsibility for software quality is left to black box testing or final system testing, then the cost of testing and cost of customer dissatisfaction may be very high.

Wastage as Wrong Specifications, Designs, Codes and Documents Must Be Replaced By Correct Specifications, Designs, Codes and Documents The cost of fixing defects may be very high in the last part of testing as there are more number of phases between defect introduction phase and defect detection phase, and defect may percolate through the development phases. Correcting the specifications, designs, codes and documents, and respective retesting/regression testing is a costly process. It can lead to schedule variance, effort variance and customer dissatisfaction. One defect fix can introduce another defect in the system.

Wastage as System Must Be Retested to Ensure That the Corrections Are Correct For every fixing of defect, there is a possibility of some other part of software getting affected in a negative manner. One needs to test software again to ensure that fixing of software has been correct, and it has not affected the other parts in a negative manner. Regression and retesting are essential when defects are found and fixed.

3.27 CATEGORIES OF DEFECT

Software defects may be categorised under different criteria. The categories of defects must be defined in the test plan. The definition may differ from organisation to organisation, project to project and customer to customer.

3.27.1 ON BASIS OF REQUIREMENT/DESIGN SPECIFICATION

- Variance from product specifications as documented in requirement specifications or design specifications represents specification related defects. These defects are responsible for 'Producer's gap'.
- Variance from user/customer expectations as business analyst/system analyst is not able to identify customer needs correctly. These variances may be in the form of implied requirements. These are responsible for 'Users gap'.

3.27.2 TYPES OF DEFECTS

- Wrongly implemented specifications are relate to the specifications as understood by developers differing significantly from what the customer wants. These may be termed 'misinterpretation of specifications'.
- Missing specifications are the specifications that are present in requirement statements but not available in the final product. The requirements are missed, as there is no requirement tracing through product development.
- Features not supported by specifications but present in the product represent something extra. This is something added by developers though these features are not supported by specifications.

3.27.3 ROOT CAUSES OF DEFECTS

- Wrong requirements given by user/customer can be a basic cause of defect. This is due to the inability of the customer to put the requirements in words, or specifying requirements which are not required.
- Business analyst/system analyst interprets customer needs wrongly can be another major cause of defect. This is due to the inability of business analyst/system analyst to elicit requirements.
- System design architect does not understand requirements correctly and hence, the architecture is wrong. This may be due to communication gap or inability of the architect in understanding the requirements.
- Incorrect program specifications, guidelines, and standards are used by respective people. If the organisation processes are not capable, then defects are introduced in the product so produced.
- Errors in coding represent lack of developer's skills in understanding design and implementing it correctly.
- Data entry error caused by the users while using a product. This can be possible when users are not protected adequately. This indicates design problems.
- Errors in testing—false call/failure to detect an existing defect in the product. The first part introduces defects in a correct product while the second part allows defects to go to the customer.
- Mistake in error correction, where defect is introduced while correcting some identified defect.

3.28 DEFECT, ERROR, OR MISTAKE IN SOFTWARE

The problems with software work product may be put under different categories on the basis of who has found it and when it has been found (as shown in Table 3.5).



Table 3.5

Comparison of mistake, error and defect

Mistake	Error	Defect
An issue identified while reviewing own documents, or peer review may be termed 'mistake'. Very low cost of finding mistakes and can be fixed immediately. Most of the time, problems and resolutions are not documented properly.	An issue identified internally or in unit testing may be termed 'error'. Slightly more cost of finding an error and needs some time for fixing. Sometimes, problems and resolutions are documented, but may not be used for process improvements.	An issue identified in black box testing or by customer is termed 'defect'. Most costly and needs longer time for fixing defects. Problems and resolutions are officially documented and used for process improvements.

3.29 DEVELOPING TEST STRATEGY

Test planning includes developing a strategy about how the test team will perform testing. Some key components of testing strategy are as follows.

- Test factors required in particular phase of development
- Test phase corresponding to development phase

Process of developing test strategy goes through the following stages.

Select and Rank Test Factors for the Given Application The test team must identify critical success factors/quality factors/test factors for the software product under testing. A software may have some specific requirements from user's point of view. Test factors must be analysed and prioritised or ranked. Some test factors may be related to each other (either direct or inverse relationship). The trade-off decisions may be taken after consulting with customer, if possible, when the relationship is inverted.

Identify System Development Phases and Related Test Factors The critical success factors may have varying importance as per development life cycle phases. One needs to consider the importance of these factors as per the life cycle phase, that one is going through. The test approach will change accordingly.

Identify Associated Risks with Each Selected Test Factor In Case if it is Not Achieved Trade-offs may lead to few risks of development and testing the software. Customer must be involved in doing trade-offs of test factors and the possible risks of not selecting proper test factor. The risks with probability and impact need to be used to arrive at the decision of trade-off.

Identify Phase in Which Risks of Not Meeting a Test Factor Need to Be Addressed The risks may be tackled in different ways during development life cycle phases. As the phase is over, one needs to assess the actual impact of the risks and effectiveness of devised countermeasures for the same.

3.30 DEVELOPING TESTING METHODOLOGIES (TEST PLAN)

Developing test tactics is the job of project-level test manager/test lead. Different projects may need different tactics as per type of product/customer. Designing and defining of test methodology may take the following route.

3.30.1 ACQUIRE AND STUDY TEST STRATEGY AS DEFINED EARLIER

Test strategy is developed by a test team familiar with business risks associated with software usage. Testing must address the critical success factors for the project and the risk involved in not achieving it.

3.30.2 DETERMINE THE TYPE OF DEVELOPMENT PROJECT BEING EXECUTED

Development projects may be categorised differently by different organisation, as shown below.

- Traditionally developed software by following known methods of development such as waterfall development life cycle. An organisation has a history available, and the lessons learned in previous projects can be used to avoid contingencies in the given project.
- Commercially Off The Shelf (COTS) purchased software which may be integrated in the given software. Requirements and designs of such software may not be available, and integration in terms of parameters passing can be a major constraint.
- Maintenance activities such as bug fixing, enhancement, porting, and reengineering will have their own challenges, such as availability of design documents, compatibilities of various technologies, and code readability.
- Agile methodology of development has small iterations of development and heavy regression testing.
- Iterative method of development has continuously changing requirements and all other artifacts must be updated accordingly.
- Spiral development, where new things are added in system again and again. Generally followed methodology in spiral development is modular design, development and testing.

Another possible way of categorising software is on the basis of its criticality. It can be as follows.

- Life affecting software
- Huge money affecting software
- Software which cannot be tested in real life and requires simulators for testing
- Other software

3.30.3 DETERMINE THE TYPE OF SOFTWARE SYSTEM BEING MADE

Type of software system defines how data processing will be performed by the software. It may involve the following.

- Determine the project scope (whether it is multivendor or multisite development). Distributed development and integration of parts developed by different vendors can be a challenging task.
- New developments including scope of development and scope of testing, when the products are enhanced from existing levels.
- Changes to existing system such as bug fixing, enhancement, reengineering, and porting.

3.30.4 IDENTIFY TACTICAL RISKS RELATED TO DEVELOPMENT

Risks may be introduced in a software due to its nature, type of customer, type of developing organisation, development methodologies used, and skills of teams. Risk may differ from project to project.

- *Structural Risks (Refers to Methods Used to Build a Product)* If the projects are supposed to use existing libraries or designs which may need complex algorithms to be written, the structure of the software

may pose the highest risk. Complex structures with interfaces in relation to many other systems can make the architecture fragile.

- *Technical Risks (Refers to Technology Used to Build and Operate the System)* If the organisation is new to a particular technology, or the technology itself is new to the world, then it can lead to this kind of risk. Unproven technology or inability to work with the latest technology are the problems faced by developers as well as users.
- *Size Risks (Refers to Size of All Aspects of Software)* As the software size increases, it becomes more complex. Maintaining integrity of a very big software itself is a challenge.

3.30.5 DETERMINE WHEN TESTING MUST OCCUR DURING LIFE CYCLE

- Testing phases starting from proposal, contract or requirement testing till acceptance testing, and their integration decide the test strategy for the project.
- Previously collected information, if available, is to be used to decide how much testing is to be done, at what time and in which phases. It defines the cost of testing, cost of customer dissatisfaction and any trade-offs.
- Build tactical test plan which will be used by the test team in execution of testing related activities
- To describe software being tested, test objectives, risks, business functions to be tested, and any specific tests to be performed.

3.30.6 STEPS TO DEVELOP CUSTOMISED TEST STRATEGY

- Select and rank quality factors/test factors as expected by the customer in the final product. Quality factors must be prioritised. Generally, the scale used is 1–99, where '1' indicates higher priority while '99' indicates lower priority. No two quality factors have the same ratings.
- Identify the system development phase where these factors must be controlled. As the defects may originate in different phases, quality factors may change their priority during each phase of development life cycle.
- Identify business risks associated with system under development. If quality factors are not met, these may induce some risk in a product.
- Place risks in a matrix so that one may be able to analyse them. This may be used to devise preventive, corrective and detective measures to control risks.

Table 3.6 shows how test strategy matrix can be developed

Table 3.6

Typical test strategy matrix

Quality factors	Test phases	Requirement	Design	Coding	Testing	Deployment	Maintenance
Factors	Compliance Accuracy Reliability			Risks associated at different phases for different factors.			

3.30.7 TYPE OF DEVELOPMENT METHODOLOGY IMPACT TEST PLAN DECISIONS

Table 3.7 shows, in general, which test tactics can be used depending upon the type of development activity. This is an indicative list and may differ from situation to situation, product to product and customer to customer.

Table 3.7

Development model and testing tactics

Type	Characteristics	Test tactics
Traditional system development such as waterfall model of development	<ul style="list-style-type: none"> Uses a system development methodology as defined User knows requirements completely and these are defined Development determines structure of application 	<ul style="list-style-type: none"> Test at end of each task/step/phase to ensure that exit criteria is achieved Verify that specifications match user needs exactly Test functions and structures as per test plan
Iterative development/prototyping development	<ul style="list-style-type: none"> Complete set of requirements unknown to users and developers Structure predefined by development Refactoring may be done, if required 	<ul style="list-style-type: none"> Verify that case tools are used properly where required by testing Test functionality first Test other areas of product such as integration, system etc
System maintenance methodology	Modify structure if it represents a limiting factor for maintenance activities	<ul style="list-style-type: none"> Test structure consistency as it may affect entire application Works best with release methods of maintenance Requires heavy regression testing as system is updated
Purchase/contracted software COTS	<ul style="list-style-type: none"> System unknown to testers May contain defects in hidden form Functionality defined in user manual Documentation may vary from software to software 	<ul style="list-style-type: none"> Verify that functionality matches needs of business Test functionality as per business need Test fitness for use into production environment

3.31 TESTING PROCESS

Testing is a process made of many milestones. Testers need to achieve them, one by one, to achieve the final goal of testing. Each milestone forms a basis on which the next stage is built. The milestones may vary from organisation to organisation and project to project. Following are few milestones commonly used by many organisations.

Defining Test Policy Test policies are defined by senior management of the organisation or test management, and may or may not form a part of the test plan. Test policy at organisation level defines the intent of test management. Test policy is dependent on the maturity of an organisation in development and test process, customer type, type of software, development methodology, etc. Test policy may be tailored at project level, depending upon the scope and historical information available from similar projects.

Defining Test Strategy Definition of test strategy includes how the test team will be organised, and how it will work to achieve test objectives, decision about coverage, automation, etc. Test strategy provides the actions to the intents defined by test policy. Test strategy helps the test team to understand the approach of testing.

Preparing Test Plan Test planning is done by test managers, test leads or senior testers, as the case may be. Test policy sets a tone, whereas test strategy adds the actions required to complete test policy. Test plan tries to answer six basic questions—What, When, Where, Why, Which and How. Test plans are for individual product/project and customer. They are derived from test strategy and give details of execution of testing activity.

Establishing Testing Objectives to Be Achieved Test objectives measure the effectiveness and efficiency of a testing process. They also define test achievements that they plan for. Test objectives are also defined from the quality objectives for the project or product, and the critical success factors for testing. Testing objectives must be ‘SMART’ (specific, measurable, agreed upon, realistic, and time bound).

Designing Test Scenarios and Test Cases How the test scenarios and test cases will be defined should be explained by the test strategy. A test scenario represents user scenario which acts as a framework for defining test cases. It may have actors and transactions. Similarly, there may be few scenarios arising from system requirements. Theoretically, each transaction in a test scenario eventually becomes a test case.

Writing/Reviewing Test Cases Writing and reviewing test cases along with test scenarios and updating requirement traceability matrix accordingly are the tasks done by senior testers or test leads for the project. The traceability matrix gets completed by adding the test cases and finally, the test results. One more column in the traceability matrix would be the results of execution of test cases, i.e., test results.

Defining Test Data Test data may be defined on the basis of different techniques available for the purpose. It may include boundary value analysis, error guessing, equivalence partitioning, and state transition. Test data must include valid as well as invalid set of data. It must include some special values generated from error guessing. Test data definitions may be important from testing point of view as different iterations must have different test data sets though it may be used by same test cases.

Creation of Test Bed Testing needs creation of environment for testing. It may be a real-life environment or a simulated environment using some simulators. Test bed defines some of the assumptions in a test plan which may induce certain risks of testing. It must reflect real-life situations as closely as possible. Simulators may need definition of few risks, as testing is not done in real environment.

Executing Test Cases Execution of actual test cases with the test data defined for testing the software involves applying test cases as well as test data and trying to get the actual results. It sometimes involves updating test cases or test data, if some mistakes are found in initially defined test cases or test data.

Test Result Logging results of testing in test log is the last part of the testing iteration. There may be several iterations of testing planned and executed. The defect database may be populated, if expected results are not matching with the actual results. One needs to make sure that the expected results are traceable to requirements.

Test Result Analysis Testing is a process of SWOT analysis of software under development and testing. Examining test results and analysis may lead to interpretation of software in terms of capabilities and weakness. At the end of testing, the test team must recommend the next step after testing is completed.

to the project manager—whether the software is ready to go to the next stage or it needs further rework and retesting.

Performing Retesting/Regression Testing When Defects Are Resolved By Development Team When defects are given to a development team, they perform analysis and fix the defects. Retesting is done to find out whether the defects declared as fixed and verified by the development team are really fixed or not. Regression testing is done to confirm that the changed part has not affected (in a negative way) any other parts of software, which were working earlier.

Root Cause Analysis and Corrective/Preventive Actions Root cause analysis is required to initiate corrective actions. Development/test team performs post-mortem reviews to understand the weakness of development as well as test process, and initiates actions to improve them. Process improvement is the last activity after the project is closed and formally accepted by the customer. All defect prevention activities must lead to process improvements.

3.32 ATTITUDE TOWARDS TESTING (COMMON PEOPLE ISSUES)

Attitude of development team and senior management or project management towards a test team is a very important aspect to build morale of the test team. It may be initiated from test policy and may be percolated down to test strategy definition and test planning. Some of the views about test team are as follows.

- New members of development team are not accustomed to view testing as a discovery process where defects are found in the product. The defects found are taken as personal blames rather than system/process lacunae. Sometimes, people try to defend themselves, considering it as an attack on the individual.
- ‘We take pride on what we developed’ or ‘we wish to prove that it is right’ or ‘it is not my fault’ are very common responses. Developers may not accept the defect in the first place. If they accept the presence of a defect, then they try to put blame on somebody else. Root cause analysis is very difficult in such situations as people may attach personal ego to it.
- Conflict between developer and tester can create differences between project teams and test teams. In reality, the sole aim of development and testing must be a customer satisfaction, and defects must be considered as something which prevents achievement of this objective.

3.33 TEST METHODOLOGIES/APPROACHES

The two major disciplines in testing are given below.

Black Box Testing Black box testing is an attesting methodology where product is tested as per software specifications or requirement statement defined by business analysts/system analysts/customer. Black box testing mainly talks about the requirement specification given by customer, or intended requirements as perceived by testers. It deals with testing of an executable and is independent of platform, database, etc. This testing is with the view as if a user is testing the system.

White Box Testing White box testing is a testing methodology where software is tested for the structures. White box testing covers verification of work products as per structure, architecture, coding standards and guidelines of software. It mainly deals with the structure and design of the software product.

White box testing requires that testers must have knowledge about development processes and artifacts including various platforms, databases, etc.

There is one more methodology covering both testing methodologies at the same time.

Gray Box Testing Gray box testing talks about a combination of both approaches, viz. black box testing and white box testing at the same time. There may be various shades of black box testing as well as white box testing in this type of testing, depending upon the requirements of product. Though not a rule, gray box testing mainly concentrates on integration testing part along with unit testing.

Broadly, all other testing techniques may be put in any of these three categories, viz. black box testing, white box testing and gray box testing.

3.33.1 BLACK BOX TESTING (DOMAIN TESTING/SPECIFICATION TESTING)

Black box testing involves testing system/components considering inputs, outputs and general functionalities as defined in requirement specifications. It does not consider any internal processing by the system. Black box testing is independent of platform, database, and system to make sure that the system works as per requirements defined as well as implied ones. Actual system (production environment) is simulated, if it is difficult to create a real-life scenario in test laboratory. It does not make any assumption about technicalities of development process, platform, tools, etc. It represents user scenario and actual user interactions.

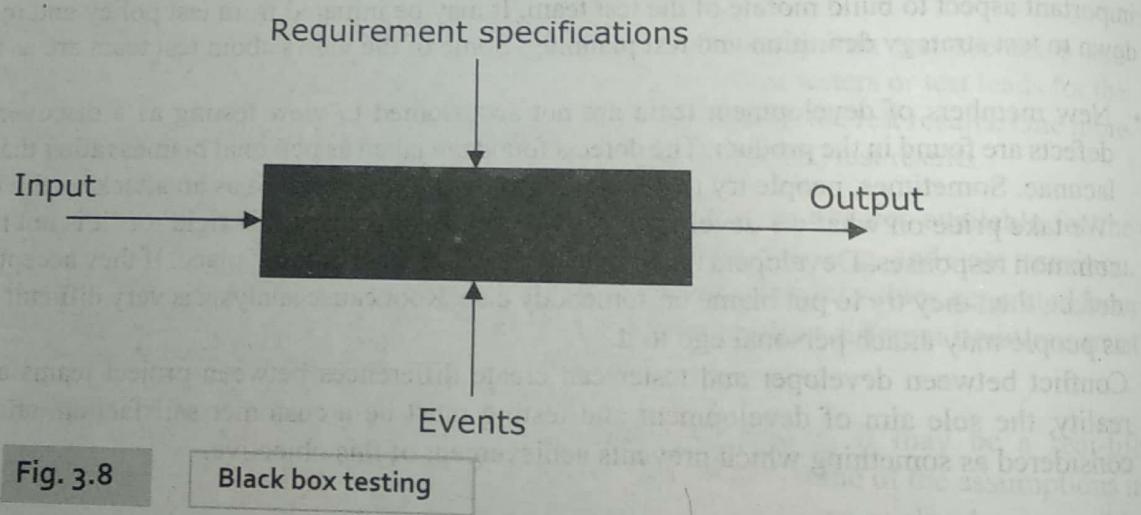


Fig. 3.8

Black box testing

Figure 3.8 shows a block box testing schematically. Black box functional testing is generally conducted for integration testing, system testing, and acceptance testing where the users/customers or the testers as representatives of the customer execute a system as if it is used by users in production environment.

Advantages of 'Black Box Testing' Black box testing is used primarily to test the behavior of an application with respect to expressed or implied requirements of the customer.

- Black box testing is the only method to prove that software does what it is supposed to do and it does not do something which can cause a problem to user/customer.
- It is the only method to show that software is living and it really works.
- Some types of testing can be done only by black box testing methodologies, for example, performance and security.

Disadvantages of 'Black Box Testing' Black box testing has many disadvantages so far as software development methodology is concerned.

- Some logical errors in coding can be missed in black box testing as black box testing efforts are driven by requirements and not by the design. It uses boundary value analysis, equivalence partitioning, and some internal structure problems can be missed.
- Some redundant testing is possible as requirements may execute the same branch of code again and again. If an application calls common functions again and again, then it will be tested so many times that it leads to redundant testing.

Test Case Designing Methodologies Black box testing methodology defines how the user is going to interact with the system without any assumption about how the system is built. As there is no view of how software is built, defining test cases is very difficult. Test cases may be defined using the user scenario called 'test scenario'. Completeness of test scenario is essential for good testing. Theoretically, each sentence in the test scenario may become a test case. Scenario contains activities in terms of transactions and actors.

Test Data Definition Black box testing is mainly driven by the test data used during testing. It may not be feasible to test all possible data which user may be using while working with an application. Some special techniques are applied for defining test data which can give adequate coverage, and also limits the number of test cases and the risk of failure of an application during use. Some of these techniques are mentioned below.

- Equivalence partitioning
- Boundary value analysis
- Cause and effect graph
- State transition testing
- Use case based testing
- Error guessing

3.3.2 WHITE BOX TESTING

White box testing is done on the basis of internal structures of software as defined by requirements, designs, coding standards, and guidelines. It starts with reviews of requirements, designs, and codes. White box testing can ensure that relationship between the requirements, designs, and codes can be interpreted. White box testing is mainly a verification technique where one can ensure that software is built correctly. Figure 3.9 shows a white box testing schematically.

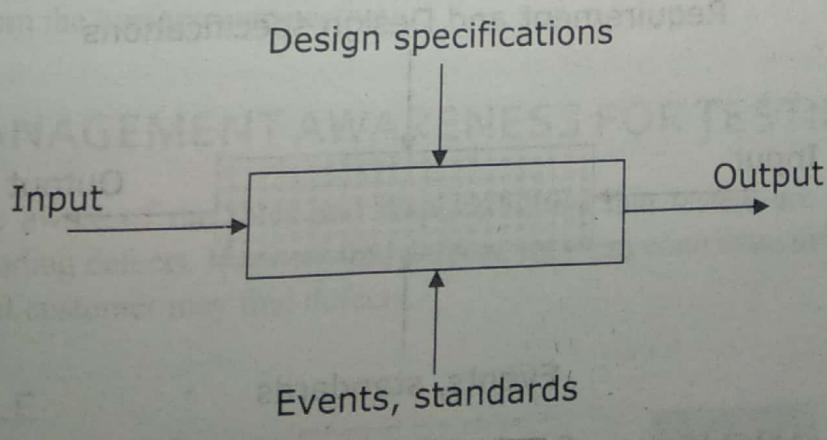


Fig. 3.9

- Advantages of 'White Box Testing'** White box testing is a primary method of verification.
- Only white box testing can ensure that defined processes, procedures, and methods of development have really been followed during software testing. It can check whether the coding standards, commenting and reuse have been followed or not.
 - White box testing or verification can give early warnings, if something is not done properly. It is the most cost effective way of finding defects as it helps in reducing stage contamination.
 - Some characteristics of software work product can be verified only. There is no chance of validating them. For example, code complexity, commenting styles, and reuse.

Disadvantages of 'White Box Testing' White box testing being a verification technique has few shortcomings.

- It does not ensure that user requirements are met correctly. There is no execution of code, and one does not know whether it will really work or not.
- It does not establish whether decisions, conditions, paths, and statements covered during reviews are sufficient or not for the given set of requirements.
- Sometimes, white box testing is dominated by the usage of checklists. Some defects in checklists may reflect directly in the work product. One must do a thorough analysis of all defects.

Test Case Designing Test case designing is based on how test artifacts are created and used during testing. It defines how documents are written and interpreted by each person involved in software development life cycle. Some of the techniques used for white box testing are as follows.

- Statement coverage
- Decision coverage
- Condition coverage
- Path coverage
- Logic coverage

3.33.3 GRAY BOX TESTING

Gray box testing is done on the basis of internal structures of software as defined by requirements, design, coding standards, and guidelines as well as the functional and non-functional requirement specifications. Gray box testing combines verification techniques with validation techniques where one can ensure software is built correctly, and also works. Figure 3.10 shows a gray box testing schematically.

Requirement and Design specifications

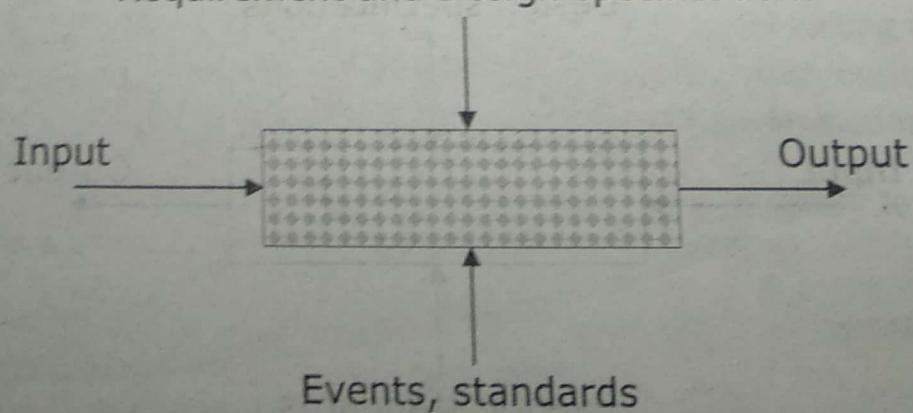


Fig. 3.10

Gray box testing

Advantages of 'Gray Box Testing'

- Gray box testing tries to combine the advantages of white box testing and Black box testing. It checks whether the work product works in a correct manner, both functionally as well as structurally.

Disadvantages of 'Gray Box Testing'

- Generally, gray box testing is conducted with some automation tools. Knowledge of such tools along with their configuration is essential for performing gray box testing.

3.34 PEOPLE CHALLENGES IN SOFTWARE TESTING

Testing is a process and must be improved continuously. People need to analyse and take actions on the shortcomings found in the process, so that they can be improved continuously. Few expectations of software process improvement needs from testers are given below.

- The tester is responsible for improving testing process to ensure better products with less number of defects going to customer, thus enhancing customer satisfaction. All defects must be found and the confidence level must be built in the process that can give customer satisfaction. Proper coverage as required by test plan must be achieved.
- Testing needs trained and skilled people who can deliver products with minimum defects to the stakeholders. Testers have to improve their skills through continuous learning.
- The tester needs a positive team attitude for creative destruction of software. Defect in the software is an opportunity to improve the product and not to blame developers. Testers must be able to pinpoint the lacunae in software development process as the defects are found.
- Testing is creative work and a challenging task. Feasible test scenarios and test cases, as well as effective ways of looking for defects are essential to improve testing effectiveness.
- Programmers and testers work together to improve the quality of software developed and delivered to customer, and the process used for software development and testing. The ultimate aim is customer satisfaction.
- Testers hunt for defects—they pursue defects not people, including developers. Every defect is considered as a process shortcoming. Defect closure needs retesting and regression testing to find whether the defect is really fixed or not, and to ensure that there is no negative impact of a defect on existing functions.
- Testing needs patience, fairness, ambition, creditability, capability, and diligence on part of testers. Every defect must be seen from the business perspective.

3.35 RAISING MANAGEMENT AWARENESS FOR TESTING

The management must be aware of the roles and responsibilities that testers are performing to achieve customer satisfaction by finding defects. If testers find defects, they can contribute in building good software by reducing probability that customer may find defects.

3.35.1 TESTER'S ROLE

While establishing a test function in an organisation, the management has some objectives to be achieved. Test team needs to understand these objectives and fulfill them.

- Calculate testing cost, effectiveness of testing and ensure that management understands the same. By doing good testing, number of customer complaints must reduce and cost of failure must go down.
- Demonstrate cost reduction and increase in effectiveness over a time span (as rework and scrap reduce over a long horizon). This can be shown by reduced customer complaints as well as less rework.
- Highlight needs and benefits of training—in test team as well as development team—on testing activities and skills, so that testers can perform better. Many developers need information about unit testing, integration testing and their role in such testing.
- Collect and distribute information on testing to all team members as well as development team/organisation which can be used for improvement.
- Get involved in test budgeting. Testing needs people, money, time, training and other resources. The organisation may have to develop budget to procure all these aspects.

3.36 SKILLS REQUIRED BY TESTER

Testing needs a disciplined approach. A tester is the person entrusted by an organisation to work as the devil's agent. He/she is a person working for the client, finding the obvious defects in the processes and products. The main purpose of testing is to demonstrate that defects are present, and point towards the weaker areas in the software as well as processes used to build it, so that actions can be initiated in that direction. It must build confidence in management and customer that the application with which they will be working is usable and does not have defects. One must try to build maximum possible skills, and training is one of the effective methods to build a good testing team.

3.36.1 GENERAL SKILLS

Written and Verbal Presentation Skill Presenting test results, or discussing about an application or defects involves communication with many people. Testers are supposed to present test results and tell development team, customer and management about the present status of application and where further improvements can be done. Testers must be good in presentation skills.

Effective Listening Skill Testers need to listen to the customers voice as well as views of developers. Listening to customer as well as developer—to understand the needs and requirements correctly—is required to ensure that the scenarios and test cases can be written in a proper way. Tester's listening skills can convert testing into effective testing. Listening skills can give them complete information about the process, software application and also what the customer and management are looking for.

Facilitation Skill Facilitation of development team as well as customer is done by testers, so that defects are taken in the proper spirit. Testers must be able to tell the exact nature of a defect, how it is happening and how it will affect the users. Testers must contribute to improve development process and take part in building a better product.

Software Development, Operations and Maintenance Good knowledge of software development life cycle and software testing life cycle help testers in designing test scenarios and test cases accordingly. The defect age and cost of testing are important parameters to be controlled by testers.

Continuous Education Testers must undergo continuous education and training to build and enforce quality practices in development processes. They need to undergo training for test planning, test case definition, test data definition, methods and processes applied for testing, and reporting defects.

3.36.2 TESTING SKILLS

Concepts of Testing A tester must have complete knowledge about testing as a discipline. He/she must understand methods, processes, and concepts of testing. He/she must be capable of doing test planning, designing test scenario, writing test cases, and defining test strategy, and defining test data.

Levels of Testing Testing is a multitier activity where the application goes from one level to another after successful completion of the previous level. Testers are involved in each phase of software development right from proposal and contract, followed by requirement till acceptance testing. Testers must ensure that each phase is passed successfully.

Techniques for Validation and Verification Techniques of verification/validation must be understood and facilitated by testers to the development team, customer and management. While writing test cases, he/she needs to define test case pass/fail criteria to validate the test case and product.

Selection and Use of Testing Tools Testing involves use of various tools including automation tools, defect tracking tools, configuration management tools, and simulators. A tester must understand and use the tools effectively.

Knowledge of Testing Standards Testing standards are defined by software quality management. There can be some international standards or organisation/customer defined standards. Testers need to understand these standards, and apply them effectively so that a common understanding can be achieved.

Risk Assessment and Management Testing is risk-driven activity. Test cases and test data must be defined to minimise risks to the final users in production environment. Testing efforts must be managed to improve their effectiveness and efficiency. Managing testing involves planning, organising, directing, coordinating, and controlling testing process.

Developing Test Plan Test plan development is generally done by test managers or test leads while implementation of these plans is done by testers. Individual testers must plan for their part in overall test plan for the project.

Defining Acceptance Criteria Definition of acceptance criteria is an important milestone for testing. Generally, acceptance criteria are defined by customer well before the project starts. Testers need to define acceptance criteria for the phases and iterations of testing. There are various forms of acceptance criteria which will be discussed later. The phase-end acceptance criteria may be defined by the testers in test plan.

Checking of Testing Processes Testers follow the processes as defined in the test plan. They need to audit the testing processes to check the compliance and effectiveness, and also initiate actions if deviations are observed. Testers must contribute in testing process improvements.

Execution of Test Plan Testers are given the responsibility of executing a test plan. It includes defining test scenario, test cases, and test data, and their execution. They put test results in test log and defects in defect logging tool. Resolved defects are taken for retesting. They must perform regression testing when

planned. Testers must do analysis of test results to define weaker and stronger areas of software development and testing process. They must be able to define test coverage such as code coverage, statement coverage, branch coverage, requirement coverage, and function coverage.

Continuous Improvement of Testing Process Testers must plan for continuous improvement of testing process. Testing process must be subjected to improvements followed by phase of consolidations. Actions must be planned for improving process, and the results must be compared with expectations. If some deviations are observed, new actions can be initiated.



Testing tips

Customer pays for a product on the basis of the value he finds in acquiring such product. Cost of development and cost of testing define the profit available to an organisation by selling such product/project. Market forces define the sales price of the product/project. There is always a pressure on development and testing to reduce the cost to improve profitability. The following list may be considered as a generic guideline for reducing cost of testing or improving the value of a product.

Reduce Software Development Risk Development activities may introduce several risks starting from requirement capturing, through design and development, coding and so on. Software testing must be effective to locate the defects as early as possible so that stage contamination can be reduced.

Perform Testing Effectively Testing must be able to capture defects as effectively and efficiently as possible. It must give adequate confidence to users that application will not meet any accidental failures. It must be able to find as many defects as possible, so that they will be fixed eventually and probability of failure at customer place is minimised.

Uncover Maximum Number of Defects Each defect uncovered must reduce a chance of customer complaint. If testing can find 100% defects present in the given software, it will not be possible for customer to see any failure during use. Though it is very difficult, one must try to find all conceivable defects. Successful tester is one who finds maximum number of defects.

Use Business Logic Testers must have a good knowledge about the domain under testing. This is true for system testers where it is expected that they would be working as normal users. They must use business logic to improve efficiency and effectiveness of testing, and also testing must give enough confidence to users that there will not be any accidental failures.

Testing Must Occur Throughout SDLC Defect found as early as possible can reduce cost of failure by preventing stage contamination. Testing concentrating more on system testing may not be very effective as software developed may be very fragile.

Testing Must Cover Functional/Structural Parts Often, people consider functional testing as a complete testing. One must keep in mind that there are five types of requirements mentioned by 'TELOS' (Technical Economic Legal Operational System). Only operational requirements may cover functional as well as non-functional requirements. It must also cover the way software is built to give better screen designs, optimum performance, and security.

Summary

This chapter establishes the basics of software testing. It starts with a historical perspective of testing and then explains how testing evolved from mere debugging to defect prevention technique. It then discusses the benefits of independent testing. 'TQM' concept of testing, 'Big Bang' approach of testing, and benefits of 'TQM' testing are elucidated in detail.

The chapter also presents the definitions of a successful tester and the basic principles of software testing. It offers a detailed exposition of the process of creating test policy, test strategy, and test plan. It also gives an indepth understanding of 'Black Box Testing', 'White Box Testing' and 'Gray Box Testing'. The chapter concludes with skills required by a good tester and challenges faced by a tester.

- 1) Explain the evolution of software testing from debugging to prevention based testing.
- 2) Explain why independent testing is required.
- 3) Explain big bang approach of software testing.
- 4) Explain total quality management approach of software testing.
- 5) Explain concept of TQM cost perspective.
- 6) Explain testing as a process of software certification.
- 7) Explain the basic principles on which testing is based.
- 8) Explain the concept of test team's defect finding efficiency.
- 9) Explain test case's defect finding efficiency.
- 10) What are the challenges faced by testers?
- 11) Explain the process of developing test strategy.
- 12) Explain the process of developing test methodology.
- 13) Which skills are expected in a good tester?



CHAPTER 2

Testing throughout the software life cycle

Testing is not a stand-alone activity. It has its place within a software development life cycle model and therefore the life cycle applied will largely determine how testing is organized. There are many different forms of testing. Because several disciplines, often with different interests, are involved in the development life cycle, it is important to clearly understand and define the various test levels and types. This chapter discusses the most commonly applied software development models, test levels and test types. Maintenance can be seen as a specific instance of a development process. The way maintenance influences the test process, levels and types and how testing can be organized is described in the last section of this chapter.

2.1 SOFTWARE DEVELOPMENT MODELS

- 1 Understand the relationship between development, test activities and work products in the development life cycle and give examples based on project and product characteristics and context. (K2)**
- 2 Recognize the fact that software development models must be adapted to the context of project and product characteristics. (K1)**
- 3 Recall reasons for different levels of testing and characteristics of good testing in any life cycle model. (K1)**

The development process adopted for a project will depend on the project aims and goals. There are numerous development life cycles that have been developed in order to achieve different required objectives. These life cycles range from lightweight and fast methodologies, where time to market is of the essence, through to fully controlled and documented methodologies where quality and reliability are key drivers. Each of these methodologies has its place in modern software development and the most appropriate development process should be applied to each project. The models specify the various stages of the process and the order in which they are carried out.

The life cycle model that is adopted for a project will have a big impact on the testing that is carried out. Testing does not exist in isolation; test activities

are highly related to software development activities. It will define the what, where, and when of our planned testing, influence regression testing, and largely determine which test techniques to use. The way testing is organized must fit the development life cycle or it will fail to deliver its benefit. If time to market is the key driver, then the testing must be fast and efficient. If a fully documented software development life cycle, with an audit trail of evidence, is required, the testing must be fully documented.

In every development life cycle, a part of testing is focused on **verification** testing and a part is focused on **validation** testing. Verification is concerned with evaluating a work product, component or system to determine whether it meets the requirements set. In fact, verification focuses on the question 'Is the deliverable built according to the specification?'. Validation is concerned with evaluating a work product, component or system to determine whether it meets the user needs and requirements. Validation focuses on the question 'Is the deliverable fit for purpose, e.g. does it provide a solution to the problem?'.

2.1.1 V-model

Before discussing the **V-model**, we will look at the model which came before it. The waterfall model was one of the earliest models to be designed. It has a natural timeline where tasks are executed in a sequential fashion. We start at the top of the waterfall with a feasibility study and flow down through the various project tasks finishing with implementation into the live environment. Design flows through into development, which in turn flows into build, and finally on into test. Testing tends to happen towards the end of the project life cycle so defects are detected close to the live implementation date. With this model it has been difficult to get feedback passed backwards up the waterfall and there are difficulties if we need to carry out numerous iterations for a particular phase.

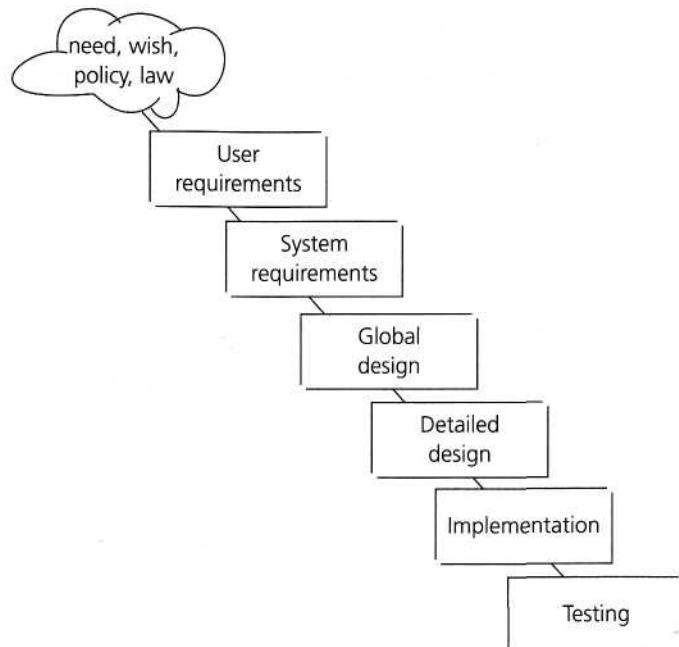


FIGURE 2.1 Waterfall model

The V-model was developed to address some of the problems experienced using the traditional waterfall approach. Defects were being found too late in the life cycle, as testing was not involved until the end of the project. Testing also added lead time due to its late involvement. The V-model provides guidance that testing needs to begin as early as possible in the life cycle, which, as we've seen in Chapter 1, is one of the fundamental principles of structured testing. It also shows that testing is not only an execution-based activity. There are a variety of activities that need to be performed before the end of the coding phase. These activities should be carried out in *parallel* with development activities, and testers need to work with developers and business analysts so they can perform these activities and tasks and produce a set of test deliverables. The work products produced by the developers and business analysts during development are the basis of testing in one or more levels. By starting test design early, defects are often found in the test basis documents. A good practice is to have testers involved even earlier, during the review of the (draft) test basis documents. The V-model is a model that illustrates how testing activities (verification and validation) can be integrated into each phase of the life cycle. Within the V-model, validation testing takes place especially during the early stages, e.g. reviewing the user requirements, and late in the life cycle, e.g. during user acceptance testing.

Although variants of the V-model exist, a common type of V-model uses four **test levels**. The four test levels used, each with their own objectives, are:

- component testing: searches for defects in and verifies the functioning of software components (e.g. modules, programs, objects, classes etc.) that are separately testable;
- integration testing: tests interfaces between components, interactions to different parts of a system such as an operating system, file system and hardware or interfaces between systems;
- system testing: concerned with the behavior of the whole system/product as defined by the scope of a development project or product. The main focus of system testing is verification against specified requirements;
- acceptance testing: validation testing with respect to user needs, requirements, and business processes conducted to determine whether or not to accept the system.

The various test levels are explained and discussed in detail in Section 2.2. In practice, a V-model may have more, fewer or different levels of development and testing, depending on the project and the software product. For example, there may be component integration testing after component testing and system integration testing after system testing. Test levels can be combined or reorganized depending on the nature of the project or the system architecture. For the integration of a **commercial off-the-shelf (COTS) software** product into a system, a purchaser may perform only integration testing at the system level (e.g. integration to the infrastructure and other systems) and at a later stage acceptance testing.

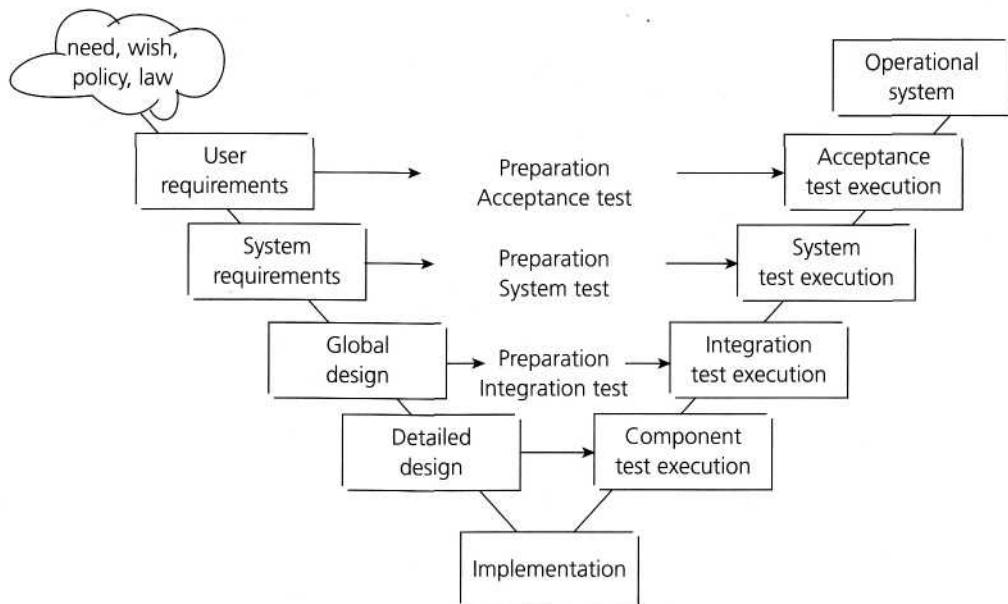


FIGURE 2.2 V-model

Note that the types of work products mentioned in Figure 2.2 on the left side of the V-model are just an illustration. In practice they come under many different names. References for generic work products include the Capability Maturity Model Integration (CMMI) or the 'Software life cycle processes' from ISO/IEC 12207. The CMMI is a framework for process improvement for both system engineering and software engineering. It provides guidance on where to focus and how, in order to increase the level of process maturity [Chrissis *et al.*, 2004]. ISO/IEC 12207 is an integrated software life cycle process standard that is rapidly becoming more popular.

2.1.2 Iterative life cycles

Not all life cycles are sequential. There are also iterative or incremental life cycles where, instead of one large development time line from beginning to end, we cycle through a number of smaller self-contained life cycle phases for the same project. As with the V-model, there are many variants of iterative life cycles.

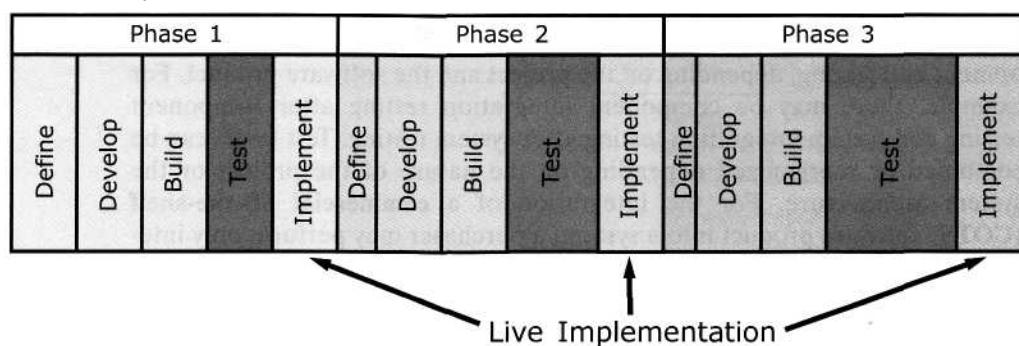


FIGURE 2.3 Iterative development model

A common feature of iterative approaches is that the delivery is divided into increments or builds with each increment adding new functionality. The initial increment will contain the infrastructure required to support the initial build functionality. The increment produced by an iteration may be tested at several levels as part of its development. Subsequent increments will need testing for the new functionality, regression testing of the existing functionality, and integration testing of both new and existing parts. Regression testing is increasingly important on all iterations after the first one. This means that more testing will be required at each subsequent delivery phase which must be allowed for in the project plans. This life cycle can give early market presence with critical functionality, can be simpler to manage because the workload is divided into smaller pieces, and can reduce initial investment although it may cost more in the long run. Also early market presence will mean validation testing is carried out at each increment, thereby giving early feedback on the business value and fitness-for-use of the product.

Examples of iterative or **incremental development models** are prototyping, Rapid Application Development (RAD), Rational Unified Process (RUP) and agile development. For the purpose of better understanding iterative development models and the changing role of testing a short explanation of both RAD and agile development is provided.

Rapid Application Development

Rapid Application Development (RAD) is formally a parallel development of functions and subsequent integration.

Components/functions are developed in parallel as if they were mini projects, the developments are time-boxed, delivered, and then assembled into a working prototype. This can very quickly give the customer something to see and use and to provide feedback regarding the delivery and their requirements. Rapid change and development of the product is possible using this methodology. However the product specification will need to be developed for the product at some point, and the project will need to be placed under more formal controls prior to going into production. This methodology allows early

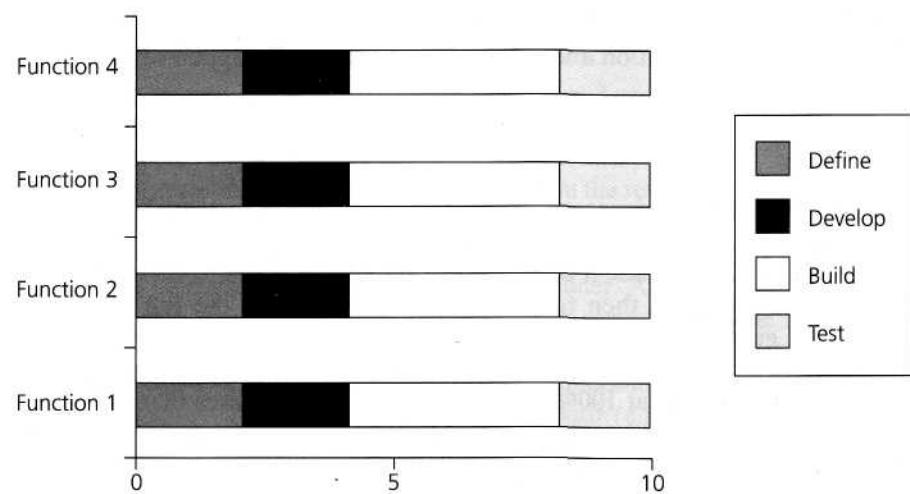


FIGURE 2.4 RAD model

validation of technology risks and a rapid response to changing customer requirements.

Dynamic System Development Methodology [DSDM] is a refined RAD process that allows controls to be put in place in order to stop the process from getting out of control. Remember we still need to have the essentials of good development practice in place in order for these methodologies to work. We need to maintain strict configuration management of the rapid changes that we are making in a number of parallel development cycles. From the testing perspective we need to plan this very carefully and update our plans regularly as things will be changing very rapidly (see Chapter 5 for more on test plans).

The RAD development process encourages active customer feedback. The customer gets early visibility of the product, can provide feedback on the design and can decide, based on the existing functionality, whether to proceed with the development, what functionality to include in the next delivery cycle or even to halt the project if it is not delivering the expected value. An early business-focused solution in the market place gives an early return on investment (ROI) and can provide valuable marketing information for the business. Validation with the RAD development process is thus an early and major activity.

Agile development

Extreme Programming (XP) is currently one of the most well-known agile development life cycle models. (See [Agile] for ideas behind this approach.) The methodology claims to be more human friendly than traditional development methods. Some characteristics of XP are:

- It promotes the generation of business stories to define the functionality.
- It demands an on-site customer for continual feedback and to define and carry out functional acceptance testing.
- It promotes pair programming and shared code ownership amongst the developers.
- It states that component test scripts shall be written before the code is written and that those tests should be automated.
- It states that integration and testing of the code shall happen several times a day.
- It states that we always implement the simplest solution to meet today's problems.

With XP there are numerous iterations each requiring testing. XP developers write every test case they can think of and automate them. Every time a change is made in the code it is component tested and then integrated with the existing code, which is then fully integration-tested using the full set of test cases. This gives continuous integration, by which we mean that changes are incorporated continuously into the software build. At the same time, all test cases must be running at 100% meaning that all the test cases that have been identified and automated are executed and pass. XP is not about doing extreme activities during the development process, it is about doing known value-adding activities in an extreme manner.

2.1.3 Testing within a life cycle model

In summary, whichever life cycle model is being used, there are several characteristics of good testing:

- for every development activity there is a corresponding testing activity;
- each test level has test objectives specific to that level;
- the analysis and design of tests for a given test level should begin during the corresponding development activity;
- testers should be involved in reviewing documents as soon as drafts are available in the development cycle.

2.2 TEST LEVELS

1 Compare the different levels of testing: major objectives, typical objects of testing, typical targets of testing (e.g. functional or structural) and related work products, people who test, types of defects and failures to be identified. (K2)

The V-model for testing was introduced in Section 2.1. This section looks in more detail at the various test levels. The key characteristics for each test level are discussed and defined to be able to more clearly separate the various test levels. A thorough understanding and definition of the various test levels will identify missing areas and prevent overlap and repetition. Sometimes we may wish to introduce deliberate overlap to address specific risks. Understanding whether we want overlaps and removing the gaps will make the test levels more complementary thus leading to more effective and efficient testing.

2.2.1 Component testing

Component testing, also known as unit, module and program testing, searches for defects in, and verifies the functioning of software (e.g. modules, programs, objects, classes, etc.) that are separately testable.

Component testing may be done in isolation from the rest of the system depending on the context of the development life cycle and the system. Most often **stubs** and **drivers** are used to replace the missing software and simulate the interface between the software components in a simple manner. A stub is called from the software component to be tested; a driver calls a component to be tested (see Figure 2.5).

Component testing may include testing of functionality and specific non-functional characteristics such as resource-behavior (e.g. memory leaks), performance or **robustness testing**, as well as structural testing (e.g. decision coverage). Test cases are derived from work products such as the software design or the data model.

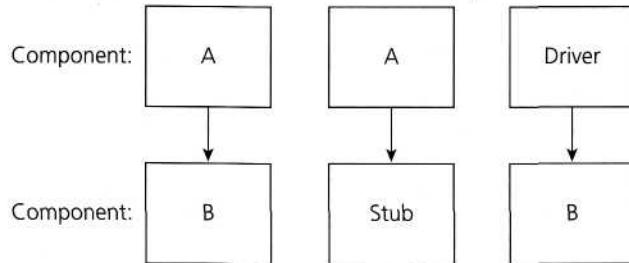


FIGURE 2.5 Stubs and drivers

Typically, component testing occurs with access to the code being tested and with the support of the development environment, such as a unit test framework or debugging tool, and in practice usually involves the programmer who wrote the code. Sometimes, depending on the applicable level of risk, component testing is carried out by a different programmer thereby introducing independence. Defects are typically fixed as soon as they are found, without formally recording the incidents found.

One approach in component testing, used in Extreme Programming (XP), is to prepare and automate test cases before coding. This is called a test-first approach or **test-driven development**. This approach is highly iterative and is based on cycles of developing test cases, then building and integrating small pieces of code, and executing the component tests until they pass.

2.2.2 Integration testing

Integration testing tests interfaces between components, interactions to different parts of a system such as an operating system, file system and hardware or interfaces between systems. Note that integration testing should be differentiated from other integration activities. Integration testing is often carried out by the integrator, but preferably by a specific integration tester or test team.

There may be more than one level of integration testing and it may be carried out on test objects of varying size. For example:

- component integration testing tests the interactions between software components and is done after component testing;
- system integration testing tests the interactions between different systems and may be done after system testing. In this case, the developing organization may control only one side of the interface, so changes may be destabilizing. Business processes implemented as workflows may involve a series of systems that can even run on different platforms.

The greater the scope of integration, the more difficult it becomes to isolate failures to a specific interface, which may lead to an increased risk. This leads to varying approaches to integration testing. One extreme is that all components or systems are integrated simultaneously, after which everything is tested as a whole. This is called 'big-bang' integration testing. Big-bang testing has the advantage that everything is finished before integration testing starts. There is no need to simulate (as yet unfinished) parts. The major disadvantage is that in

general it is time-consuming and difficult to trace the cause of failures with this late integration. So big-bang integration may seem like a good idea when planning the project, being optimistic and expecting to find no problems. If one thinks integration testing will find defects, it is a good practice to consider whether time might be saved by breaking down the integration test process. Another extreme is that all programs are integrated one by one, and a test is carried out after each step (incremental testing). Between these two extremes, there is a range of variants. The incremental approach has the advantage that the defects are found early in a smaller assembly when it is relatively easy to detect the cause. A disadvantage is that it can be time-consuming since stubs and drivers have to be developed and used in the test. Within incremental integration testing a range of possibilities exist, partly depending on the system architecture:

- Top-down: testing takes place from top to bottom, following the control flow or architectural structure (e.g. starting from the GUI or main menu). Components or systems are substituted by stubs.
- Bottom-up: testing takes place from the bottom of the control flow upwards. Components or systems are substituted by drivers.
- Functional incremental: integration and testing takes place on the basis of the functions or functionality, as documented in the functional specification.

The preferred integration sequence and the number of integration steps required depend on the location in the architecture of the high-risk interfaces. The best choice is to start integration with those interfaces that are expected to cause most problems. Doing so prevents major defects at the end of the integration test stage. In order to reduce the risk of late defect discovery, integration should normally be incremental rather than 'big-bang'. Ideally testers should understand the architecture and influence integration planning. If integration tests are planned before components or systems are built, they can be developed in the order required for most efficient testing.

At each stage of integration, testers concentrate solely on the integration itself. For example, if they are integrating component A with component B they are interested in testing the communication between the components, not the functionality of either one. Both functional and structural approaches may be used. Testing of specific non-functional characteristics (e.g. performance) may also be included in integration testing. Integration testing may be carried out by the developers, but can be done by a separate team of specialist integration testers, or by a specialist group of developers/integrators including non-functional specialists.

2.2.3 System testing

System testing is concerned with the behavior of the whole system/product as defined by the scope of a development project or product. It may include tests based on risks and/or requirements specification, business processes, use cases, or other high level descriptions of system behavior, interactions with the operating system, and system resources. System testing is most often the final test on behalf of development to verify that the system to be delivered meets the specification and its purpose may be to find as many defects as possible. Most often

it is carried out by specialist testers that form a dedicated, and sometimes independent, test team within development, reporting to the development manager or project manager. In some organizations system testing is carried out by a third party team or by business analysts. Again the required level of independence is based on the applicable risk level and this will have a high influence on the way system testing is organized.

System testing should investigate both **functional** and **non-functional requirements** of the system. Typical non-functional tests include performance and reliability. Testers may also need to deal with incomplete or undocumented requirements. System testing of functional requirements starts by using the most appropriate specification-based (black-box) techniques for the aspect of the system to be tested. For example, a decision table may be created for combinations of effects described in business rules. Structure-based (white-box) techniques may also be used to assess the thoroughness of testing elements such as menu dialog structure or web page navigation (see Chapter 4 for more on the various types of technique).

System testing requires a controlled **test environment** with regard to, amongst other things, control of the software versions, testware and the test data (see Chapter 5 for more on configuration management). A system test is executed by the development organization in a (properly controlled) environment. The test environment should correspond to the final target or production environment as much as possible in order to minimize the risk of environment-specific failures not being found by testing.

2.2.4 Acceptance testing

When the development organization has performed its system test and has corrected all or most defects, the system will be delivered to the user or customer for **acceptance testing**. The acceptance test should answer questions such as: 'Can the system be released?', 'What, if any, are the outstanding (business) risks?' and 'Has development met their obligations?'. Acceptance testing is most often the responsibility of the user or customer, although other stakeholders may be involved as well. The execution of the acceptance test requires a test environment that is for most aspects, representative of the production environment ('as-if production').

The goal of acceptance testing is to establish confidence in the system, part of the system or specific non-functional characteristics, e.g. usability, of the system. Acceptance testing is most often focused on a validation type of testing, whereby we are trying to determine whether the system is fit for purpose. Finding defects should not be the main focus in acceptance testing. Although it assesses the system's readiness for deployment and use, it is not necessarily the final level of testing. For example, a large-scale system integration test may come after the acceptance of a system.

Acceptance testing may occur at more than just a single level, for example:

- A Commercial Off The Shelf (COTS) software product may be acceptance tested when it is installed or integrated.
- Acceptance testing of the usability of a component may be done during component testing.

- Acceptance testing of a new functional enhancement may come before system testing.

Within the acceptance test for a business-supporting system, two main test types can be distinguished; as a result of their special character, they are usually prepared and executed separately. The user acceptance test focuses mainly on the functionality thereby validating the fitness-for-use of the system by the business user, while the **operational acceptance test** (also called production acceptance test) validates whether the system meets the requirements for operation. The user acceptance test is performed by the users and application managers. In terms of planning, the user acceptance test usually links tightly to the system test and will, in many cases, be organized partly overlapping in time. If the system to be tested consists of a number of more or less independent subsystems, the acceptance test for a subsystem that complies to the exit criteria of the system test can start while another subsystem may still be in the system test phase. In most organizations, system administration will perform the operational acceptance test shortly before the system is released. The operational acceptance test may include testing of backup/restore, disaster recovery, maintenance tasks and periodic check of security vulnerabilities.

Other types of acceptance testing that exist are contract acceptance testing and **compliance acceptance testing**. Contract acceptance testing is performed against a contract's acceptance criteria for producing custom-developed software. Acceptance should be formally defined when the contract is agreed. Compliance acceptance testing or regulation acceptance testing is performed against the regulations which must be adhered to, such as governmental, legal or safety regulations.

If the system has been developed for the mass market, e.g. commercial off-the-shelf software (COTS), then testing it for individual users or customers is not practical or even possible in some cases. Feedback is needed from potential or existing users in their market before the software product is put out for sale commercially. Very often this type of system undergoes two stages of acceptance test. The first is called **alpha testing**. This test takes place at the developer's site. A cross-section of potential users and members of the developer's organization are invited to use the system. Developers observe the users and note problems. Alpha testing may also be carried out by an independent test team. **Beta testing**, or field testing, sends the system to a cross-section of users who install it and use it under real-world working conditions. The users send records of incidents with the system to the development organization where the defects are repaired.

Note that organizations may use other terms, such as factory acceptance testing and site acceptance testing for systems that are tested before and after being moved to a customer's site.

2.3 TEST TYPES: THE TARGETS OF TESTING

- 1 Compare four software test types (functional, non-functional, structural and change-related) by example. (K2)
- 2 Recognize that functional and structural tests occur at any test level. (K1)
- 3 Identify and describe non-functional test types based on non-functional requirements. (K2)
- 4 Identify and describe test types based on the analysis of a software system's structure or architecture. (K2)
- 5 Describe the purpose of confirmation testing and regression testing. (K2)

Test types are introduced as a means of clearly defining the objective of a certain test level for a programme or project. We need to think about different types of testing because testing the functionality of the component or system may not be sufficient at each level to meet the overall test objectives. Focusing the testing on a specific test objective and, therefore, selecting the appropriate type of test helps making and communicating decisions against test objectives easier.

A **test type** is focused on a particular test objective, which could be the testing of a function to be performed by the component or system; a non-functional quality characteristic, such as reliability or usability; the structure or architecture of the component or system; or related to changes, i.e. confirming that defects have been fixed (confirmation testing, or re-testing) and looking for unintended changes (regression testing). Depending on its objectives, testing will be organized differently. For example, component testing aimed at performance would be quite different to component testing aimed at achieving decision coverage.

2.3.1 Testing of function (functional testing)

The function of a system (or component) is 'what it does'. This is typically described in a requirements specification, a functional specification, or in use cases. There may be some functions that are 'assumed' to be provided that are not documented that are also part of the requirement for a system, though it is difficult to test against undocumented and implicit requirements. Functional tests are based on these functions, described in documents or understood by the testers and may be performed at all test levels (e.g. test for components may be based on a component specification).

Functional testing considers the specified behavior and is often also referred to as **black-box testing**. This is not entirely true, since black-box testing also includes non-functional testing (see Section 2.3.2).

Function (or functionality) **testing** can, based upon ISO 9126, be done focusing on suitability, **interoperability, security**, accuracy and compliance. Security testing, for example, investigates the functions (e.g. a firewall) relating to detection of threats, such as viruses, from malicious outsiders.

Testing functionality can be done from two perspectives: requirements-based or business-process-based.

Requirements-based testing uses a specification of the functional requirements for the system as the basis for designing tests. A good way to start is to use the table of contents of the requirements specification as an initial test inventory or list of items to test (or not to test). We should also prioritize the requirements based on risk criteria (if this is not already done in the specification) and use this to prioritize the tests. This will ensure that the most important and most critical tests are included in the testing effort.

Business-process-based testing uses knowledge of the business processes. Business processes describe the scenarios involved in the day-to-day business use of the system. For example, a personnel and payroll system may have a business process along the lines of: someone joins the company, he or she is paid on a regular basis, and he or she finally leaves the company. Use cases originate from object-oriented development, but are nowadays popular in many development life cycles. They also take the business processes as a starting point, although they start from tasks to be performed by users. Use cases are a very useful basis for test cases from a business perspective.

The techniques used for functional testing are often **specification-based**, but experienced-based techniques can also be used (see Chapter 4 for more on test techniques). Test conditions and test cases are derived from the functionality of the component or system. As part of test designing, a model may be developed, such as a process model, state transition model or a plain-language specification.

2.3.2 Testing of software product characteristics (non-functional testing)

A second target for testing is the testing of the quality characteristics, or non-functional attributes of the system (or component or integration group). Here we are interested in how well or how fast something is done. We are testing something that we need to measure on a scale of measurement, for example time to respond.

Non-functional testing, as functional testing, is performed at all test levels. Non-functional testing includes, but is not limited to, **performance testing, load testing, stress testing**, usability testing, maintainability testing, reliability testing and portability testing. It is the testing of 'how well' the system works.

Many have tried to capture software quality in a collection of characteristics and related sub-characteristics. In these models some elementary characteristics keep on reappearing, although their place in the hierarchy can differ. The International Organization for Standardization (ISO) has defined a set of quality characteristics [ISO/IEC 9126, 2001]. This set reflects a major step towards consensus in the IT industry and thereby addresses the general notion of software quality. The ISO 9126 standard defines six quality characteristics and the subdivision of each quality characteristic into a number of

sub-characteristics. This standard is getting more and more recognition in the industry, enabling development, testing and their stakeholders to use a common terminology for quality characteristics and thereby for non-functional testing.

The characteristics and their sub-characteristics are, respectively:

- functionality, which consists of five sub-characteristics: suitability, accuracy, security, interoperability and compliance; this characteristic deals with functional testing as described in Section 2.3.1;
- **reliability**, which is defined further into the sub-characteristics maturity (robustness), fault-tolerance, recoverability and compliance;
- **usability**, which is divided into the sub-characteristics understandability, learnability, operability, attractiveness and compliance;
- **efficiency**, which is divided into time behavior (performance), resource utilization and compliance;
- **maintainability**, which consists of five sub-characteristics: analyzability, changeability, stability, testability and compliance;
- **portability**, which also consists of five sub-characteristics: adaptability, installability, co-existence, replaceability and compliance.

2.3.3 Testing of software structure/architecture (structural testing)

The third target of testing is the structure of the system or component. If we are talking about the structure of a system, we may call it the system architecture. Structural testing is often referred to as '**white-box**' or 'glass-box' because we are interested in what is happening 'inside the box'.

Structural testing is most often used as a way of measuring the thoroughness of testing through the coverage of a set of structural elements or coverage items. It can occur at any test level, although it is true to say that it tends to be mostly applied at component and integration and generally is less likely at higher test levels, except for business-process testing. At component integration level it may be based on the architecture of the system, such as a calling hierarchy. A system, system integration or acceptance testing test basis could be a business model or menu structure.

At component level, and to a lesser extent at component integration testing, there is good tool support to measure **code coverage**. Coverage measurement tools assess the percentage of executable elements (e.g. statements or decision outcomes) that have been exercised (i.e. covered) by a **test suite**. If coverage is not 100%, then additional tests may need to be written and run to cover those parts that have not yet been exercised. This of course depends on the exit criteria. (Coverage techniques are covered in Chapter 4.)

The techniques used for structural testing are structure-based techniques, also referred to as **white-box techniques**. Control flow models are often used to support structural testing.

2.3.4 Testing related to changes (confirmation and regression testing)

The final target of testing is the testing of changes. This category is slightly different to the others because if you have made a change to the software, you will have changed the way it functions, the way it performs (or both) and its structure. However we are looking here at the specific types of tests relating to changes, even though they may include all of the other test types.

Confirmation testing (re-testing)

When a test fails and we determine that the cause of the failure is a software defect, the defect is reported, and we can expect a new version of the software that has had the defect fixed. In this case we will need to execute the test again to confirm that the defect has indeed been fixed. This is known as **confirmation testing** (also known as re-testing).

When doing confirmation testing, it is important to ensure that the test is executed in exactly the same way as it was the first time, using the same inputs, data and environment. If the test now passes does this mean that the software is now correct? Well, we now know that at least one part of the software is correct - where the defect was. But this is not enough. The fix may have introduced or uncovered a different defect elsewhere in the software. The way to detect these 'unexpected side-effects' of fixes is to do regression testing.

Regression testing

Like confirmation testing, regression testing involves executing test cases that have been executed before. The difference is that, for regression testing, the test cases probably passed the last time they were executed (compare this with the test cases executed in confirmation testing - they failed the last time).

The term '**regression testing**' is something of a misnomer. It would be better if it were called 'anti-regression' testing because we are executing tests with the intent of checking that the system has not regressed (that is, it does not now have more defects in it as a result of some change). More specifically, the purpose of regression testing is to verify that modifications in the software or the environment have not caused unintended adverse side effects and that the system still meets its requirements.

It is common for organizations to have what is usually called a regression test suite or regression test pack. This is a set of test cases that is specifically used for regression testing. They are designed to collectively exercise most functions (certainly the most important ones) in a system but not test any one in detail. It is appropriate to have a regression test suite at every level of testing (component testing, integration testing, system testing, etc.). All of the test cases in a regression test suite would be executed every time a new version of software is produced and this makes them ideal candidates for **automation**. If the regression test suite is very large it may be more appropriate to select a subset for execution.

Regression tests are executed whenever the software changes, either as a result of fixes or new or changed functionality. It is also a good idea to execute them when some aspect of the environment changes, for example when a new version of a database management system is introduced or a new version of a source code compiler is used.

Maintenance of a regression test suite should be carried out so it evolves over time in line with the software. As new functionality is added to a system new regression tests should be added and as old functionality is changed or removed so too should regression tests be changed or removed. As new tests are added a regression test suite may become very large. If all the tests have to be executed manually it may not be possible to execute them all every time the regression suite is used. In this case a subset of the test cases has to be chosen. This selection should be made in light of the latest changes that have been made to the software. Sometimes a regression test suite of automated tests can become so large that it is not always possible to execute them all. It may be possible and desirable to eliminate some test cases from a large regression test suite for example if they are repetitive (tests which exercise the same conditions) or can be combined (if they are always run together). Another approach is to eliminate test cases that have not found a defect for a long time (though this approach should be used with some care!).

2.4 MAINTENANCE TESTING

- 1 Compare maintenance testing (testing an operational system) to testing a new application with respect to test types, triggers for testing and amount of testing. (K2)**
- 2 Identify reasons for maintenance testing (modifications, migration and retirement). (K2)**
- 3 Describe the role of regression testing and impact analysis in maintenance. (K2)**

Once deployed, a system is often in service for years or even decades. During this time the system and its operational environment is often corrected, changed or extended. Testing that is executed during this life cycle phase is called '**maintenance testing**'.

Note that maintenance testing is different from **maintainability testing**, which defines how easy it is to maintain the system.

The development and test process applicable to new developments does not change fundamentally for maintenance purposes. The same test process steps will apply and, depending on the size and risk of the changes made, several levels of testing are carried out: a component test, an integration test, a system test and an acceptance test. A maintenance test process usually begins with the receipt of an application for a change or a release plan. The test manager will use this as a basis for producing a test plan. On receipt of the new or changed specifications, corresponding test cases are specified or adapted. On receipt of the test object, the new and modified tests and the regression tests are executed. On completion of the testing, the testware is once again preserved.

Comparing maintenance testing to testing a new application is merely a matter of an approach from a different angle, which gives rise to a number of

changes in emphasis. There are several areas where most differences occur, for example regarding the test basis. A 'catching-up' operation is frequently required when systems are maintained. Specifications are often 'missing', and a set of testware relating to the specifications simply does not exist. It may well be possible to carry out this catching-up operation along with testing a new maintenance release, which may reduce the cost. If it is impossible to compile any specifications from which test cases can be written, including expected results, an alternative test basis, e.g. a **test oracle**, should be sought by way of compromise. A search should be made for documentation which is closest to the specifications and which can be managed by developers as well as testers. In such cases it is advisable to draw the customer's attention to the lower test quality which may be achieved. Be aware of possible problems of 'daily production'. In the worst case nobody knows what is being tested, many test cases are executing the same scenario and if an incident is found it is often hard to trace it back to the actual defect since no traceability to test designs and/or requirements exists. Note that reproducibility of tests is also important for maintenance testing.

One aspect which, in many cases, differs somewhat from the development situation is the test organization. New development and their appropriate test activities are usually carried out as parts of a project, whereas maintenance tests are normally executed as an activity in the regular organization. As a result, there is often some lack of resources and flexibility, and the test process may experience more competition from other activities.

2.4.1 Impact analysis and regression testing

Usually maintenance testing will consist of two parts:

- testing the changes
- regression tests to show that the rest of the system has not been affected by the maintenance work.

In addition to testing what has been changed, maintenance testing includes extensive regression testing to parts of the system that have not been changed. A major and important activity within maintenance testing is impact analysis. During **impact analysis**, together with stakeholders, a decision is made on what parts of the system may be unintentionally affected and therefore need careful regression testing. Risk analysis will help to decide where to focus regression testing - it is unlikely that the team will have time to repeat all the existing tests.

If the test specifications from the original development of the system are kept, one may be able to reuse them for regression testing and to adapt them for changes to the system. This may be as simple as changing the expected results for your existing tests. Sometimes additional tests may need to be built. Extension or enhancement to the system may mean new areas have been specified and tests would be drawn up just as for the development. It is also possible that updates are needed to an automated test set, which is often used to support regression testing.

2.4.2 Triggers for maintenance testing

As stated maintenance testing is done on an existing operational system. It is triggered by modifications, migration, or retirement of the system. Modifications include planned enhancement changes (e.g. release-based), corrective and emergency changes, and changes of **environment**, such as planned operating system or database upgrades, or patches to newly exposed or discovered vulnerabilities of the operating system. Maintenance testing for migration (e.g. from one platform to another) should include **operational testing** of the new environment, as well as the changed software. Maintenance testing for the retirement of a system may include the testing of data migration or archiving, if long data-retention periods are required.

Since modifications are most often the main part of maintenance testing for most organizations, this will be discussed in more detail. From the point of view of testing, there are two types of modifications. There are modifications in which testing may be planned, and there are ad-hoc corrective modifications, which cannot be planned at all. Ad-hoc corrective maintenance takes place when the search for solutions to defects cannot be delayed. Special test procedures are required at that time.

Planned modifications

The following types of planned modification may be identified:

- perfective modifications (adapting software to the user's wishes, for instance by supplying new functions or enhancing performance);
- adaptive modifications (adapting software to environmental changes such as new hardware, new systems software or new legislation);
- corrective planned modifications (deferrable correction of defects).

The standard structured test approach is almost fully applicable to planned modifications. On average, planned modification represents over 90% of all maintenance work on systems. [Pol and van Veenendaal]

Ad-hoc corrective modifications

Ad-hoc corrective modifications are concerned with defects requiring an immediate solution, e.g. a production run which dumps late at night, a network that goes down with a few hundred users on line, a mailing with incorrect addresses. There are different rules and different procedures for solving problems of this kind. It will be impossible to take the steps required for a structured approach to testing. If, however, a number of activities are carried out prior to a possible malfunction, it may be possible to achieve a situation in which reliable tests can be executed in spite of 'panic stations' all round. To some extent this type of maintenance testing is often like first aid - patching up - and at a later stage the standard test process is then followed to establish a robust fix, test it and establish the appropriate level of documentation.

A risk analysis of the operational systems should be performed in order to establish which functions or programs constitute the greatest risk to the operational services in the event of disaster. It is then established - in respect of the functions at risk - which (test) actions should be performed if a particular malfunction occurs. Several types of malfunction may be identified and there are

various ways of responding to them for each function at risk. A possible reaction might be that a relevant function at risk should always be tested, or that, under certain circumstances, testing might be carried out in retrospect (the next day, for instance). If it is decided that a particular function at risk should always be tested whenever relevant, a number of standard tests, which could be executed almost immediately, should be prepared for this purpose. The standard tests would obviously be prepared and maintained in accordance with the structured test approach.

Even in the event of ad-hoc modifications, it is therefore possible to bring about an improvement in quality by adopting a specific test approach. It is important to make a thorough risk analysis of the system and to specify a set of standard tests accordingly.

CHAPTER REVIEW

Let's review what you have learned in this chapter.

From Section 2.1, you should now understand the relationship between development and testing within a development life cycle, including the test activities and test (work) products. You should know that the development model to use should fit, or must be adapted to fit, the project and product characteristics. You should be able to recall the reasons for different levels of testing and characteristics of good testing in any life cycle model. You should know the glossary terms **(commercial) off-the-shelf software (COTS)**, **incremental development model**, **test level**, **validation**, **verification** and **V-model**.

From Section 2.2, you should know the typical levels of testing. You should be able to compare the different levels of testing with respect to their major objectives, typical objects of testing, typical targets of testing (e.g. functional or structural) and related work products. You should also know which persons perform the testing activities at the various test levels, the types of defects found and failures to be identified. You should know the glossary terms **alpha testing**, **beta testing**, **component testing**, **driver**, **functional requirements**, **integration**, **integration testing**, **non-functional testing**, **operational testing**, **regulation acceptance testing (compliance testing)**, **robustness testing**, **stub**, **system testing**, **test-driven development**, **test environment** and **user acceptance testing**.

From Section 2.3, you should know the four major types of test (functional, non-functional, structural and change-related) and should be able to provide some concrete examples for each of these. You should understand that functional and structural tests occur at any test level and be able to explain how they are applied in the various test levels. You should be able to identify and describe non-functional test types based on non-functional requirements and product quality characteristics. Finally you should be able to explain the purpose of confirmation testing (re-testing) and regression testing in the context of change-related testing. You should know the glossary terms **black-box testing**, **code coverage**, **confirmation testing (re-testing)**, **functional testing**, **interoperability testing**, **load testing**, **Maintainability testing**, **performance testing**, **portability testing**, **regression testing**, **reliability testing**, **security testing**, **specification-based testing**, **stress testing**, **structural testing**, **test suite**, **usability testing** and **white-box testing**.

From Section 2.4, you should be able to compare maintenance testing to testing of new applications. You should be able to identify triggers and reasons for maintenance testing, such as modifications, migration and retirement. Finally you should be able to describe the role of regression testing and impact analysis within maintenance testing. You should know the glossary terms **impact analysis** and **maintenance testing**.

SAMPLE EXAM QUESTIONS

Question 1 What are good practices for testing within the development life cycle?

- a. Early test analysis and design.
- b. Different test levels are defined with specific objectives.
- c. Testers will start to get involved as soon as coding is done.
- d. A and B above.

Question 2 Which option best describes objectives for test levels with a life cycle model?

- a. Objectives should be generic for any test level.
- b. Objectives are the same for each test level.
- c. The objectives of a test level don't need to be defined in advance.
- d. Each level has objectives specific to that level.

Question 3 Which of the following is a test type?

- a. Component testing
- b. Functional testing
- c. System testing
- d. Acceptance testing

Question 4 Which of the following is a non-functional quality characteristic?

- a. Feasibility
- b. Usability
- c. Maintenance
- d. Regression

Question 5 Which of these is a functional test?

- a. Measuring response time on an on-line booking system.
- b. Checking the effect of high volumes of traffic in a call-center system.
- c. Checking the on-line bookings screen information and the database contents against the information on the letter to the customers.
- d. Checking how easy the system is to use.

Question 6 Which of the following is a true statement regarding the process of fixing emergency changes?

- a. There is no time to test the change before it goes live, so only the best developers should do this work and should not involve testers as they slow down the process.
- b. Just run the retest of the defect actually fixed.
- c. Always run a full regression test of the whole system in case other parts of the system have been adversely affected.
- d. Retest the changed area and then use risk assessment to decide on a reasonable subset of the whole regression test to run in case other parts of the system have been adversely affected.

Question 7 A regression test:

- a. Is only run once.
- b. Will always be automated.
- c. Will check unchanged areas of the software to see if they have been affected.
- d. Will check changed areas of the software to see if they have been affected.

Question 8 Non-functional testing includes:

- a. Testing to see where the system does not function correctly.
- b. Testing the quality attributes of the system including reliability and usability.
- c. Gaining user approval for the system.
- d. Testing a system feature using only the software required for that function.

Question 9 Beta testing is:

- a. Performed by customers at their own site.
- b. Performed by customers at the software developer's site.
- c. Performed by an independent test team.
- d. Useful to test software developed for a specific customer or user.

Chapter 5

Boundary Value Testing

In Chapter 3, we saw that a function maps values from one set (its domain) to values in another set (its range) and that the domain and range can be cross products of other sets. Any program can be considered to be a function in the sense that program inputs form its domain and program outputs form its range. In this and the next two chapters, we examine how to use knowledge of the functional nature of a program to identify test cases for the program. Input domain testing (also called “boundary value testing”) is the best-known specification-based testing technique. Historically, this form of testing has focused on the input domain; however, it is often a good supplement to apply many of these techniques to develop range-based test cases.

There are two independent considerations that apply to input domain testing. The first asks whether or not we are concerned with invalid values of variables. Normal boundary value testing is concerned only with valid values of the input variables. Robust boundary value testing considers invalid and valid variable values. The second consideration is whether we make the “single fault” assumption common to reliability theory. This assumes that faults are due to incorrect values of a single variable. If this is not warranted, meaning that we are concerned with interaction among two or more variables, we need to take the cross product of the individual variables. Taken together, the two considerations yield four variations of boundary value testing:

- Normal boundary value testing
- Robust boundary value testing
- Worst-case boundary value testing
- Robust worst-case boundary value testing

For the sake of comprehensible drawings, the discussion in this chapter refers to a function, F , of two variables x_1 and x_2 . When the function F is implemented as a program, the input variables x_1 and x_2 will have some (possibly unstated) boundaries:

$$a \leq x_1 \leq b$$

$$c \leq x_2 \leq d$$

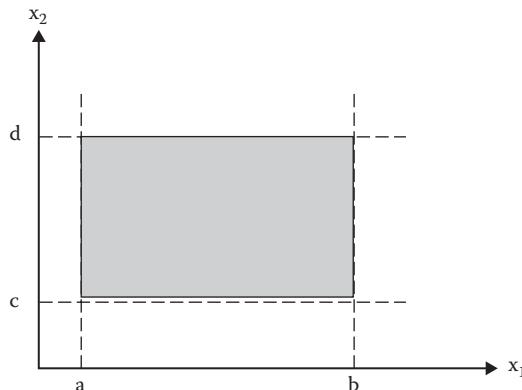


Figure 5.1 Input domain of a function of two variables.

Unfortunately, the intervals $[a, b]$ and $[c, d]$ are referred to as the ranges of x_1 and x_2 , so right away we have an overloaded term. The intended meaning will always be clear from its context. Strongly typed languages (such as Ada® and Pascal) permit explicit definition of such variable ranges. In fact, part of the historical reason for strong typing was to prevent programmers from making the kinds of errors that result in faults that are easily revealed by boundary value testing. Other languages (such as COBOL, FORTRAN, and C) are not strongly typed, so boundary value testing is more appropriate for programs coded in these languages. The input space (domain) of our function F is shown in Figure 5.1. Any point within the shaded rectangle and including the boundaries is a legitimate input to the function F .

5.1 Normal Boundary Value Testing

All four forms of boundary value testing focus on the boundary of the input space to identify test cases. The rationale behind boundary value testing is that errors tend to occur near the extreme values of an input variable. Loop conditions, for example, may test for $<$ when they should test for \leq , and counters often are “off by one.” (Does counting begin at zero or at one?) The basic idea of boundary value analysis is to use input variable values at their minimum, just above the minimum, a nominal value, just below their maximum, and at their maximum. A commercially available testing tool (originally named T) generates such test cases for a properly specified program. This tool has been successfully integrated with two popular front-end CASE tools (Teamwork from Cadre Systems, and Software through Pictures from Aonix [part of Atego]; for more information, see <http://www.aonix.com/pdf/2140-AON.pdf>). The T tool refers to these values as min, min+, nom, max-, and max. The robust forms add two values, min- and max+.

The next part of boundary value analysis is based on a critical assumption; it is known as the “single fault” assumption in reliability theory. This says that failures are only rarely the result of the simultaneous occurrence of two (or more) faults. The All Pairs testing approach (described in Chapter 20) contradicts this, with the observation that, in software-controlled medical systems, almost all faults are the result of interaction between a pair of variables. Thus, the normal and robust variations cases are obtained by holding the values of all but one variable at their nominal

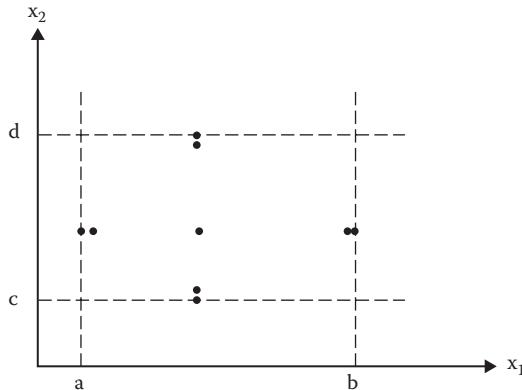


Figure 5.2 Boundary value analysis test cases for a function of two variables.

values, and letting that variable assume its full set of test values. The normal boundary value analysis test cases for our function F of two variables (illustrated in Figure 5.2) are

$$\{<x_{1\text{nom}}, x_{2\text{min}}>, <x_{1\text{nom}}, x_{2\text{min+}}>, <x_{1\text{nom}}, x_{2\text{nom}}>, <x_{1\text{nom}}, x_{2\text{max-}}>, <x_{1\text{nom}}, x_{2\text{max}}>, <x_{1\text{min}}, x_{2\text{nom}}>, <x_{1\text{min}}, x_{2\text{nom+}}>, <x_{1\text{max-}}, x_{2\text{nom}}>, <x_{1\text{max}}, x_{2\text{nom}}>\}$$

5.1.1 Generalizing Boundary Value Analysis

The basic boundary value analysis technique can be generalized in two ways: by the number of variables and by the kinds of ranges. Generalizing the number of variables is easy: if we have a function of n variables, we hold all but one at the nominal values and let the remaining variable assume the min, min+, nom, max-, and max values, repeating this for each variable. Thus, for a function of n variables, boundary value analysis yields $4n + 1$ unique test cases.

Generalizing ranges depends on the nature (or more precisely, the type) of the variables themselves. In the NextDate function, for example, we have variables for the month, the day, and the year. In a FORTRAN-like language, we would most likely encode these, so that January would correspond to 1, February to 2, and so on. In a language that supports user-defined types (like Pascal or Ada), we could define the variable month as an enumerated type {Jan., Feb., ..., Dec.}. Either way, the values for min, min+, nom, max-, and max are clear from the context. When a variable has discrete, bounded values, as the variables in the commission problem have, the min, min+, nom, max-, and max are also easily determined. When no explicit bounds are present, as in the triangle problem, we usually have to create “artificial” bounds. The lower bound of side lengths is clearly 1 (a negative side length is silly); but what might we do for an upper bound? By default, the largest representable integer (called MAXINT in some languages) is one possibility; or we might impose an arbitrary upper limit such as 200 or 2000. For other data types, as long as a variable supports an ordering relation (see Chapter 3 for a definition), we can usually infer the min, min+, nominal, max-, and max values. Test values for alphabet characters, for example, would be {a, b, m, y, and z}.

Boundary value analysis does not make much sense for Boolean variables; the extreme values are TRUE and FALSE, but no clear choice is available for the remaining three. We will see in

Chapter 7 that Boolean variables lend themselves to decision table-based testing. Logical variables also present a problem for boundary value analysis. In the ATM example, a customer's PIN is a logical variable, as is the transaction type (deposit, withdrawal, or inquiry). We could go through the motions of boundary value analysis testing for such variables, but the exercise is not very satisfying to the tester's intuition.

5.1.2 Limitations of Boundary Value Analysis

Boundary value analysis works well when the program to be tested is a function of several independent variables that represent bounded physical quantities. Mathematically, the variables need to be described by a true ordering relation, in which, for every pair $\langle a, b \rangle$ of values of a variable, it is possible to say that $a \leq b$ or $b \leq a$. (See Chapter 3 for a detailed definition of ordering relations.) Sets of car colors, for example, or football teams, do not support an ordering relation; thus, no form of boundary value testing is appropriate for such variables. The key words here are independent and physical quantities. A quick look at the boundary value analysis test cases for NextDate (in Section 5.5) shows them to be inadequate. Very little stress occurs on February and on leap years. The real problem here is that interesting dependencies exist among the month, day, and year variables. Boundary value analysis presumes the variables to be truly independent. Even so, boundary value analysis happens to catch end-of-month and end-of-year faults. Boundary value analysis test cases are derived from the extrema of bounded, independent variables that refer to physical quantities, with no consideration of the nature of the function, nor of the semantic meaning of the variables. We see boundary value analysis test cases to be rudimentary because they are obtained with very little insight and imagination. As with so many things, you get what you pay for.

The physical quantity criterion is equally important. When a variable refers to a physical quantity, such as temperature, pressure, air speed, angle of attack, load, and so forth, physical boundaries can be extremely important. (In an interesting example of this, Sky Harbor International Airport in Phoenix had to close on June 26, 1992, because the air temperature was 122°F. Aircraft pilots were unable to make certain instrument settings before takeoff: the instruments could only accept a maximum air temperature of 120°F.) In another case, a medical analysis system uses stepper motors to position a carousel of samples to be analyzed. It turns out that the mechanics of moving the carousel back to the starting cell often causes the robot arm to miss the first cell.

As an example of logical (vs. physical) variables, we might look at PINs or telephone numbers. It is hard to imagine what faults might be revealed by testing PIN values of 0000, 0001, 5000, 9998, and 9999.

5.2 Robust Boundary Value Testing

Robust boundary value testing is a simple extension of normal boundary value testing: in addition to the five boundary value analysis values of a variable, we see what happens when the extrema are exceeded with a value slightly greater than the maximum (max+) and a value slightly less than the minimum (min-). Robust boundary value test cases for our continuing example are shown in Figure 5.3.

Most of the discussion of boundary value analysis applies directly to robustness testing, especially the generalizations and limitations. The most interesting part of robustness testing is not with the inputs but with the expected outputs. What happens when a physical quantity exceeds its

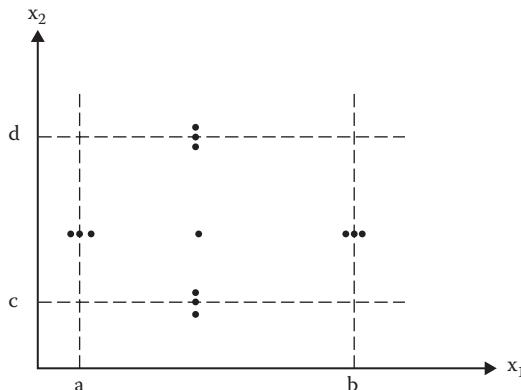


Figure 5.3 Robustness test cases for a function of two variables.

maximum? If it is the angle of attack of an airplane wing, the aircraft might stall. If it is the load capacity of a public elevator, we hope nothing special would happen. If it is a date, like May 32, we would expect an error message. The main value of robustness testing is that it forces attention on exception handling. With strongly typed languages, robustness testing may be very awkward. In Pascal, for example, if a variable is defined to be within a certain range, values outside that range result in run-time errors that abort normal execution. This raises an interesting question of implementation philosophy: is it better to perform explicit range checking and use exception handling to deal with “robust values,” or is it better to stay with strong typing? The exception handling choice mandates robustness testing.

5.3 Worst-Case Boundary Value Testing

Both forms of boundary value testing, as we said earlier, make the single fault assumption of reliability theory. Owing to their similarity, we treat both normal worst-case boundary testing and robust worst-case boundary testing in this subsection. Rejecting single-fault assumption means that we are interested in what happens when more than one variable has an extreme value. In electronic circuit analysis, this is called “worst-case analysis”; we use that idea here to generate worst-case test cases. For each variable, we start with the five-element set that contains the min, min+, nom, max-, and max values. We then take the Cartesian product (see Chapter 3) of these sets to generate test cases. The result of the two-variable version of this is shown in Figure 5.4.

Worst-case boundary value testing is clearly more thorough in the sense that boundary value analysis test cases are a proper subset of worst-case test cases. It also represents much more effort: worst-case testing for a function of n variables generates 5^n test cases, as opposed to $4n + 1$ test cases for boundary value analysis.

Worst-case testing follows the generalization pattern we saw for boundary value analysis. It also has the same limitations, particularly those related to independence. Probably the best application for worst-case testing is where physical variables have numerous interactions, and where failure of the function is extremely costly. For really paranoid testing, we could go to robust worst-case testing. This involves the Cartesian product of the seven-element sets we used in robustness testing resulting in 7^n test cases. Figure 5.5 shows the robust worst-case test cases for our two-variable function.

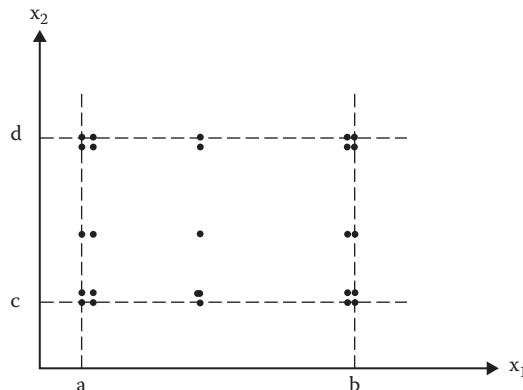


Figure 5.4 Worst-case test cases for a function of two variables.

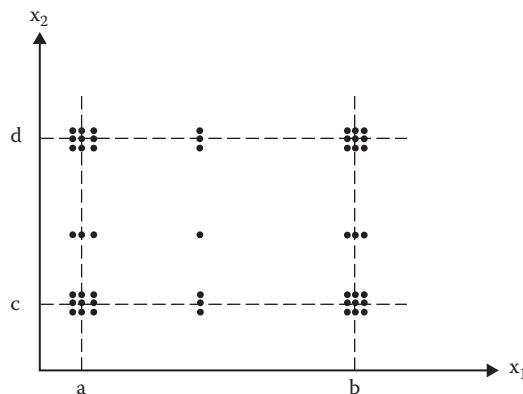


Figure 5.5 Robust worst-case test cases for a function of two variables.

5.4 Special Value Testing

Special value testing is probably the most widely practiced form of functional testing. It also is the most intuitive and the least uniform. Special value testing occurs when a tester uses domain knowledge, experience with similar programs, and information about “soft spots” to devise test cases. We might also call this *ad hoc* testing. No guidelines are used other than “best engineering judgment.” As a result, special value testing is very dependent on the abilities of the tester.

Despite all the apparent negatives, special value testing can be very useful. In the next section, you will find test cases generated by the methods we just discussed for three of our examples. If you look carefully at these, especially for the NextDate function, you find that none is very satisfactory. Special value test cases for NextDate will include several test cases involving February 28, February 29, and leap years. Even though special value testing is highly subjective, it often results in a set of test cases that is more effective in revealing faults than the test sets generated by boundary value methods—testimony to the craft of software testing.

5.5 Examples

Each of the three continuing examples is a function of three variables. Printing all the test cases from all the methods for each problem is very space consuming, so we just have selected examples for worst-case boundary value and robust worst-case boundary value testing.

5.5.1 Test Cases for the Triangle Problem

In the problem statement, no conditions are specified on the triangle sides, other than being integers. Obviously, the lower bounds of the ranges are all 1. We arbitrarily take 200 as an upper bound. For each side, the test values are {1, 2, 100, 199, 200}. Robust boundary value test cases will add {0, 201}. Table 5.1 contains boundary value test cases using these ranges. Notice that test cases 3, 8, and 13 are identical; two should be deleted. Further, there is no test case for scalene triangles.

The cross-product of test values will have 125 test cases (some of which will be repeated)—too many to list here. The full set is available as a spreadsheet in the set of student exercises. Table 5.2 only lists the first 25 worst-case boundary value test cases for the triangle problem. You can picture them as a plane slice through the cube (actually it is a rectangular parallelepiped) in which $a = 1$ and the other two variables take on their full set of cross-product values.

Table 5.1 Normal Boundary Value Test Cases

Case	a	b	c	Expected Output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a triangle
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a triangle

Table 5.2 (Selected) Worst-Case Boundary Value Test Cases

Case	a	b	c	Expected Output
1	1	1	1	Equilateral
2	1	1	2	Not a triangle
3	1	1	100	Not a triangle
4	1	1	199	Not a triangle
5	1	1	200	Not a triangle
6	1	2	1	Not a triangle
7	1	2	2	Isosceles
8	1	2	100	Not a triangle
9	1	2	199	Not a triangle
10	1	2	200	Not a triangle
11	1	100	1	Not a triangle
12	1	100	2	Not a triangle
13	1	100	100	Isosceles
14	1	100	199	Not a triangle
15	1	100	200	Not a triangle
16	1	199	1	Not a triangle
17	1	199	2	Not a triangle
18	1	199	100	Not a triangle
19	1	199	199	Isosceles
20	1	199	200	Not a triangle
21	1	200	1	Not a triangle
22	1	200	2	Not a triangle
23	1	200	100	Not a triangle
24	1	200	199	Not a triangle
25	1	200	200	Isosceles

5.5.2 Test Cases for the NextDate Function

All 125 worst-case test cases for NextDate are listed in Table 5.3. Take some time to examine it for gaps of untested functionality and for redundant testing. For example, would anyone actually want to test January 1 in five different years? Is the end of February tested sufficiently?

Table 5.3 Worst-Case Test Cases

<i>Case</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected Output</i>
1	1	1	1812	1, 2, 1812
2	1	1	1813	1, 2, 1813
3	1	1	1912	1, 2, 1912
4	1	1	2011	1, 2, 2011
5	1	1	2012	1, 2, 2012
6	1	2	1812	1, 3, 1812
7	1	2	1813	1, 3, 1813
8	1	2	1912	1, 3, 1912
9	1	2	2011	1, 3, 2011
10	1	2	2012	1, 3, 2012
11	1	15	1812	1, 16, 1812
12	1	15	1813	1, 16, 1813
13	1	15	1912	1, 16, 1912
14	1	15	2011	1, 16, 2011
15	1	15	2012	1, 16, 2012
16	1	30	1812	1, 31, 1812
17	1	30	1813	1, 31, 1813
18	1	30	1912	1, 31, 1912
19	1	30	2011	1, 31, 2011
20	1	30	2012	1, 31, 2012
21	1	31	1812	2, 1, 1812
22	1	31	1813	2, 1, 1813
23	1	31	1912	2, 1, 1912
24	1	31	2011	2, 1, 2011
25	1	31	2012	2, 1, 2012
26	2	1	1812	2, 2, 1812
27	2	1	1813	2, 2, 1813
28	2	1	1912	2, 2, 1912

(continued)

Table 5.3 Worst-Case Test Cases (Continued)

<i>Case</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected Output</i>
29	2	1	2011	2, 2, 2011
30	2	1	2012	2, 2, 2012
31	2	2	1812	2, 3, 1812
32	2	2	1813	2, 3, 1813
33	2	2	1912	2, 3, 1912
34	2	2	2011	2, 3, 2011
35	2	2	2012	2, 3, 2012
36	2	15	1812	2, 16, 1812
37	2	15	1813	2, 16, 1813
38	2	15	1912	2, 16, 1912
39	2	15	2011	2, 16, 2011
40	2	15	2012	2, 16, 2012
41	2	30	1812	Invalid date
42	2	30	1813	Invalid date
43	2	30	1912	Invalid date
44	2	30	2011	Invalid date
45	2	30	2012	Invalid date
46	2	31	1812	Invalid date
47	2	31	1813	Invalid date
48	2	31	1912	Invalid date
49	2	31	2011	Invalid date
50	2	31	2012	Invalid date
51	6	1	1812	6, 2, 1812
52	6	1	1813	6, 2, 1813
53	6	1	1912	6, 2, 1912
54	6	1	2011	6, 2, 2011
55	6	1	2012	6, 2, 2012
56	6	2	1812	6, 3, 1812
57	6	2	1813	6, 3, 1813

(continued)

Table 5.3 Worst-Case Test Cases (Continued)

<i>Case</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected Output</i>
58	6	2	1912	6, 3, 1912
59	6	2	2011	6, 3, 2011
60	6	2	2012	6, 3, 2012
61	6	15	1812	6, 16, 1812
62	6	15	1813	6, 16, 1813
63	6	15	1912	6, 16, 1912
64	6	15	2011	6, 16, 2011
65	6	15	2012	6, 16, 2012
66	6	30	1812	7, 1, 1812
67	6	30	1813	7, 1, 1813
68	6	30	1912	7, 1, 1912
69	6	30	2011	7, 1, 2011
70	6	30	2012	7, 1, 2012
71	6	31	1812	Invalid date
72	6	31	1813	Invalid date
73	6	31	1912	Invalid date
74	6	31	2011	Invalid date
75	6	31	2012	Invalid date
76	11	1	1812	11, 2, 1812
77	11	1	1813	11, 2, 1813
78	11	1	1912	11, 2, 1912
79	11	1	2011	11, 2, 2011
80	11	1	2012	11, 2, 2012
81	11	2	1812	11, 3, 1812
82	11	2	1813	11, 3, 1813
83	11	2	1912	11, 3, 1912
84	11	2	2011	11, 3, 2011
85	11	2	2012	11, 3, 2012
86	11	15	1812	11, 16, 1812

(continued)

Table 5.3 Worst-Case Test Cases (Continued)

<i>Case</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected Output</i>
87	11	15	1813	11, 16, 1813
88	11	15	1912	11, 16, 1912
89	11	15	2011	11, 16, 2011
90	11	15	2012	11, 16, 2012
91	11	30	1812	12, 1, 1812
92	11	30	1813	12, 1, 1813
93	11	30	1912	12, 1, 1912
94	11	30	2011	12, 1, 2011
95	11	30	2012	12, 1, 2012
96	11	31	1812	Invalid date
97	11	31	1813	Invalid date
98	11	31	1912	Invalid date
99	11	31	2011	Invalid date
100	11	31	2012	Invalid date
101	12	1	1812	12, 2, 1812
102	12	1	1813	12, 2, 1813
103	12	1	1912	12, 2, 1912
104	12	1	2011	12, 2, 2011
105	12	1	2012	12, 2, 2012
106	12	2	1812	12, 3, 1812
107	12	2	1813	12, 3, 1813
108	12	2	1912	12, 3, 1912
109	12	2	2011	12, 3, 2011
110	12	2	2012	12, 3, 2012
111	12	15	1812	12, 16, 1812
112	12	15	1813	12, 16, 1813
113	12	15	1912	12, 16, 1912
114	12	15	2011	12, 16, 2011
115	12	15	2012	12, 16, 2012

(continued)

Table 5.3 Worst-Case Test Cases (Continued)

Case	Month	Day	Year	Expected Output
116	12	30	1812	12, 31, 1812
117	12	30	1813	12, 31, 1813
118	12	30	1912	12, 31, 1912
119	12	30	2011	12, 31, 2011
120	12	30	2012	12, 31, 2012
121	12	31	1812	1, 1, 1813
122	12	31	1813	1, 1, 1814
123	12	31	1912	1, 1, 1913
124	12	31	2011	1, 1, 2012
125	12	31	2012	1, 1, 2013

5.5.3 Test Cases for the Commission Problem

Instead of going through 125 boring test cases again, we will look at some more interesting test cases for the commission problem. This time, we will look at boundary values derived from the output range, especially near the threshold points of \$1000 and \$1800 where the commission percentage changes. The output space of the commission is shown in Figure 5.6. The intercepts of these threshold planes with the axes are shown.

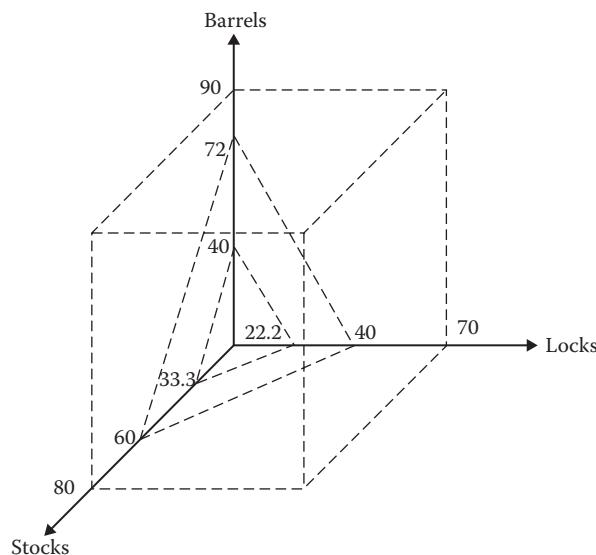
**Figure 5.6 Input space of the commission problem.**

Table 5.4 Output Boundary Value Analysis Test Cases

Case	Locks	Stocks	Barrels	Sales	Comm	Comment
1	1	1	1	100	10	Output minimum
2	1	1	2	125	12.5	Output minimum +
3	1	2	1	130	13	Output minimum +
4	2	1	1	145	14.5	Output minimum +
5	5	5	5	500	50	Midpoint
6	10	10	9	975	97.5	Border point -
7	10	9	10	970	97	Border point -
8	9	10	10	955	95.5	Border point -
9	10	10	10	1000	100	Border point
10	10	10	11	1025	103.75	Border point +
11	10	11	10	1030	104.5	Border point +
12	11	10	10	1045	106.75	Border point +
13	14	14	14	1400	160	Midpoint
14	18	18	17	1775	216.25	Border point -
15	18	17	18	1770	215.5	Border point -
16	17	18	18	1755	213.25	Border point -
17	18	18	18	1800	220	Border point
18	18	18	19	1825	225	Border point +
19	18	19	18	1830	226	Border point +
20	19	18	18	1845	229	Border point +
21	48	48	48	4800	820	Midpoint
22	70	80	89	7775	1415	Output maximum -
23	70	79	90	7770	1414	Output maximum -
24	69	80	90	7755	1411	Output maximum -
25	70	80	90	7800	1420	Output maximum

The volume between the origin and the lower plane corresponds to sales below the \$1000 threshold. The volume between the two planes is the 15% commission range. Part of the reason for using the output range to determine test cases is that cases from the input range are almost all in the 20% zone. We want to find input variable combinations that stress the sales/commission boundary values: \$100, \$1000, \$1800, and \$7800. The minimum and maximum were easy, and

Table 5.5 Output Special Value Test Cases

Case	<i>Locks</i>	<i>Stocks</i>	<i>Barrels</i>	<i>Sales</i>	<i>Comm</i>	<i>Comment</i>
1	10	11	9	1005	100.75	Border point +
2	18	17	19	1795	219.25	Border point -
3	18	19	17	1805	221	Border point +

the numbers happen to work out so that the border points are easy to generate. Here is where it gets interesting: test case 9 is the \$1000 border point. If we tweak the input variables, we get values just below and just above the border (cases 6–8 and 10–12). If we wanted to, we could pick values near the borders such as (22, 1, 1). As we continue in this way, we have a sense that we are “exercising” interesting parts of the code. We might claim that this is really a form of special value testing because we used our mathematical insight to generate test cases.

Table 5.4 contains test cases derived from boundary values on the output side of the commission function. Table 5.5 contains special value test cases.

5.6 Random Testing

At least two decades of discussion of random testing are included in the literature. Most of this interest is among academics, and in a statistical sense, it is interesting. Our three sample problems lend themselves nicely to random testing. The basic idea is that, rather than always choose the min, min+, nom, max-, and max values of a bounded variable, use a random number generator to pick test case values. This avoids any form of bias in testing. It also raises a serious question: how many random test cases are sufficient? Later, when we discuss structural test coverage metrics, we will have an elegant answer. For now, Tables 5.6 through 5.8 show the results of randomly generated test cases. They are derived from a Visual Basic application that picks values for a bounded variable $a \leq x \leq b$ as follows:

Table 5.6 Random Test Cases for Triangle Program

<i>Test Cases</i>	<i>Nontriangles</i>	<i>Scalene</i>	<i>Isosceles</i>	<i>Equilateral</i>
1289	663	593	32	1
15,436	7696	7372	367	1
17,091	8556	8164	367	1
2603	1284	1252	66	1
6475	3197	3122	155	1
5978	2998	2850	129	1
9008	4447	4353	207	1
Percentage	49.83%	47.87%	2.29%	0.01%

Table 5.7 Random Test Cases for Commission Program

Test Cases	10%	15%	20%
91	1	6	84
27	1	1	25
72	1	1	70
176	1	6	169
48	1	1	46
152	1	6	145
125	1	4	120
Percentage	1.01%	3.62%	95.37%

$$x = \text{Int}((b - a + 1) * \text{Rnd} + a)$$

where the function Int returns the integer part of a floating point number, and the function Rnd generates random numbers in the interval [0, 1]. The program keeps generating random test cases until at least one of each output occurs. In each table, the program went through seven “cycles” that ended with the “hard-to-generate” test case. In Tables 5.6 and 5.7, the last line shows what percentage of the random test cases was generated for each column. In the table for NextDate, the percentages are very close to the computed probability given in the last line of Table 5.8.

5.7 Guidelines for Boundary Value Testing

With the exception of special value testing, the test methods based on the input domain of a function (program) are the most rudimentary of all specification-based testing methods. They share the common assumption that the input variables are truly independent; and when this assumption is not warranted, the methods generate unsatisfactory test cases (such as June 31, 1912, for NextDate). Each of these methods can be applied to the output range of a program, as we did for the commission problem.

Another useful form of output-based test cases is for systems that generate error messages. The tester should devise test cases to check that error messages are generated when they are appropriate, and are not falsely generated. Boundary value analysis can also be used for internal variables, such as loop control variables, indices, and pointers. Strictly speaking, these are not input variables; however, errors in the use of these variables are quite common. Robustness testing is a good choice for testing internal variables.

There is a discussion in Chapter 10 about “the testing pendulum”—it refers to the problem of syntactic versus semantic approaches to developing test cases. Here is a short example given both ways. Consider a function F of three variables, a, b, and c. The boundaries are $0 \leq a < 10,000$, $0 \leq b < 10,000$, and $0 \leq c < 18.8$. The function F is $F = (a - b)/c$; Table 5.9 shows the normal boundary value test cases. Absent semantic knowledge, the first four test cases in Table 5.9 are what a boundary value testing tool would generate (a tool would not generate the expected output values). Even just the syntactic version is problematic—it does not avoid the division by zero possibility in test case 11.

Table 5.8 Random Test Cases for NextDate Program

<i>Test Cases</i>	<i>Days 1–30 of 31-Day Months</i>	<i>Day 31 of 31-Day Months</i>	<i>Days 1–29 of 30-Day Months</i>	<i>Day 30 of 30-Day Months</i>
913	542	17	274	10
1101	621	9	358	8
4201	2448	64	1242	46
1097	600	21	350	9
5853	3342	100	1804	82
3959	2195	73	1252	42
1436	786	22	456	13
Percentage	56.76%	1.65%	30.91%	1.13%
Probability	56.45%	1.88%	31.18%	1.88%
<i>Days 1–27 of Feb.</i>	<i>Feb. 28 of a Leap Year</i>	<i>Feb. 28 of a Non-Leap Year</i>	<i>Feb. 29 of a Leap Year</i>	<i>Impossible Days</i>
45	1	1	1	22
83	1	1	1	19
312	1	8	3	77
92	1	4	1	19
417	1	11	2	94
310	1	6	5	75
126	1	5	1	26
7.46%	0.04%	0.19%	0.08%	1.79%
7.26%	0.07%	0.20%	0.07%	1.01%

When we add the semantic information that F calculates the miles per gallon of an automobile, where a and b are end and start trip odometer values, and c is the gas tank capacity, we see more severe problems:

1. We must always have $a \geq b$. This will avoid the negative values of F (test cases 1, 2, 9, and 10).
2. Test cases 3, 8, and 12–15 all refer to trips of length 0, so they could be collapsed into one test case, probably test case 8.
3. Division by zero is an obvious problem, thereby eliminating test case 11. Applying the semantic knowledge will result in the better set of case cases in Table 5.10.
4. Table 5.10 is still problematic—we never see the effect of boundary values on the tank capacity.

Table 5.9 Normal Boundary Value Test Cases for $F = (a - b)/c$

<i>Test Case</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>F</i>
1	0	5000	9.4	-531.9
2	1	5000	9.4	-531.8
3	5000	5000	9.4	0.0
4	9998	5000	9.4	531.7
5	9999	5000	9.4	531.8
6	5000	0	9.4	531.9
7	5000	1	9.4	531.8
8	5000	5000	9.4	0.0
9	5000	9998	9.4	-531.7
10	5000	9999	9.4	-531.8
11	5000	5000	0	Undefined
12	5000	5000	1	0.0
13	5000	5000	9.4	0.0
14	5000	5000	18.7	0.0
15	5000	5000	18.8	0.0

Table 5.10 Semantic Boundary Value Test Cases for $F = (a - b)/c$

<i>Test Case</i>	<i>End Odometer</i>	<i>Start Odometer</i>	<i>Tank Capacity</i>	<i>Miles per Gallon</i>
4	9998	5000	9.4	531.7
5	9999	5000	9.4	531.8
6	5000	0	9.4	531.9
7	5000	1	9.4	531.8
8	5000	5000	9.4	0.0

EXERCISES

1. Develop a formula for the number of robustness test cases for a function of n variables.
2. Develop a formula for the number of robust worst-case test cases for a function of n variables.
3. Make a Venn diagram showing the relationships among test cases from boundary value analysis, robustness testing, worst-case testing, and robust worst-case testing.
4. What happens if we try to do output range robustness testing? Use the commission problem as an example.

5. If you did exercise 8 in Chapter 2, you are already familiar with the CRC Press website for downloads (<http://www.crcpress.com/product/isbn/9781466560680>). There you will find an Excel spreadsheet named specBasedTesting.xls. (It is an extended version of Naive.xls, and it contains the same inserted faults.) Different sheets contain worst-case boundary value test cases for the triangle, NextDate, and commission problems, respectively. Run these sets of test cases and compare the results with your naive testing from Chapter 2.
6. Apply special value testing to the miles per gallon example in Tables 5.9 and 5.10. Provide reasons for your chosen test cases.

Chapter 6

Equivalence Class Testing

The use of equivalence classes as the basis for functional testing has two motivations: we would like to have a sense of complete testing, and, at the same time, we would hope to avoid redundancy. Neither of these hopes is realized by boundary value testing—looking at the tables of test cases, it is easy to see massive redundancy, and looking more closely, serious gaps exist. Equivalence class testing echoes the two deciding factors of boundary value testing, robustness and the single/multiple fault assumption. This chapter presents the traditional view of equivalence class testing, followed by a coherent treatment of four distinct forms based on the two assumptions. The single versus multiple fault assumption yields the weak/strong distinction and the focus on invalid data yields a second distinction: normal versus robust. Taken together, these two assumptions result in Weak Normal, Strong Normal, Weak Robust, and Strong Robust Equivalence Class testing.

Two problems occur with robust forms. The first is that, very often, the specification does not define what the expected output for an invalid input should be. (We could argue that this is a deficiency of the specification, but that does not get us anywhere.) Thus, testers spend a lot of time defining expected outputs for these cases. The second problem is that strongly typed languages eliminate the need for the consideration of invalid inputs. Traditional equivalence testing is a product of the time when languages such as FORTRAN and COBOL were dominant; thus, this type of error was common. In fact, it was the high incidence of such errors that led to the implementation of strongly typed languages.

6.1 Equivalence Classes

In Chapter 3, we noted that the important aspect of equivalence classes is that they form a partition of a set, where partition refers to a collection of mutually disjoint subsets, the union of which is the entire set. This has two important implications for testing—the fact that the entire set is represented provides a form of completeness, and the disjointedness ensures a form of nonredundancy. Because the subsets are determined by an equivalence relation, the elements of a subset have something in common. The idea of equivalence class testing is to identify test cases by using one element from each equivalence class. If the equivalence classes are chosen wisely, this greatly

reduces the potential redundancy among test cases. In the triangle problem, for example, we would certainly have a test case for an equilateral triangle, and we might pick the triple (5, 5, 5) as inputs for a test case. If we did this, we would not expect to learn much from test cases such as (6, 6, 6) and (100, 100, 100). Our intuition tells us that these would be “treated the same” as the first test case; thus, they would be redundant. When we consider code-based testing in Chapters 8 and 9, we shall see that “treated the same” maps onto “traversing the same execution path.” The four forms of equivalence class testing all address the problems of gaps and redundancies that are common to the four forms of boundary value testing. Since the assumptions align, the four forms of boundary value testing also align with the four forms of equivalence class testing. There will be one point of overlap—this occurs when equivalence classes are defined by bounded variables. In such cases, a hybrid of boundary value and equivalence class testing is appropriate. The International Software Testing Qualifications Board (ISTQB) syllabi refer to this as “edge testing.” We will see this in the discussion in Section 6.3.

6.2 Traditional Equivalence Class Testing

Most of the standard testing texts (e.g., Myers, 1979; Mosley, 1993) discuss equivalence classes based on valid and invalid variable values. Traditional equivalence class testing is nearly identical to weak robust equivalence class testing (see Section 6.3.3). This traditional form focuses on invalid data values, and it is/was a consequence of the dominant style of programming in the 1960s and 1970s. Input data validation was an important issue at the time, and “Garbage In, Garbage Out” was the programmer’s watchword. In the early years, it was the program user’s responsibility to provide valid data. There was no guarantee about results based on invalid data. The term soon became known as GIGO. The usual response to GIGO was extensive input validation sections of a program. Authors and seminar leaders frequently commented that, in the classic afferent/central/efferent architecture of structured programming, the afferent portion often represented 80% of the total source code. In this context, it is natural to emphasize input data validation. Clearly, the defense against GIGO was to have extensive testing to assure data validity. The gradual shift to modern programming languages, especially those that feature strong data typing, and then to graphical user interfaces (GUIs) obviated much of the need for input data validation. Indeed, good use of user interface devices such as drop-down lists and slider bars reduces the likelihood of bad input data.

Traditional equivalence class testing echoes the process of boundary value testing. Figure 6.1 shows test cases for a function F of two variables x_1 and x_2 , as we had in Chapter 5. The extension to more realistic cases of n variables proceeds as follows:

1. Test F for valid values of all variables.
2. If step 1 is successful, then test F for invalid values of x_1 with valid values of the remaining variables. Any failure will be due to a problem with an invalid value of x_1 .
3. Repeat step 2 for the remaining variables.

One clear advantage of this process is that it focuses on finding faults due to invalid data. Since the GIGO concern was on invalid data, the kinds of combinations that we saw in the worst-case variations of boundary value testing were ignored. Figure 6.1 shows the five test cases for this process for our continuing function F of two variables.

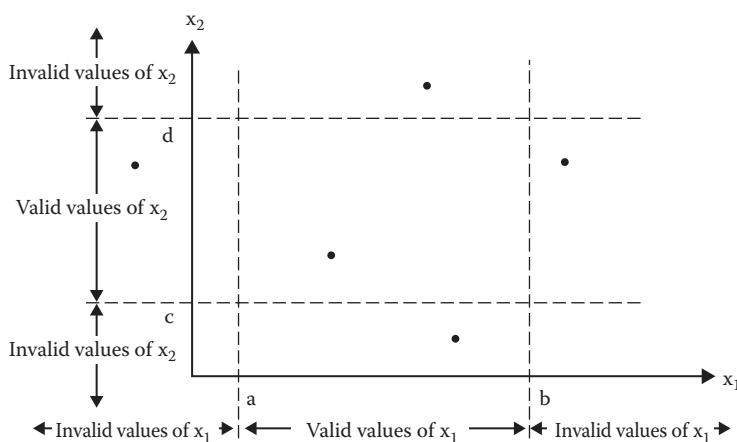


Figure 6.1 Traditional equivalence class test cases.

6.3 Improved Equivalence Class Testing

The key (and the craft!) of equivalence class testing is the choice of the equivalence relation that determines the classes. Very often, we make this choice by second-guessing the likely implementation and thinking about the functional manipulations that must somehow be present in the implementation. We will illustrate this with our continuing examples. We need to enrich the function we used in boundary value testing. Again, for the sake of comprehensible drawings, the discussion relates to a function, F, of two variables x_1 and x_2 . When F is implemented as a program, the input variables x_1 and x_2 will have the following boundaries, and intervals within the boundaries:

$$\begin{aligned} a \leq x_1 \leq d, \text{ with intervals } [a, b], [b, c], [c, d] \\ e \leq x_2 \leq g, \text{ with intervals } [e, f], [f, g] \end{aligned}$$

where square brackets and parentheses denote, respectively, closed and open interval endpoints. The intervals presumably correspond to some distinction in the program being tested, for example, the commission ranges in the commission problem. These ranges are equivalence classes. Invalid values of x_1 and x_2 are $x_1 < a$, $x_1 > d$, and $x_2 < e$, $x_2 > g$. The equivalence classes of valid values are

$$V1 = \{x_1: a \leq x_1 < b\}, V2 = \{x_1: b \leq x_1 < c\}, V3 = \{x_1: c \leq x_1 \leq d\}, V4 = \{x_2: e \leq x_2 < f\}, V5 = \{x_2: f \leq x_2 \leq g\}$$

The equivalence classes of invalid values are

$$NV1 = \{x_1: x_1 < a\}, NV2 = \{x_1: d < x_1\}, NV3 = \{x_2: x_2 < e\}, NV4 = \{x_2: g < x_2\}$$

The equivalence classes V1, V2, V3, V4, V5, NV1, NV2, NV3, and NV4 are disjoint, and their union is the entire plane. In the succeeding discussions, we will just use the interval notation rather than the full formal set definition.

6.3.1 Weak Normal Equivalence Class Testing

With the notation as given previously, weak normal equivalence class testing is accomplished by using one variable from each equivalence class (interval) in a test case. (Note the effect of the single fault assumption.) For the running example, we would end up with the three weak equivalence class test cases shown in Figure 6.2. This figure will be repeated for the remaining forms of equivalence class testing, but, for clarity, without the indication of valid and invalid ranges. These three test cases use one value from each equivalence class. The test case in the lower left rectangle corresponds to a value of x_1 in the class $[a, b]$, and to a value of x_2 in the class $[e, f]$. The test case in the upper center rectangle corresponds to a value of x_1 in the class $[b, c]$ and to a value of x_2 in the class $[f, g]$. The third test case could be in either rectangle on the right side of the valid values. We identified these in a systematic way, thus the apparent pattern. In fact, we will always have the same number of weak equivalence class test cases as classes in the partition with the largest number of subsets.

What can we learn from a weak normal equivalence class test case that fails, that is, one for which the expected and actual outputs are inconsistent? There could be a problem with x_1 , or a problem with x_2 , or maybe an interaction between the two. This ambiguity is the reason for the “weak” designation. If the expectation of failure is low, as it is for regression testing, this can be an acceptable choice. When more fault isolation is required, the stronger forms, discussed next, are indicated.

6.3.2 Strong Normal Equivalence Class Testing

Strong equivalence class testing is based on the multiple fault assumption, so we need test cases from each element of the Cartesian product of the equivalence classes, as shown in Figure 6.3. Notice the similarity between the pattern of these test cases and the construction of a truth table in propositional logic. The Cartesian product guarantees that we have a notion of “completeness” in two senses: we cover all the equivalence classes, and we have one of each possible combination of inputs. As we shall see from our continuing examples, the key to “good” equivalence class testing is the selection of the equivalence relation. Watch for the notion of inputs being “treated the same.” Most of the time, equivalence class testing defines classes of the input domain. There is no reason why we could not define equivalence relations on the output range of the program function being tested; in fact, this is the simplest approach for the triangle problem.

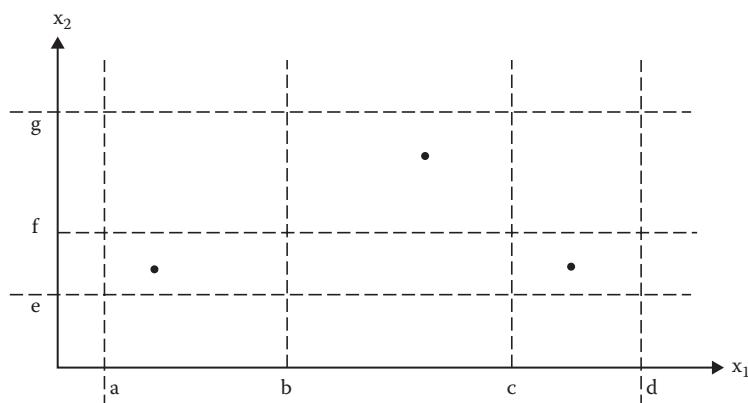


Figure 6.2 Weak normal equivalence class test cases.

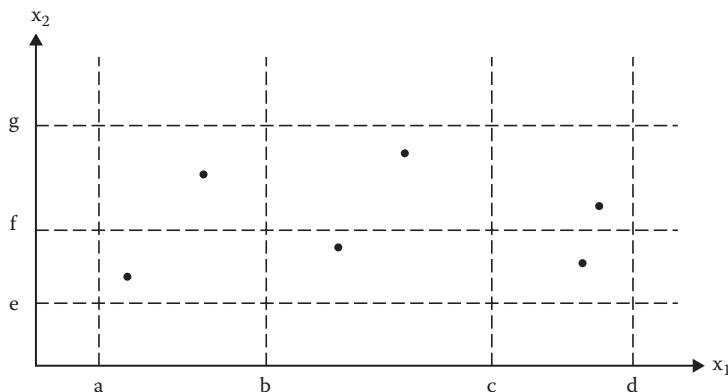


Figure 6.3 Strong normal equivalence class test cases.

6.3.3 Weak Robust Equivalence Class Testing

The name for this form is admittedly counterintuitive and oxymoronic. How can something be both weak and robust? The robust part comes from consideration of invalid values, and the weak part refers to the single fault assumption. The process of weak robust equivalence class testing is a simple extension of that for weak normal equivalence class testing—pick test cases such that each equivalence class is represented. In Figure 6.4, the test cases for valid classes are as those in Figure 6.2. The two additional test cases cover all four classes of invalid values. The process is similar to that for boundary value testing:

1. For valid inputs, use one value from each valid class (as in what we have called weak normal equivalence class testing). (Note that each input in these test cases will be valid.)
2. For invalid inputs, a test case will have one invalid value and the remaining values will all be valid. (Thus, a “single failure” should cause the test case to fail.)

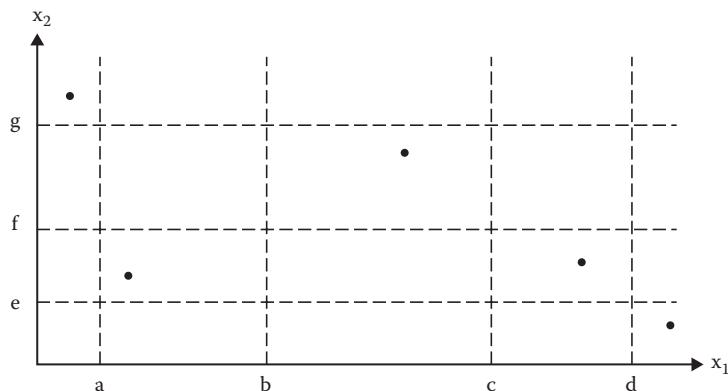


Figure 6.4 Weak robust equivalence class test cases.

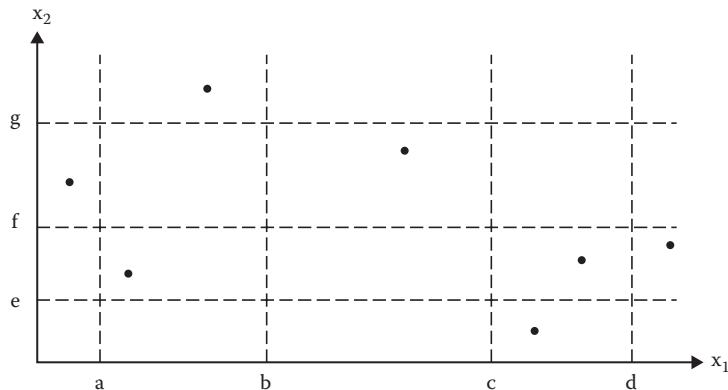


Figure 6.5 Revised weak robust equivalence class test cases.

The test cases resulting from this strategy are shown in Figure 6.4. There is a potential problem with these test cases. Consider the test cases in the upper left and lower right corners. Each of the test cases represents values from two invalid equivalence classes. Failure of either of these could be due to the interaction of two variables. Figure 6.5 presents a compromise between “pure” weak normal equivalence class testing and its robust extension.

6.3.4 Strong Robust Equivalence Class Testing

At least the name for this form is neither counterintuitive nor oxymoronic, just redundant. As before, the robust part comes from consideration of invalid values, and the strong part refers to the multiple fault assumption. We obtain test cases from each element of the Cartesian product of all the equivalence classes, both valid and invalid, as shown in Figure 6.6.

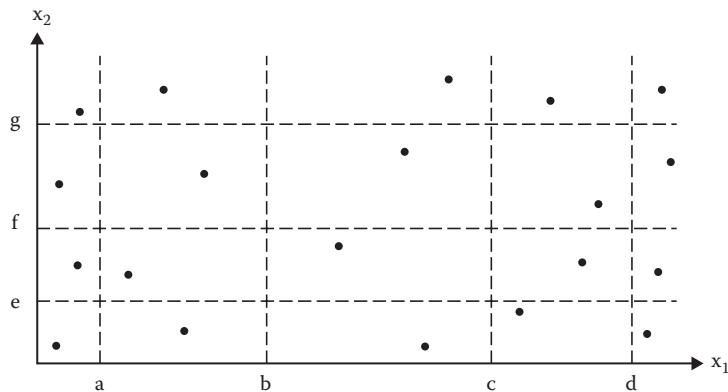


Figure 6.6 Strong robust equivalence class test cases.

6.4 Equivalence Class Test Cases for the Triangle Problem

In the problem statement, we note that four possible outputs can occur: NotATriangle, Scalene, Isosceles, and Equilateral. We can use these to identify output (range) equivalence classes as follows.

R1 = { a, b, c : the triangle with sides a, b , and c is equilateral}

R2 = { a, b, c : the triangle with sides a, b , and c is isosceles}

R3 = { a, b, c : the triangle with sides a, b , and c is scalene}

R4 = { a, b, c : sides a, b , and c do not form a triangle}

Four weak normal equivalence class test cases, chosen arbitrarily from each class are as follows:

Test Case	a	b	c	Expected Output
WN1	5	5	5	Equilateral
WN2	2	2	3	Isosceles
WN3	3	4	5	Scalene
WN4	4	1	2	Not a triangle

Because no valid subintervals of variables a, b , and c exist, the strong normal equivalence class test cases are identical to the weak normal equivalence class test cases.

Considering the invalid values for a, b , and c yields the following additional weak robust equivalence class test cases. (The invalid values could be zero, any negative number, or any number greater than 200.)

Test Case	a	b	c	Expected Output
WR1	-1	5	5	Value of a is not in the range of permitted values
WR2	5	-1	5	Value of b is not in the range of permitted values
WR3	5	5	-1	Value of c is not in the range of permitted values
WR4	201	5	5	Value of a is not in the range of permitted values
WR5	5	201	5	Value of b is not in the range of permitted values
WR6	5	5	201	Value of c is not in the range of permitted values

Here is one “corner” of the cube in three-space of the additional strong robust equivalence class test cases:

Test Case	a	b	c	Expected Output
SR1	-1	5	5	Value of a is not in the range of permitted values
SR2	5	-1	5	Value of b is not in the range of permitted values
SR3	5	5	-1	Value of c is not in the range of permitted values
SR4	-1	-1	5	Values of a, b are not in the range of permitted values
SR5	5	-1	-1	Values of b, c are not in the range of permitted values
SR6	-1	5	-1	Values of a, c are not in the range of permitted values
SR7	-1	-1	-1	Values of a, b, c are not in the range of permitted values

Notice how thoroughly the expected outputs describe the invalid input values.

Equivalence class testing is clearly sensitive to the equivalence relation used to define classes. Here is another instance of craftsmanship. If we base equivalence classes on the output domain, we obtain a richer set of test cases. What are some of the possibilities for the three integers, a, b, and c? They can all be equal, exactly one pair can be equal (this can happen in three ways), or none can be equal.

$$\begin{aligned} D1 &= \{<a, b, c>: a = b = c\} \\ D2 &= \{<a, b, c>: a = b, a \neq c\} \\ D3 &= \{<a, b, c>: a = c, a \neq b\} \\ D4 &= \{<a, b, c>: b = c, a \neq b\} \\ D5 &= \{<a, b, c>: a \neq b, a \neq c, b \neq c\} \end{aligned}$$

As a separate question, we can apply the triangle property to see if they even constitute a triangle. (For example, the triplet $<1, 4, 1>$ has exactly one pair of equal sides, but these sides do not form a triangle.)

$$\begin{aligned} D6 &= \{<a, b, c>: a \geq b + c\} \\ D7 &= \{<a, b, c>: b \geq a + c\} \\ D8 &= \{<a, b, c>: c \geq a + b\} \end{aligned}$$

If we wanted to be still more thorough, we could separate the “greater than or equal to” into the two distinct cases; thus, the set D6 would become

$$\begin{aligned} D6' &= \{<a, b, c>: a = b + c\} \\ D6'' &= \{<a, b, c>: a > b + c\} \end{aligned}$$

and similarly for D7 and D8.

6.5 Equivalence Class Test Cases for the NextDate Function

The NextDate function illustrates very well the craft of choosing the underlying equivalence relation. Recall that NextDate is a function of three variables: month, day, and year, and these have intervals of valid values defined as follows:

$$M1 = \{\text{month: } 1 \leq \text{month} \leq 12\}$$

$$D1 = \{\text{day: } 1 \leq \text{day} \leq 31\}$$

$$Y1 = \{\text{year: } 1812 \leq \text{year} \leq 2012\}$$

The invalid equivalence classes are

$$M2 = \{\text{month: } \text{month} < 1\}$$

$$M3 = \{\text{month: } \text{month} > 12\}$$

$$D2 = \{\text{day: } \text{day} < 1\}$$

$$D3 = \{\text{day: } \text{day} > 31\}$$

$$Y2 = \{\text{year: } \text{year} < 1812\}$$

$$Y3 = \{\text{year: } \text{year} > 2012\}$$

Because the number of valid classes equals the number of independent variables, only one weak normal equivalence class test case occurs, and it is identical to the strong normal equivalence class test case:

Case ID	Month	Day	Year	Expected Output
WN1, SN1	6	15	1912	6/16/1912

Here is the full set of weak robust test cases:

Case ID	Month	Day	Year	Expected Output
WR1	6	15	1912	6/16/1912
WR2	-1	15	1912	Value of month not in the range 1 ... 12
WR3	13	15	1912	Value of month not in the range 1 ... 12
WR4	6	-1	1912	Value of day not in the range 1 ... 31
WR5	6	32	1912	Value of day not in the range 1 ... 31
WR6	6	15	1811	Value of year not in the range 1812 ... 2012
WR7	6	15	2013	Value of year not in the range 1812 ... 2012

As with the triangle problem, here is one “corner” of the cube in three-space of the additional strong robust equivalence class test cases:

Case ID	Month	Day	Year	Expected Output
SR1	-1	15	1912	Value of month not in the range 1 ... 12
SR2	6	-1	1912	Value of day not in the range 1 ... 31
SR3	6	15	1811	Value of year not in the range 1812 ... 2012
SR4	-1	-1	1912	Value of month not in the range 1 ... 12 Value of day not in the range 1 ... 31
SR5	6	-1	1811	Value of day not in the range 1 ... 31 Value of year not in the range 1812 ... 2012
SR6	-1	15	1811	Value of month not in the range 1 ... 12 Value of year not in the range 1812 ... 2012
SR7	-1	-1	1811	Value of month not in the range 1 ... 12 Value of day not in the range 1 ... 31 Value of year not in the range 1812 ... 2012

If we more carefully choose the equivalence relation, the resulting equivalence classes will be more useful. Recall that earlier we said that the gist of the equivalence relation is that elements in a class are “treated the same way.” One way to see the deficiency of the traditional approach is that the “treatment” is at the valid/invalid level. We next reduce the granularity by focusing on more specific treatment.

What must be done to an input date? If it is not the last day of a month, the NextDate function will simply increment the day value. At the end of a month, the next day is 1 and the month is incremented. At the end of a year, both the day and the month are reset to 1, and the year is incremented. Finally, the problem of leap year makes determining the last day of a month interesting. With all this in mind, we might postulate the following equivalence classes:

M1 = {month: month has 30 days}
M2 = {month: month has 31 days}
M3 = {month: month is February}
D1 = {day: $1 \leq \text{day} \leq 28$ }
D2 = {day: day = 29}
D3 = {day: day = 30}
D4 = {day: day = 31}
Y1 = {year: year = 2000}
Y2 = {year: year is a non-century leap year}
Y3 = {year: year is a common year}

By choosing separate classes for 30- and 31-day months, we simplify the question of the last day of the month. By taking February as a separate class, we can give more attention to leap year

questions. We also give special attention to day values: days in D1 are (nearly) always incremented, while days in D4 only have meaning for months in M2. Finally, we have three classes of years, the special case of the year 2000, leap years, and non-leap years. This is not a perfect set of equivalence classes, but its use will reveal many potential errors.

6.5.1 Equivalence Class Test Cases

These classes yield the following weak normal equivalence class test cases. As before, the inputs are mechanically selected from the approximate middle of the corresponding class:

Case ID	Month	Day	Year	Expected Output
WN1	6	14	2000	6/15/2000
WN2	7	29	1996	7/30/1996
WN3	2	30	2002	Invalid input date
WN4	6	31	2000	Invalid input date

Mechanical selection of input values makes no consideration of our domain knowledge, thus the two impossible dates. This will always be a problem with “automatic” test case generation, because all of our domain knowledge is not captured in the choice of equivalence classes. The strong normal equivalence class test cases for the revised classes are as follows:

Case ID	Month	Day	Year	Expected Output
SN1	6	14	2000	6/15/2000
SN2	6	14	1996	6/15/1996
SN3	6	14	2002	6/15/2002
SN4	6	29	2000	6/30/2000
SN5	6	29	1996	6/30/1996
SN6	6	29	2002	6/30/2002
SN7	6	30	2000	Invalid input date
SN8	6	30	1996	Invalid input date
SN9	6	30	2002	Invalid input date
SN10	6	31	2000	Invalid input date
SN11	6	31	1996	Invalid input date
SN12	6	31	2002	Invalid input date
SN13	7	14	2000	7/15/2000
SN14	7	14	1996	7/15/1996

<i>Case ID</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected Output</i>
SN15	7	14	2002	7/15/2002
SN16	7	29	2000	7/30/2000
SN17	7	29	1996	7/30/1996
SN18	7	29	2002	7/30/2002
SN19	7	30	2000	7/31/2000
SN20	7	30	1996	7/31/1996
SN21	7	30	2002	7/31/2002
SN22	7	31	2000	8/1/2000
SN23	7	31	1996	8/1/1996
SN24	7	31	2002	8/1/2002
SN25	2	14	2000	2/15/2000
SN26	2	14	1996	2/15/1996
SN27	2	14	2002	2/15/2002
SN28	2	29	2000	3/1/2000
SN29	2	29	1996	3/1/1996
SN30	2	29	2002	Invalid input date
SN31	2	30	2000	Invalid input date
SN32	2	30	1996	Invalid input date
SN33	2	30	2002	Invalid input date
SN34	2	31	2000	Invalid input date
SN35	2	31	1996	Invalid input date
SN36	2	31	2002	Invalid input date

Moving from weak to strong normal testing raises some of the issues of redundancy that we saw with boundary value testing. The move from weak to strong, whether with normal or robust classes, always makes the presumption of independence, and this is reflected in the cross product of the equivalence classes. Three month classes times four day classes times three year classes results in 36 strong normal equivalence class test cases. Adding two invalid classes for each variable will result in 150 strong robust equivalence class test cases (too many to show here!).

We could also streamline our set of test cases by taking a closer look at the year classes. If we merge Y1 and Y2, and call the result the set of leap years, our 36 test cases would drop down to 24. This change suppresses special attention to considerations in the year 2000, and it also adds some complexity to the determination of which years are leap years. Balance this against how much might be learned from the present test cases.

6.6 Equivalence Class Test Cases for the Commission Problem

The input domain of the commission problem is “naturally” partitioned by the limits on locks, stocks, and barrels. These equivalence classes are exactly those that would also be identified by traditional equivalence class testing. The first class is the valid input; the other two are invalid. The input domain equivalence classes lead to very unsatisfactory sets of test cases. Equivalence classes defined on the output range of the commission function will be an improvement.

The valid classes of the input variables are

$$L1 = \{\text{locks: } 1 \leq \text{locks} \leq 70\}$$

$$L2 = \{\text{locks} = -1\} \text{ (occurs if locks} = -1 \text{ is used to control input iteration)}$$

$$S1 = \{\text{stocks: } 1 \leq \text{stocks} \leq 80\}$$

$$B1 = \{\text{barrels: } 1 \leq \text{barrels} \leq 90\}$$

The corresponding invalid classes of the input variables are

$$L3 = \{\text{locks: locks} = 0 \text{ OR locks} < -1\}$$

$$L4 = \{\text{locks: locks} > 70\}$$

$$S2 = \{\text{stocks: stocks} < 1\}$$

$$S3 = \{\text{stocks: stocks} > 80\}$$

$$B2 = \{\text{barrels: barrels} < 1\}$$

$$B3 = \{\text{barrels: barrels} > 90\}$$

One problem occurs, however. The variable “locks” is also used as a sentinel to indicate no more telegrams. When a value of -1 is given for locks, the while loop terminates, and the values of totalLocks, totalStocks, and totalBarrels are used to compute sales, and then commission.

Except for the names of the variables and the interval endpoint values, this is identical to our first version of the NextDate function. Therefore, we will have exactly one weak normal equivalence class test case—and again, it is identical to the strong normal equivalence class test case. Note that the case for locks = -1 just terminates the iteration. We will have eight weak robust test cases.

Case ID	Locks	Stocks	Barrels	Expected Output
WR1	10	10	10	\$100
WR2	-1	40	45	Program terminates
WR3	-2	40	45	Value of locks not in the range 1 ... 70
WR4	71	40	45	Value of locks not in the range 1 ... 70
WR5	35	-1	45	Value of stocks not in the range 1 ... 80
WR6	35	81	45	Value of stocks not in the range 1 ... 80
WR7	35	40	-1	Value of barrels not in the range 1 ... 90
WR8	35	40	91	Value of barrels not in the range 1 ... 90

Here is one “corner” of the cube in 3-space of the additional strong robust equivalence class test cases:

Case ID	Locks	Stocks	Barrels	Expected Output
SR1	-2	40	45	Value of locks not in the range 1 ... 70
SR2	35	-1	45	Value of stocks not in the range 1 ... 80
SR3	35	40	-2	Value of barrels not in the range 1 ... 90
SR4	-2	-1	45	Value of locks not in the range 1 ... 70 Value of stocks not in the range 1 ... 80
SR5	-2	40	-1	Value of locks not in the range 1 ... 70 Value of barrels not in the range 1 ... 90
SR6	35	-1	-1	Value of stocks not in the range 1 ... 80 Value of barrels not in the range 1 ... 90
SR7	-2	-1	-1	Value of locks not in the range 1 ... 70 Value of stocks not in the range 1 ... 80 Value of barrels not in the range 1 ... 90

Notice that, of strong test cases—whether normal or robust—only one is a legitimate input. If we were really worried about error cases, this might be a good set of test cases. It can hardly give us a sense of confidence about the calculation portion of the problem, however. We can get some help by considering equivalence classes defined on the output range. Recall that sales is a function of the number of locks, stocks, and barrels sold:

$$\text{Sales} = 45 \text{ locks} + 30 \text{ stocks} + 25 \text{ barrels}$$

We could define equivalence classes of three variables by commission ranges:

$$\begin{aligned} S1 &= \{\langle \text{locks}, \text{stocks}, \text{barrels} \rangle : \text{sales} \leq 1000\} \\ S2 &= \{\langle \text{locks}, \text{stocks}, \text{barrels} \rangle : 1000 < \text{sales} \leq 1800\} \\ S3 &= \{\langle \text{locks}, \text{stocks}, \text{barrels} \rangle : \text{sales} > 1800\} \end{aligned}$$

Figure 5.6 helps us get a better feel for the input space. Elements of S1 are points with integer coordinates in the pyramid near the origin. Elements of S2 are points in the “triangular slice” between the pyramid and the rest of the input space. Finally, elements of S3 are all those points in the rectangular volume that are not in S1 or in S2. All the error cases found by the strong equivalence classes of the input domain are outside of the rectangular space shown in Figure 5.6.

As was the case with the triangle problem, the fact that our input is a triplet means that we no longer take test cases from a Cartesian product.

Test Case	Locks	Stocks	Barrels	Sales	Commission
OR1	5	5	5	500	50
OR2	15	15	15	1500	175
OR3	25	25	25	2500	360

These test cases give us some sense that we are exercising important parts of the problem. Together with the weak robust test cases, we would have a pretty good test of the commission problem. We might want to add some boundary checking, just to make sure the transitions at sales of \$1000 and \$1800 are correct. This is not particularly easy because we can only choose values of locks, stocks, and barrels. It happens that the constants in this example are contrived so that there are “nice” triplets.

6.7 Edge Testing

The *ISTQB Advanced Level Syllabus* (ISTQB, 2012) describes a hybrid of boundary value analysis and equivalence class testing and gives it the name “edge testing.” The need for this occurs when contiguous ranges of a particular variable constitute equivalence classes. Figure 6.2 shows three equivalence classes of valid values for x_1 and two classes for x_2 . Presumably, these classes refer to variables that are “treated the same” in some application. This suggests that there may be faults near the boundaries of the classes, and edge testing will exercise these potential faults. For the example in Figure 6.2, a full set of edge testing test values are as follows:

Normal test values for x_1 : {a, a+, b−, b, b+, c−, c, c+, d−, d}
 Robust test values for x_1 : {a−, a, a+, b−, b, b+, c−, c, c+, d−, d, d+}
 Normal test values for x_2 : {e, e+, f−, f, f+, g−, g}
 Robust test values for x_2 : {e−, e, e+, f−, f, f+, g−, g, g+}

One subtle difference is that edge test values do not include the nominal values that we had with boundary value testing. Once the sets of edge values are determined, edge testing can follow any of the four forms of equivalence class testing. The numbers of test cases obviously increase as with the variations of boundary value and equivalence class testing.

6.8 Guidelines and Observations

Now that we have gone through three examples, we conclude with some observations about, and guidelines for, equivalence class testing.

1. Obviously, the weak forms of equivalence class testing (normal or robust) are not as comprehensive as the corresponding strong forms.
2. If the implementation language is strongly typed (and invalid values cause run-time errors), it makes no sense to use the robust forms.

3. If error conditions are a high priority, the robust forms are appropriate.
4. Equivalence class testing is appropriate when input data is defined in terms of intervals and sets of discrete values. This is certainly the case when system malfunctions can occur for out-of-limit variable values.
5. Equivalence class testing is strengthened by a hybrid approach with boundary value testing. (We can “reuse” the effort made in defining the equivalence classes.)
6. Equivalence class testing is indicated when the program function is complex. In such cases, the complexity of the function can help identify useful equivalence classes, as in the NextDate function.
7. Strong equivalence class testing makes a presumption that the variables are independent, and the corresponding multiplication of test cases raises issues of redundancy. If any dependencies occur, they will often generate “error” test cases, as they did in the NextDate function. (The decision table technique in Chapter 7 resolves this problem.)
8. Several tries may be needed before the “right” equivalence relation is discovered, as we saw in the NextDate example. In other cases, there is an “obvious” or “natural” equivalence relation. When in doubt, the best bet is to try to second-guess aspects of any reasonable implementation. This is sometimes known as the “competent programmer hypothesis.”
9. The difference between the strong and weak forms of equivalence class testing is helpful in the distinction between progression and regression testing.

EXERCISES

1. Starting with the 36 strong normal equivalence class test cases for the NextDate function, revise the day classes as discussed, and then find the other nine test cases.
2. If you use a compiler for a strongly typed language, discuss how it would react to robust equivalence class test cases.
3. Revise the set of weak normal equivalence classes for the extended triangle problem that considers right triangles.
4. Compare and contrast the single/multiple fault assumption with boundary value and equivalence class testing.
5. The spring and fall changes between standard and daylight savings time create an interesting problem for telephone bills. In the spring, this switch occurs at 2:00 a.m. on a Sunday morning (late March, early April) when clocks are reset to 3:00 a.m. The symmetric change takes place usually on the last Sunday in October, when the clock changes from 2:59:59 back to 2:00:00.

Develop equivalence classes for a long-distance telephone service function that bills calls using the following rate structure:

Call duration \leq 20 minutes charged at \$0.05 per minute or fraction of a minute

Call duration $>$ 20 minutes charged at \$1.00 plus \$0.10 per minute or fraction of a minute in excess of 20 minutes.

Make these assumptions:

- Chargeable time of a call begins when the called party answers, and ends when the calling party disconnects.
 - Call durations of seconds are rounded up to the next larger minute.
 - No call lasts more than 30 hours.
6. If you did exercise 8 in Chapter 2, and exercise 5 in Chapter 5, you are already familiar with the CRC Press website for downloads (<http://www.crcpress.com/product/isbn/97818466560680>). There you will find an Excel spreadsheet named specBasedTesting.xls.

(It is an extended version of Naive.xls, and it contains the same inserted faults.) Different sheets contain strong, normal equivalence class test cases for the triangle, NextDate, and commission problems, respectively. Run these sets of test cases and compare the results with your naive testing from Chapter 2 and your boundary value testing from Chapter 5.

References

- ISTQB Advanced Level Working Party, *ISTQB Advanced Level Syllabus*, 2012.
Mosley, D.J., *The Handbook of MIS Application Software Testing*, Yourdon Press, Prentice Hall, Englewood Cliffs, NJ, 1993.
Myers, G.J., *The Art of Software Testing*, Wiley Interscience, New York, 1979.

Chapter 7

Decision Table–Based Testing

Of all the functional testing methods, those based on decision tables are the most rigorous because of their strong logical basis. Two closely related methods are used: cause-and-effect graphing (Elmendorf, 1973; Myers, 1979) and the decision tableau method (Mosley, 1993). These are more cumbersome to use and are fully redundant with decision tables; both are covered in Mosley (1993). For the curious, or for the sake of completeness, Section 7.5 offers a short discussion of cause-and-effect graphing.

7.1 Decision Tables

Decision tables have been used to represent and analyze complex logical relationships since the early 1960s. They are ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions. Some of the basic decision table terms are illustrated in Table 7.1.

A decision table has four portions: the part to the left of the bold vertical line is the stub portion; to the right is the entry portion. The part above the bold horizontal line is the condition portion, and below is the action portion. Thus, we can refer to the condition stub, the condition entries, the action stub, and the action entries. A column in the entry portion is a rule. Rules indicate which actions, if any, are taken for the circumstances indicated in the condition portion of the rule. In the decision table in Table 7.1, when conditions c1, c2, and c3 are all true, actions a1 and a2 occur. When c1 and c2 are both true and c3 is false, then actions a1 and a3 occur. The entry for c3 in the rule where c1 is true and c2 is false is called a “don’t care” entry. The don’t care entry has two major interpretations: the condition is irrelevant, or the condition does not apply. Sometimes people will enter the “n/a” symbol for this latter interpretation.

When we have binary conditions (true/false, yes/no, 0/1), the condition portion of a decision table is a truth table (from propositional logic) that has been rotated 90°. This structure guarantees that we consider every possible combination of condition values. When we use decision tables for test case identification, this completeness property of a decision table guarantees a form of complete testing. Decision tables in which all the conditions are binary are called Limited Entry Decision Tables (LETDs). If conditions are allowed to have several values, the resulting tables

Table 7.1 Portions of a Decision Table

<i>Stub</i>	<i>Rule 1</i>	<i>Rule 2</i>	<i>Rules 3, 4</i>	<i>Rule 5</i>	<i>Rule 6</i>	<i>Rules 7, 8</i>
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	—	T	F	—
a1	X	X		X		
a2	X				X	
a3		X		X		
a4			X			X

are called Extended Entry Decision Tables (EEDTs). We will see examples of both types for the NextDate problem. Decision tables are deliberately declarative (as opposed to imperative); no particular order is implied by the conditions, and selected actions do not occur in any particular order.

7.2 Decision Table Techniques

To identify test cases with decision tables, we interpret conditions as inputs and actions as outputs. Sometimes conditions end up referring to equivalence classes of inputs, and actions refer to major functional processing portions of the item tested. The rules are then interpreted as test cases. Because the decision table can mechanically be forced to be complete, we have some assurance that we will have a comprehensive set of test cases. Several techniques that produce decision tables are more useful to testers. One helpful style is to add an action to show when a rule is logically impossible. In the decision table in Table 7.2, we see examples of don't care entries and impossible rule usage. If the integers a, b, and c do not constitute a triangle, we do not even care about possible

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		

equalities, as indicated in the first rule. In rules 3, 4, and 6, if two pairs of integers are equal, by transitivity, the third pair must be equal; thus, the negative entry makes these rules impossible.

The decision table in Table 7.3 illustrates another consideration: the choice of conditions can greatly expand the size of a decision table. Here, we have expanded the old condition (c1: a, b, c form a triangle?) to a more detailed view of the three inequalities of the triangle property. If any one of these fails, the three integers do not constitute sides of a triangle.

We could expand this still further because there are two ways an inequality could fail: one side could equal the sum of the other two, or it could be strictly greater.

When conditions refer to equivalence classes, decision tables have a characteristic appearance. Conditions in the decision table in Table 7.4 are from the NextDate problem; they refer to the mutually exclusive possibilities for the month variable. Because a month is in exactly one equivalence class, we cannot ever have a rule in which two entries are true. The don't care entries (—) really mean “must be false.” Some decision table aficionados use the notation $F!$ to make this point.

Use of don't care entries has a subtle effect on the way in which complete decision tables are recognized. For a limited entry decision table with n conditions, there must be 2^n independent

Table 7.3 Refined Decision Table for Triangle Problem

c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c?	—	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
a1: Not a triangle	X	X	X								
a2: Scalene											X
a3: Isosceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

Table 7.4 Decision Table with Mutually Exclusive Conditions

Conditions	R1	R2	R3
c1: Month in M1?	T	—	—
c2: Month in M2?	—	T	—
c3: Month in M3?	—	—	T
a1			
a2			
a3			

rules. When don't care entries really indicate that the condition is irrelevant, we can develop a rule count as follows: rules in which no don't care entries occur count as one rule, and each don't care entry in a rule doubles the count of that rule. The rule counts for the decision table in Table 7.3 are shown in Table 7.5. Notice that the sum of the rule counts is 64 (as it should be).

If we applied this simplistic algorithm to the decision table in Table 7.4, we get the rule counts shown in Table 7.6. We should only have eight rules, so we clearly have a problem. To see where the problem lies, we expand each of the three rules, replacing the “—” entries with the T and F possibilities, as shown in Table 7.7.

Notice that we have three rules in which all entries are T: rules 1.1, 2.1, and 3.1. We also have two rules with T, T, F entries: rules 1.2 and 2.2. Similarly, rules 1.3 and 3.2 are identical; so are rules 2.3 and 3.3. If we delete the repetitions, we end up with seven rules; the missing rule is the one in which all conditions are false. The result of this process is shown in Table 7.8. The impossible rules are also shown.

Table 7.5 Decision Table for Table 7.3 with Rule Counts

c1: $a < b + c$?	F	T	T	T	T	T	T	T	T	T	T	T
c2: $b < a + c$?	—	F	T	T	T	T	T	T	T	T	T	T
c3: $c < a + b$?	—	—	F	T	T	T	T	T	T	T	T	T
c4: $a = b$?	—	—	—	T	T	T	T	F	F	F	F	F
c5: $a = c$?	—	—	—	T	T	F	F	T	T	F	F	F
c6: $b = c$?	—	—	—	T	F	T	F	T	F	T	F	F
Rule count	32	16	8	1	1	1	1	1	1	1	1	1
a1: Not a triangle	X	X	X									
a2: Scalene												X
a3: Isosceles							X		X	X		
a4: Equilateral				X								
a5: Impossible					X	X		X				

Table 7.6 Rule Counts for a Decision Table with Mutually Exclusive Conditions

Conditions	R1	R2	R3
c1: Month in M1	T	—	—
c2: Month in M2	—	T	—
c3: Month in M3	—	—	T
Rule count	4	4	4
a1			

Table 7.7 Impossible Rules in Table 7.7

Conditions	1.1	1.2	1.3	1.4	2.1	2.2	2.3	2.4	3.1	3.2	3.3	3.4
c1: Month in M1	T	T	T	T	T	T	F	F	T	T	F	F
c2: Month in M2	T	T	F	F	T	T	T	T	F	T	T	F
c3: Month in M3	T	F	T	F	T	F	T	F	T	T	T	T
Rule count	1	1	1	1	1	1	1	1	1	1	1	1
a1: Impossible	X	X	X	—	X	X	X	—	X	X	—	—

Table 7.8 Mutually Exclusive Conditions with Impossible Rules

	1.1	1.2	1.3	1.4	2.3	2.4	3.4	
c1: Month in M1	T	T	T	T	F	F	F	F
c2: Month in M2	T	T	F	F	T	T	F	F
c3: Month in M3	T	F	T	F	T	F	T	F
Rule count	1	1	1	1	1	1	1	1
a1: Impossible	X	X	X	—	X	—	—	X

Table 7.9 A Redundant Decision Table

Conditions	1–4	5	6	7	8	9
c1	T	F	F	F	F	T
c2	—	T	T	F	F	F
c3	—	T	F	T	F	F
a1	X	X	X	—	—	X
a2	—	X	X	X	—	—
a3	X	—	X	X	X	X

The ability to recognize (and develop) complete decision tables puts us in a powerful position with respect to redundancy and inconsistency. The decision table in Table 7.9 is redundant—three conditions and nine rules exist. (Rule 9 is identical to rule 4.) Notice that the action entries in rule 9 are identical to those in rules 1–4. As long as the actions in a redundant rule are identical to the corresponding part of the decision table, we do not have much of a problem. If the action entries are different, as in Table 7.10, we have a bigger problem.

If the decision table in Table 7.10 were to process a transaction in which c1 is true and both c2 and c3 are false, both rules 4 and 9 apply. We can make two observations:

1. Rules 4 and 9 are inconsistent.
2. The decision table is nondeterministic.

Table 7.10 An Inconsistent Decision Table

<i>Conditions</i>	1–4	5	6	7	8	9
c1	T	F	F	F	F	T
c2	—	T	T	F	F	F
c3	—	T	F	T	F	F
a1	X	X	X	—	—	—
a2	—	X	X	X	—	X
a3	X	—	X	X	X	—

Rules 4 and 9 are inconsistent because the action sets are different. The whole table is non-deterministic because there is no way to decide whether to apply rule 4 or rule 9. The bottom line for testers is that care should be taken when don't care entries are used in a decision table.

7.3 Test Cases for the Triangle Problem

Using the decision table in Table 7.3, we obtain 11 functional test cases: three impossible cases, three ways to fail the triangle property, one way to get an equilateral triangle, one way to get a scalene triangle, and three ways to get an isosceles triangle (see Table 7.11). We still need to provide

Table 7.11 Test Cases from Table 7.3

<i>Case ID</i>	a	b	c	<i>Expected Output</i>
DT1	4	1	2	Not a triangle
DT2	1	4	2	Not a triangle
DT3	1	2	4	Not a triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

actual values for the variables in the conditions, but we cannot do this for the impossible rules. If we extended the decision table to show both ways to fail an inequality, we would pick up three more test cases (where one side is exactly the sum of the other two). Some judgment is required in this because of the exponential growth of rules. In this case, we would end up with many more don't care entries and more impossible rules.

7.4 Test Cases for the NextDate Function

The NextDate function was chosen because it illustrates the problem of dependencies in the input domain. This makes it a perfect example for decision table-based testing, because decision tables can highlight such dependencies. Recall that, in Chapter 6, we identified equivalence classes in the input domain of the NextDate function. One of the limitations we found in Chapter 6 was that indiscriminate selection of input values from the equivalence classes resulted in “strange” test cases, such as finding the next date to June 31, 1812. The problem stems from the presumption that the variables are independent. If they are, a Cartesian product of the classes makes sense. When logical dependencies exist among variables in the input domain, these dependencies are lost (suppressed is better) in a Cartesian product. The decision table format lets us emphasize such dependencies using the notion of the “impossible” action to denote impossible combinations of conditions (which are actually impossible rules). In this section, we will make three tries at a decision table formulation of the NextDate function.

7.4.1 First Try

Identifying appropriate conditions and actions presents an opportunity for craftsmanship. Suppose we start with a set of equivalence classes close to the one we used in Chapter 6.

```
M1 = {month: month has 30 days}
M2 = {month: month has 31 days}
M3 = {month: month is February}
D1 = {day: 1 ≤ day ≤ 28}
D2 = {day: day = 29}
D3 = {day: day = 30}
D4 = {day: day = 31}
Y1 = {year: year is a leap year}
Y2 = {year: year is not a leap year}
```

If we wish to highlight impossible combinations, we could make a limited entry decision table with the following conditions and actions. (Note that the equivalence classes for the year variable collapse into one condition in Table 7.12.)

This decision table will have 256 rules, many of which will be impossible. If we wanted to show why these rules were impossible, we might revise our actions to the following:

- a1: Day invalid for this month
- a2: Cannot happen in a non-leap year
- a3: Compute the next date

Table 7.12 First Try Decision Table with 256 Rules

<i>Conditions</i>									
c1: Month in M1?	T								
c2: Month in M2?		T							
c3: Month in M3?			T						
c4: Day in D1?									
c5: Day in D2?									
c6: Day in D3?									
c7: Day in D4?									
c8: Year in Y1?									
a1: Impossible									
a2: Next date									

7.4.2 Second Try

If we focus on the leap year aspect of the `NextDate` function, we could use the set of equivalence classes as they were in Chapter 6. These classes have a Cartesian product that contains 36 triples, with several that are impossible.

To illustrate another decision table technique, this time we will develop an extended entry decision table, and we will take a closer look at the action stub. In making an extended entry decision table, we must ensure that the equivalence classes form a true partition of the input domain. (Recall from Chapter 3 that a partition is a set of disjoint subsets where the union is the entire set.) If there were any “overlaps” among the rule entries, we would have a redundant case in which more than one rule could be satisfied. Here, Y_2 is the set of years between 1812 and 2012, evenly divisible by four excluding the year 2000.

$M_1 = \{\text{month: month has 30 days}\}$
 $M_2 = \{\text{month: month has 31 days}\}$
 $M_3 = \{\text{month: month is February}\}$
 $D_1 = \{\text{day: } 1 \leq \text{day} \leq 28\}$
 $D_2 = \{\text{day: day} = 29\}$
 $D_3 = \{\text{day: day} = 30\}$
 $D_4 = \{\text{day: day} = 31\}$
 $Y_1 = \{\text{year: year} = 2000\}$
 $Y_2 = \{\text{year: year is a non-century leap year}\}$
 $Y_3 = \{\text{year: year is a common year}\}$

In a sense, we could argue that we have a “gray box” technique, because we take a closer look at the `NextDate` problem statement. To produce the next date of a given date, only five possible actions are needed: incrementing and resetting the day and month, and incrementing the year.

(We will not let time go backward by resetting the year.) To follow the metaphor, we still cannot see inside the implementation box—the implementation could be a table look-up.

These conditions would result in a decision table with 36 rules that correspond to the Cartesian product of the equivalence classes. Combining rules with don't care entries yields the decision table in Table 7.13, which has 16 rules. We still have the problem with logically impossible rules, but this formulation helps us identify the expected outputs of a test case. If you complete the action entries in this table, you will find some cumbersome problems with December (in rule 8) and other problems with Feb. 28 in rules 9, 11, and 12. We fix these next.

Table 7.13 Second Try Decision Table with 36 Rules

	1	2	3	4	5	6	7	8
c1: Month in	M1	M1	M1	M1	M2	M2	M2	M2
c2: Day in	D1	D2	D3	D4	D1	D2	D3	D4
c3: Year in	—	—	—	—	—	—	—	—
Rule count	3	3	3	3	3	3	3	3
Actions								
a1: Impossible				X				
a2: Increment day	X	X			X	X	X	
a3: Reset day			X					X
a4: Increment month				X				?
a5: Reset month								?
a6: Increment year								?
	9	10	11	12	13	14	15	16
c1: Month in	M3							
c2: Day in	D1	D1	D1	D2	D2	D2	D3	D4
c3: Year in	Y1	Y2	Y3	Y1	Y2	Y3	—	—
Rule count	1	1	1	1	1	1	3	3
Actions								
a1: Impossible						X	X	X
a2: Increment day	X	X	?					
a3: Reset day			?	X	X			
a4: Increment month	X		X	X	X			
a5: Reset month								
a6: Increment year								

7.4.3 Third Try

We can clear up the end-of-year considerations with a third set of equivalence classes. This time, we are very specific about days and months, and we revert to the simpler leap year or non-leap year condition of the first try—so the year 2000 gets no special attention. (We could do a fourth try, showing year equivalence classes as in the second try, but by now you get the point.)

```

M1 = {month: month has 30 days}
M2 = {month: month has 31 days except December}
M3 = {month: month is December}
M4 = {month: month is February}
D1 = {day: 1 ≤ day ≤ 27}
D2 = {day: day = 28}
D3 = {day: day = 29}
D4 = {day: day = 30}
D5 = {day: day = 31}
Y1 = {year: year is a leap year}
Y2 = {year: year is a common year}

```

The Cartesian product of these contains 40 elements. The result of combining rules with don't care entries is given in Table 7.14; it has 22 rules, compared with the 36 of the second try. Recall from Chapter 1 the question of whether a large set of test cases is necessarily better than a smaller set. Here, we have a 22-rule decision table that gives a clearer picture of the NextDate function than does the 36-rule decision table. The first five rules deal with 30-day months; notice that the leap year considerations are irrelevant. The next two sets of rules (6–15) deal with 31-day months, where rules 6–10 deal with months other than December and rules 11–15 deal with December. No impossible rules are listed in this portion of the decision table, although there is some redundancy that an efficient tester might question. Eight of the 10 rules simply increment the day. Would we really require eight separate test cases for this subfunction? Probably not; but note the insights we can get from the decision table. Finally, the last seven rules focus on February in common and leap years.

The decision table in Table 7.14 is the basis for the source code for the NextDate function in Chapter 2. As an aside, this example shows how good testing can improve programming. All the decision table analysis could have been done during the detailed design of the NextDate function.

We can use the algebra of decision tables to further simplify these 22 test cases. If the action sets of two rules in a limited entry decision table are identical, there must be at least one condition that allows two rules to be combined with a don't care entry. This is the decision table equivalent of the “treated the same” guideline that we used to identify equivalence classes. In a sense, we are identifying equivalence classes of rules. For example, rules 1, 2, and 3 involve day classes D1, D2, and D3 for 30-day months. These can be combined similarly for day classes D1, D2, D3, and D4 in the 31-day month rules, and D4 and D5 for February. The result is in Table 7.15.

The corresponding test cases are shown in Table 7.16.

Table 7.14 Decision Table for NextDate Function

	1	2	3	4	5	6	7	8	9	10		
c1: Month in	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2		
c2: Day in	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5		
c3: Year in	—	—	—	—	—	—	—	—	—	—		
Actions												
a1: Impossible					X							
a2: Increment day	X	X	X			X	X	X	X			
a3: Reset day				X						X		
a4: Increment month				X						X		
a5: Reset month												
a6: Increment year												
	11	12	13	14	15	16	17	18	19	20	21	22
c1: Month in	M3	M3	M3	M3	M3	M4						
c2: Day in	D1	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D5
c3: Year in	—	—	—	—	—	—	Y1	Y2	Y1	Y2	—	—
Actions												
a1: Impossible										X	X	X
a2: Increment day	X	X	X	X		X	X					
a3: Reset day					X			X	X			
a4: Increment month								X	X			
a5: Reset month					X							
a6: Increment year					X							

7.5 Test Cases for the Commission Problem

The commission problem is not well served by a decision table analysis. This is not surprising because very little decisional logic is used in the problem. Because the variables in the equivalence classes are truly independent, no impossible rules will occur in a decision table in which conditions correspond to the equivalence classes. Thus, we will have the same test cases as we did for equivalence class testing.

Table 7.15 Reduced Decision Table for NextDate Function

	1–3	4	5	6–9	10			
c1: Month in	M1	M1	M1	M2	M2			
c2: Day in	D1, D2, D3	D4	D5	D1, D2, D3, D4	D5			
c3: Year in	—	—	—	—	—			
Actions								
a1: Impossible			X					
a2: Increment day	X			X				
a3: Reset day		X			X			
a4: Increment month		X			X			
a5: Reset month								
a6: Increment year								
	11–14	15	16	17	18	19	20	21, 22
c1: Month in	M3	M3	M4	M4	M4	M4	M4	M4
c2: Day in	D1, D2, D3, D4	D5	D1	D2	D2	D3	D3	D4, D5
c3: Year in	—	—	—	Y1	Y2	Y1	Y2	—
Actions								
a1: Impossible							X	X
a2: Increment day	X		X	X				
a3: Reset day		X			X	X		
a4: Increment month					X	X		
a5: Reset month		X						
a6: Increment year		X						

7.6 Cause-and-Effect Graphing

In the early years of computing, the software community borrowed many ideas from the hardware community. In some cases this worked well, but in others, the problems of software just did not fit well with established hardware techniques. Cause-and-effect graphing is a good example of this. The base hardware concept was the practice of describing circuits composed of discrete components with AND, OR, and NOT gates. There was usually an input side of a circuit diagram, and

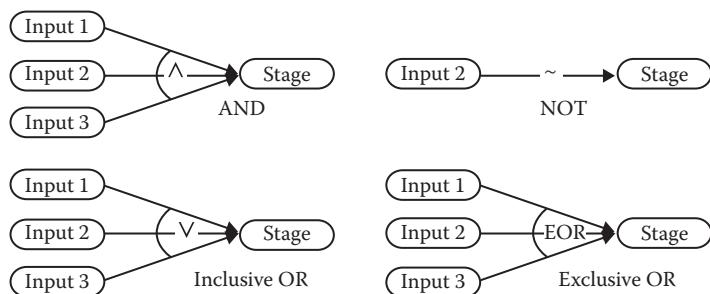
Table 7.16 Decision Table Test Cases for NextDate

Case ID	Month	Day	Year	Expected Output
1–3	4	15	2001	4/16/2001
4	4	30	2001	5/1/2001
5	4	31	2001	Invalid input date
6–9	1	15	2001	1/16/2001
10	1	31	2001	2/1/2001
11–14	12	15	2001	12/16/2001
15	12	31	2001	1/1/2002
16	2	15	2001	2/16/2001
17	2	28	2004	2/29/2004
18	2	28	2001	3/1/2001
19	2	29	2004	3/1/2004
20	2	29	2001	Invalid input date
21, 22	2	30	2001	Invalid input date

the flow of inputs through the various components could be generally traced from left to right. With this, the effects of hardware faults such as stuck-at-one/zero could be traced to the output side. This greatly facilitated circuit testing.

Cause-and-effect graphs attempt to follow this pattern, by showing unit inputs on the left side of a drawing, and using AND, OR, and NOT “gates” to express the flow of data across stages of a unit. Figure 7.1 shows the basic cause-and-effect graph structures. The basic structures can be augmented by less used operations: Identity, Masks, Requires, and Only One.

The most that can be learned from a cause-and-effect graph is that, if there is a problem at an output, the path(s) back to the inputs that affected the output can be retraced. There is little support for actually identifying test cases. Figure 7.2 shows a cause-and-effect graph for the commission problem.

**Figure 7.1 Cause-and-effect graphing operations.**

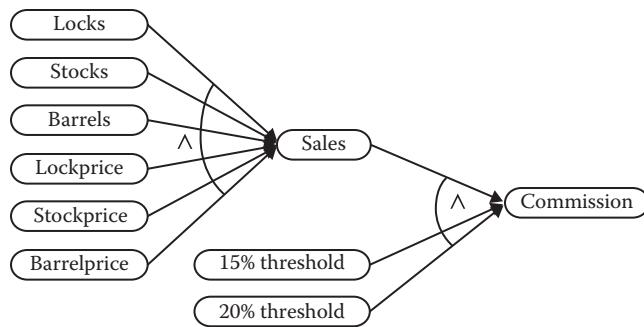


Figure 7.2 Cause-and-effect graph for commission problem.

7.7 Guidelines and Observations

As with the other testing techniques, decision table–based testing works well for some applications (such as NextDate) and is not worth the trouble for others (such as the commission problem). Not surprisingly, the situations in which it works well are those in which a lot of decision making takes place (such as the triangle problem), and those in which important logical relationships exist among input variables (the NextDate function).

1. The decision table technique is indicated for applications characterized by any of the following:
 - a. Prominent if–then–else logic
 - b. Logical relationships among input variables
 - c. Calculations involving subsets of the input variables
 - d. Cause-and-effect relationships between inputs and outputs
 - e. High cyclomatic complexity (see Chapter 9)
2. Decision tables do not scale up very well (a limited entry table with n conditions has 2^n rules). There are several ways to deal with this—use extended entry decision tables, algebraically simplify tables, “factor” large tables into smaller ones, and look for repeating patterns of condition entries. Try factoring the extended entry table for NextDate (Table 7.14).
3. As with other techniques, iteration helps. The first set of conditions and actions you identify may be unsatisfactory. Use it as a stepping stone and gradually improve on it until you are satisfied with a decision table.

EXERCISES

1. Develop a decision table and additional test cases for the right triangle addition to the triangle problem (see Chapter 2 exercises). Note that there can be isosceles right triangles, but not with integer sides.
2. Develop a decision table for the “second try” at the NextDate function. At the end of a 31-day month, the day is always reset to 1. For all non-December months, the month is incremented; and for December, the month is reset to January, and the year is incremented.
3. Develop a decision table for the YesterDate function (see Chapter 2 exercises).
4. Expand the commission problem to consider “violations” of the sales limits. Develop the corresponding decision tables and test cases for a “company friendly” version and a “salesperson friendly” version.

5. Discuss how well decision table testing deals with the multiple fault assumption.
6. Develop decision table test cases for the time change problem (Chapter 6, problem 5).
7. If you did exercise 8 in Chapter 2, exercise 5 in Chapter 5, and exercise 6 in Chapter 6, you are already familiar with the CRC Press website for downloads (<http://www.crcpress.com/product/isbn/9781466560680>). There you will find an Excel spreadsheet named specBasedTesting.xls. (It is an extended version of Naive.xls, and it contains the same inserted faults.) Different sheets contain decision table-based test cases for the triangle, NextDate, and commission problems, respectively. Run these sets of test cases and compare the results with your naive testing from Chapter 2, your boundary value testing from Chapter 5, and your equivalence class testing from Chapter 6.
8. The retirement pension salary of a Michigan public school teacher is a percentage of the average of their last 3 years of teaching. Normally, the number of years of teaching service is the percentage multiplier. To encourage senior teachers to retire early, the Michigan legislature enacted the following incentive in May of 2010:

Teachers must apply for the incentive before June 11, 2010. Teachers who are currently eligible to retire (age ≥ 63 years) shall have a multiplier of 1.6% on their salary up to, and including, \$90,000, and 1.5% on compensation in excess of \$90,000. Teachers who meet the 80 total years of age plus years of teaching shall have a multiplier of 1.55% on their salary up to, and including, \$90,000 and 1.5% on compensation in excess of \$90,000.

Make a decision table to describe the retirement pension policy; be sure to consider the retirement eligibility criteria carefully. What are the compensation multipliers for a person who is currently 64 with 20 years of teaching whose salary is \$95,000?

References

- Elmendorf, W.R., *Cause-Effect Graphs in Functional Testing*, IBM System Development Division, Poughkeepsie, NY, TR-00.2487, 1973.
- Mosley, D.J., *The Handbook of MIS Application Software Testing*, Yourdon Press, Prentice Hall, Englewood Cliffs, NJ, 1993.
- Myers, G.J., *The Art of Software Testing*, Wiley Interscience, New York, 1979.

Chapter 8

Path Testing

The distinguishing characteristic of code-based testing methods is that, as the name implies, they are all based on the source code of the program tested, and not on the specification. Because of this absolute basis, code-based testing methods are very amenable to rigorous definitions, mathematical analysis, and useful measurement. In this chapter, we examine the two most common forms of path testing. The technology behind these has been available since the mid-1970s, and the originators of these methods now have companies that market very successful tools that implement the techniques. Both techniques start with the program graph; we repeat the improved definition from Chapter 4 here.

8.1 Program Graphs

Definition

Given a program written in an imperative programming language, its *program graph* is a directed graph in which nodes are statement fragments, and edges represent flow of control. (A complete statement is a “default” statement fragment.)

If i and j are nodes in the program graph, an edge exists from node i to node j if and only if the statement fragment corresponding to node j can be executed immediately after the statement fragment corresponding to node i .

8.1.1 Style Choices for Program Graphs

Deriving a program graph from a given program is an easy process. It is illustrated here with four of the basic structured programming constructs (Figure 8.1), and also with our pseudocode implementation of the triangle program from Chapter 2. Line numbers refer to statements and statement fragments. An element of judgment can be used here: sometimes it is convenient to keep

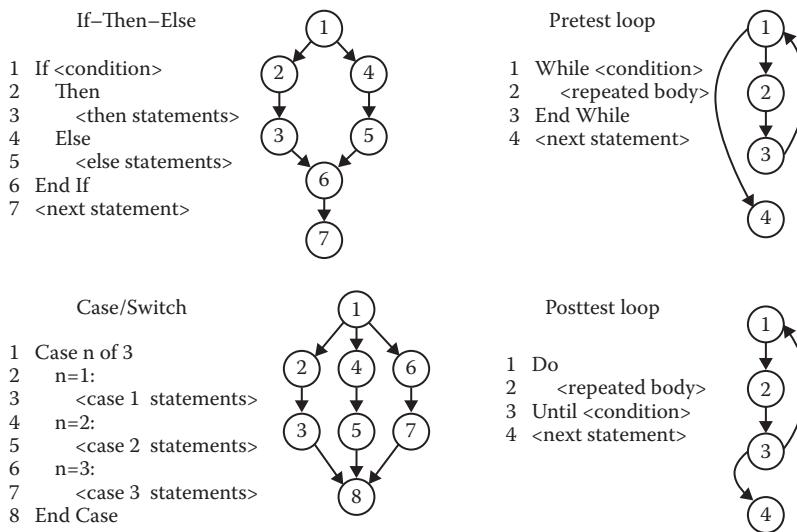


Figure 8.1 Program graphs of four structured programming constructs.

a fragment as a separate node; other times it seems better to include this with another portion of a statement. For example, in Figure 8.2, line 14 could be split into two lines:

```

14      Then If (a = b) AND (b = c)
14a          Then
14b              If (a = b) AND (b = c)

```

This latitude collapses onto a unique DD-path graph, so the differences introduced by differing judgments are moot. (A mathematician would make the point that, for a given program, several distinct program graphs might be used, all of which reduce to a unique DD-path graph.) We also need to decide whether to associate nodes with nonexecutable statements such as variable and type declarations; here we do not. A program graph of the second version of the triangle problem (see Chapter 2) is given in Figure 8.2.

Nodes 4 through 8 are a sequence, nodes 9 through 12 are an if-then-else construct, and nodes 13 through 22 are nested if-then-else constructs. Nodes 4 and 23 are the program source and sink nodes, corresponding to the single entry, single-exit criteria. No loops exist, so this is a directed acyclic graph. The importance of the program graph is that program executions correspond to paths from the source to the sink nodes. Because test cases force the execution of some such program path, we now have a very explicit description of the relationship between a test case and the part of the program it exercises. We also have an elegant, theoretically respectable way to deal with the potentially large number of execution paths in a program.

There are detractors of path-based testing. Figure 8.3 is a graph of a simple (but unstructured!) program; it is typical of the kind of example detractors use to show the (practical) impossibility of completely testing even simple programs. (This example first appeared in Schach [1993].) In this program, five paths lead from node B to node F in the interior of the loop. If the loop may have up to 18 repetitions, some 4.77 trillion distinct program execution paths exist. (Actually, it

```

1 Program triangle2
2 Dim a,b,c As Integer
3 Dim IsATriangle As Boolean
4 Output("Enter 3 integers which are sides of a triangle")
5 Input(a,b,c)
6 Output("Side A is", a)
7 Output("Side B is", b)
8 Output("Side C is", c)
9 If (a < b + c) AND (b < a + c) AND (c < a + b)
10 Then IsATriangle = True
11 Else IsATriangle = False
12 EndIf
13 If IsATriangle
14 Then If (a = b) AND (b = c)
15 Then Output ("Equilateral")
16 Else If (a≠b) AND (a≠c) AND (b≠c)
17 Then Output ("Scalene")
18 Else Output ("Isosceles")
19 EndIf
20 EndIf
21 Else Output("Not a Triangle")
22 EndIf
23 End triangle2

```

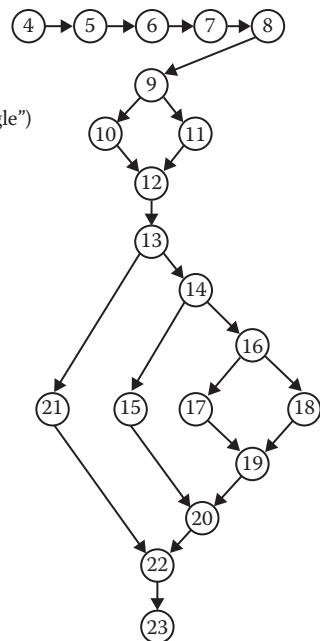


Figure 8.2 Program graph of triangle program.

is 4,768,371,582,030 paths.) The detractor's argument is a good example of the logical fallacy of extension—take a situation, extend it to an extreme, show that the extreme supports your point, and then apply it back to the original question. The detractors miss the point of code-based testing—later in this chapter, we will see how this enormous number can be reduced, with good reasons, to a more manageable size.

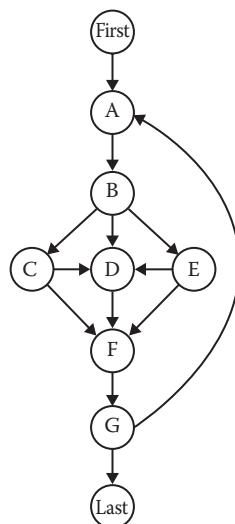


Figure 8.3 Trillions of paths.

8.2 DD-Paths

The best-known form of code-based testing is based on a construct known as a decision-to-decision path (DD-path) (Miller, 1977). The name refers to a sequence of statements that, in Miller's words, begins with the "outway" of a decision statement and ends with the "inway" of the next decision statement. No internal branches occur in such a sequence, so the corresponding code is like a row of dominoes lined up so that when the first falls, all the rest in the sequence fall. Miller's original definition works well for second-generation languages like FORTRAN II, because decision-making statements (such as arithmetic IFs and DO loops) use statement labels to refer to target statements. With modern languages (e.g., Pascal, Ada®, C, Visual Basic, Java), the notion of statement fragments resolves the difficulty of applying Miller's original definition. Otherwise, we end up with program graphs in which some statements are members of more than one DD-path. In the ISTQB literature, and also in Great Britain, the DD-path concept is known as a "linear code sequence and jump" and is abbreviated by the acronym LCSAJ. Same idea, longer name.

We will define DD-paths in terms of paths of nodes in a program graph. In graph theory, these paths are called chains, where a chain is a path in which the initial and terminal nodes are distinct, and every interior node has $\text{indeg} = 1$ and $\text{outdeg} = 1$. (See Chapter 4 for a formal definition.) Notice that the initial node is 2-connected to every other node in the chain, and no instances of 1- or 3-connected nodes occur, as shown in Figure 8.4. The length (number of edges) of the chain in Figure 8.4 is 6.

Definition

A *DD-path* is a sequence of nodes in a program graph such that

- Case 1: It consists of a single node with $\text{indeg} = 0$.
- Case 2: It consists of a single node with $\text{outdeg} = 0$.
- Case 3: It consists of a single node with $\text{indeg} \geq 2$ or $\text{outdeg} \geq 2$.
- Case 4: It consists of a single node with $\text{indeg} = 1$ and $\text{outdeg} = 1$.
- Case 5: It is a maximal chain of length ≥ 1 .

Cases 1 and 2 establish the unique source and sink nodes of the program graph of a structured program as initial and final DD-paths. Case 3 deals with complex nodes; it assures that no node is contained in more than one DD-path. Case 4 is needed for "short branches"; it also preserves the one-fragment, one DD-path principle. Case 5 is the "normal case," in which a DD-path is a single entry, single-exit sequence of nodes (a chain). The "maximal" part of the case 5 definition is used to determine the final node of a normal (nontrivial) chain.

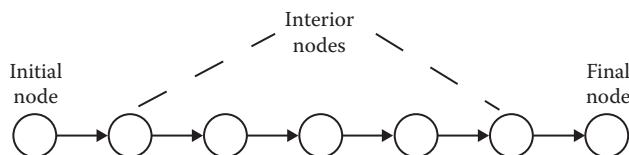


Figure 8.4 Chain of nodes in a directed graph.

Definition

Given a program written in an imperative language, its *DD-path graph* is the directed graph in which nodes are DD-paths of its program graph, and edges represent control flow between successor DD-paths.

This is a complex definition, so we will apply it to the program graph in Figure 8.2. Node 4 is a case 1 DD-path; we will call it “first.” Similarly, node 23 is a case 2 DD-path, and we will call it “last.” Nodes 5 through 8 are case 5 DD-paths. We know that node 8 is the last node in this DD-path because it is the last node that preserves the 2-connectedness property of the chain. If we go beyond node 8 to include node 9, we violate the $\text{indegree} = \text{outdegree} = 1$ criterion of a chain. If we stop at node 7, we violate the “maximal” criterion. Nodes 10, 11, 15, 17, 18, and 21 are case 4 DD-paths. Nodes 9, 12, 13, 14, 16, 19, 20, and 22 are case 3 DD-paths. Finally, node 23 is a case 2 DD-path. All this is summarized in Figure 8.5.

In effect, the DD-path graph is a form of condensation graph (see Chapter 4); in this condensation, 2-connected components are collapsed into individual nodes that correspond to case 5 DD-paths. The single-node DD-paths (corresponding to cases 1–4) are required to preserve the convention that a statement (or statement fragment) is in exactly one DD-path. Without this convention, we end up with rather clumsy DD-path graphs, in which some statement fragments are in several DD-paths.

This process should not intimidate testers—high-quality commercial tools are available, which generate the DD-path graph of a given program. The vendors make sure that their products work for a wide variety of programming languages. In practice, it is reasonable to manually create

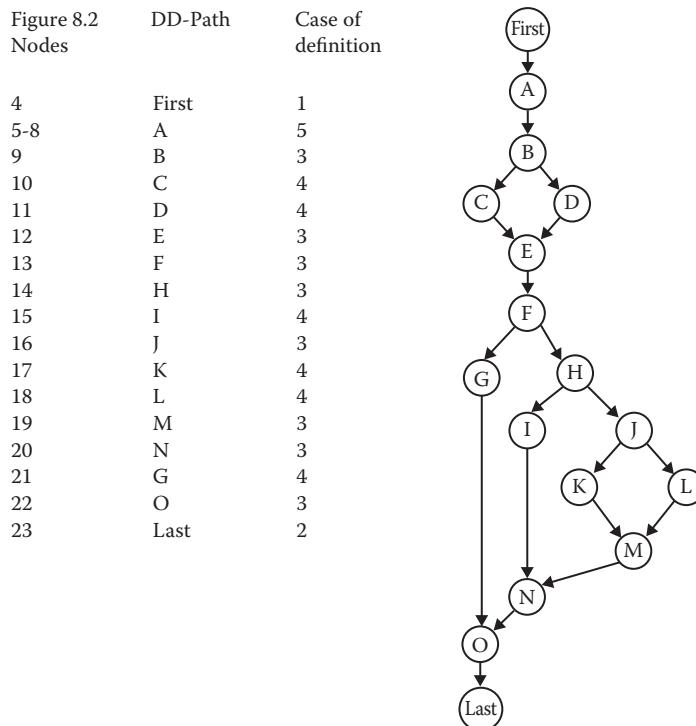


Figure 8.5 DD-path graph for triangle program.

DD-path graphs for programs up to about 100 source lines. Beyond that, most testers look for a tool.

Part of the confusion with this example is that the triangle problem is logic intensive and computationally sparse. This combination yields many short DD-paths. If the THEN and ELSE clauses contained blocks of computational statements, we would have longer chains, as we will see in the commission problem.

8.3 Test Coverage Metrics

The *raison d'être* of DD-paths is that they enable very precise descriptions of test coverage. Recall (from Chapters 5 through 7) that one of the fundamental limitations of specification-based testing is that it is impossible to know either the extent of redundancy or the possibility of gaps corresponding to the way a set of functional test cases exercises a program. Back in Chapter 1, we had a Venn diagram showing relationships among specified, programmed, and tested behaviors. Test coverage metrics are a device to measure the extent to which a set of test cases covers (or exercises) a program.

8.3.1 Program Graph-Based Coverage Metrics

Given a program graph, we can define the following set of test coverage metrics. We will use them to relate to other published sets of coverage metrics.

Definition

Given a set of test cases for a program, they constitute *node coverage* if, when executed on the program, every node in the program graph is traversed. Denote this level of coverage as G_{node} , where the G stands for program graph.

Since nodes correspond to statement fragments, this guarantees that every statement fragment is executed by some test case. If we are careful about defining statement fragment nodes, this also guarantees that statement fragments that are outcomes of a decision-making statement are executed.

Definition

Given a set of test cases for a program, they constitute *edge coverage* if, when executed on the program, every edge in the program graph is traversed. Denote this level of coverage as G_{edge} .

The difference between G_{node} and G_{edge} is that, in the latter, we are assured that all outcomes of a decision-making statement are executed. In our triangle problem (see Figure 8.2), nodes 9, 10, 11, and 12 are a complete if–then–else statement. If we required nodes to correspond to full statements, we could execute just one of the decision alternatives and satisfy the statement coverage criterion. Because we allow statement fragments, it is natural to divide such a statement into separate nodes (the condition test, the true outcome, and the false outcome). Doing so results in predicate outcome coverage. Whether or not our convention is followed, these coverage metrics

require that we find a set of test cases such that, when executed, every node of the program graph is traversed at least once.

Definition

Given a set of test cases for a program, they constitute *chain coverage* if, when executed on the program, every chain of length greater than or equal to 2 in the program graph is traversed. Denote this level of coverage as G_{chain} .

The G_{chain} coverage is the same as node coverage in the DD-path graph that corresponds to the given program graph. Since DD-paths are important in E.F. Miller's original formulation of test covers (defined in Section 8.3.2), we now have a clear connection between purely program graph constructs and Miller's test covers.

Definition

Given a set of test cases for a program, they constitute *path coverage* if, when executed on the program, every path from the source node to the sink node in the program graph is traversed. Denote this level of coverage as G_{path} .

This coverage is open to severe limitations when there are loops in a program (as in Figure 8.3). E.F. Miller partially anticipated this when he postulated the C_2 metric for loop coverage. Referring back to Chapter 4, observe that every loop in a program graph represents a set of strongly (3-connected) nodes. To deal with the size implications of loops, we simply exercise every loop, and then form the condensation graph of the original program graph, which must be a directed acyclic graph.

8.3.2 E.F. Miller's Coverage Metrics

Several widely accepted test coverage metrics are used; most of those in Table 8.1 are due to the early work of Miller (1977). Having an organized view of the extent to which a program is tested makes it possible to sensibly manage the testing process. Most quality organizations now expect the C_1 metric (DD-path coverage) as the minimum acceptable level of test coverage.

These coverage metrics form a lattice (see Chapter 9 for a lattice of data flow coverage metrics) in which some are equivalent and some are implied by others. The importance of the lattice is that there are always fault types that can be revealed at one level and can escape detection by inferior levels of testing. Miller (1991) observes that when DD-path coverage is attained by a set of test cases, roughly 85% of all faults are revealed. The test coverage metrics in Table 8.1 tell us what to test but not how to test it. In this section, we take a closer look at techniques that exercise source code. We must keep an important distinction in mind: Miller's test coverage metrics are based on program graphs in which nodes are full statements, whereas our formulation allows statement fragments (which can be entire statements) to be nodes.

8.3.2.1 Statement Testing

Because our formulation of program graphs allows statement fragments to be individual nodes, Miller's C_0 metric is subsumed by our G_{node} metric.

Table 8.1 Miller's Test Coverage Metrics

Metric	Description of Coverage
C_0	Every statement
C_1	Every DD-path
C_{1p}	Every predicate to each outcome
C_2	C_1 coverage + loop coverage
C_d	C_1 coverage + every dependent pair of DD-paths
C_{MCC}	Multiple condition coverage
C_{ik}	Every program path that contains up to k repetitions of a loop (usually $k = 2$)
C_{stat}	"Statistically significant" fraction of paths
C_∞	All possible execution paths

Statement coverage is generally viewed as the bare minimum. If some statements have not been executed by the set of test cases, there is clearly a severe gap in the test coverage. Although less adequate than DD-path coverage, the statement coverage metric (C_0) is still widely accepted: it is mandated by ANSI (American National Standards Institute) Standard 187B and has been used successfully throughout IBM since the mid-1970s.

8.3.2.2 DD-Path Testing

When every DD-path is traversed (the C_1 metric), we know that each predicate outcome has been executed; this amounts to traversing every edge in the DD-path graph (or program graph). Therefore, the C_1 metric is exactly our G_{chain} metric.

For if–then and if–then–else statements, this means that both the true and the false branches are covered (C_{1p} coverage). For CASE statements, each clause is covered. Beyond this, it is useful to ask how we might test a DD-path. Longer DD-paths generally represent complex computations, which we can rightly consider as individual functions. For such DD-paths, it may be appropriate to apply a number of functional tests, especially those for boundary and special values.

8.3.2.3 Simple Loop Coverage

The C_2 metric requires DD-path coverage (the C_1 metric) plus loop testing.

The simple view of loop testing is that every loop involves a decision, and we need to test both outcomes of the decision: one is to traverse the loop, and the other is to exit (or not enter) the loop. This is carefully proved in Huang (1979). Notice that this is equivalent to the G_{edge} test coverage.

8.3.2.4 Predicate Outcome Testing

This level of testing requires that every outcome of a decision (predicate) must be exercised. Because our formulation of program graphs allows statement fragments to be individual nodes,

Miller's C_{lp} metric is subsumed by our G_{edge} metric. Neither E.F. Miller's test covers nor the graph-based covers deal with decisions that are made on compound conditions. They are the subjects of Section 8.3.3.

8.3.2.5 Dependent Pairs of DD-Paths

Identification of dependencies must be made at the code level. This cannot be done just by considering program graphs. The C_d metric foreshadows the topic of Chapter 9—data flow testing. The most common dependency among pairs of DD-paths is the define/reference relationship, in which a variable is defined (receives a value) in one DD-path and is referenced in another DD-path. The importance of these dependencies is that they are closely related to the problem of infeasible paths. We have good examples of dependent pairs of DD-paths: in Figure 8.5, C and H are such a pair, as are DD-paths D and H. The variable IsATriangle is set to TRUE at node C, and FALSE at node D. Node H is the branch taken when IsATriangle is TRUE when the condition at node F. Any path containing nodes D and H is infeasible. Simple DD-path coverage might not exercise these dependencies; thus, a deeper class of faults would not be revealed.

8.3.2.6 Complex Loop Coverage

Miller's C_{ik} metric extends the loop coverage metric to include full paths from source to sink nodes that contain loops.

The condensation graphs we studied in Chapter 4 provide us with an elegant resolution to the problems of testing loops. Loop testing has been studied extensively, and with good reason—loops are a highly fault-prone portion of source code. To start, an amusing taxonomy of loops occurs (Beizer, 1984): concatenated, nested, and horrible, shown in Figure 8.6.

Concatenated loops are simply a sequence of disjoint loops, while nested loops are such that one is contained inside another. Knotted (Beizer calls them “horrible”) loops cannot occur when the structured programming precepts are followed, but they can occur in languages like Java with try/catch. When it is possible to branch into (or out from) the middle of a loop, and these branches

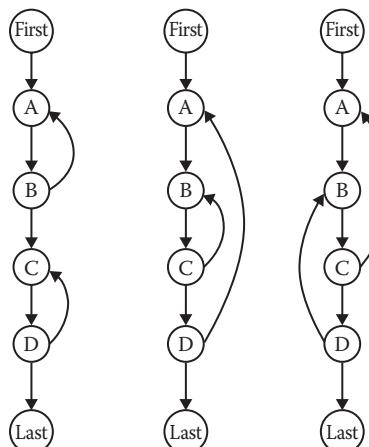


Figure 8.6 Concatenated, nested, and knotted loops.

are internal to other loops, the result is Beizer's knotted loop. We can also take a modified boundary value approach, where the loop index is given its minimum, nominal, and maximum values (see Chapter 5). We can push this further to full boundary value testing and even robustness testing. If the body of a simple loop is a DD-path that performs a complex calculation, this should also be tested, as discussed previously. Once a loop has been tested, the tester condenses it into a single node. If loops are nested, this process is repeated starting with the innermost loop and working outward. This results in the same multiplicity of test cases we found with boundary value analysis, which makes sense, because each loop index variable acts like an input variable. If loops are knotted, it will be necessary to carefully analyze them in terms of the data flow methods discussed in Chapter 9. As a preview, consider the infinite loop that could occur if one loop tampers with the value of the other loop's index.

8.3.2.7 Multiple Condition Coverage

Miller's C_{MCC} metric addresses the question of testing decisions made by compound conditions. Look closely at the compound conditions in DD-paths B and H. Instead of simply traversing such predicates to their true and false outcomes, we should investigate the different ways that each outcome can occur. One possibility is to make a decision table; a compound condition of three simple conditions will have eight rules (see Table 8.2), yielding eight test cases. Another possibility is to reprogram compound predicates into nested simple if–then–else logic, which will result in more DD-paths to cover. We see an interesting tradeoff: statement complexity versus path complexity. Multiple condition coverage assures that this complexity is not swept under the DD-path coverage rug. This metric has been refined to Modified Condition Decision Coverage (MCDC), defined in Section 8.3.3.

8.3.2.8 "Statistically Significant" Coverage

The C_{stat} metric is awkward—what constitutes a statistically significant set of full program paths? Maybe this refers to a comfort level on the part of the customer/user.

8.3.2.9 All Possible Paths Coverage

The subscript in Miller's C_∞ metric says it all—this can be enormous for programs with loops, *a la* Figure 8.3. This can make sense for programs without loops, and also for programs for which loop testing reduces the program graph to its condensation graph.

8.3.3 A Closer Look at Compound Conditions

There is an excellent reference (Chilenski, 2001) that is 214 pages long and is available on the Web. The definitions in this subsection are derived from this reference. They will be related to the definitions in Sections 8.3.1 and 8.3.2.

8.3.3.1 Boolean Expression (per Chilenski)

"A *Boolean expression* evaluates to one of two possible (Boolean) outcomes traditionally known as False and True."

A Boolean expression may be a simple Boolean variable, or a compound expression containing one or more Boolean operators. Chilenski clarifies Boolean operators into four categories:

<i>Operator Type</i>	<i>Boolean Operators</i>
Unary (single operand)	NOT(\sim),
Binary (two operands)	AND(\wedge), OR(\vee), XOR(\oplus)
Short circuit operators	AND (AND-THEN), OR (OR-ELSE)
Relational operators	=, \neq , $<$, \leq , $>$, \geq

In mathematical logic, Boolean expressions are known as *logical expressions*, where a logical expression can be

1. A simple proposition that contains no logical connective
2. A compound proposition that contains at least one logical connective

Synonyms: *predicate, proposition, condition.*

In programming languages, Chilenski's Boolean expressions appear as conditions in decision making statements: If–Then, If–Then–Else, If–ElseIf, Case/Switch, For, While, and Until loops. This subsection is concerned with the testing needed for compound conditions. Compound conditions are shown as single nodes in a program graph; hence, the complexity they introduce is obscured.

8.3.3.2 Condition (per Chilenski)

“A *condition* is an operand of a Boolean operator (Boolean functions, objects and operators).

Generally this refers to the lowest level conditions (i.e., those operands that are not Boolean operators themselves), which are normally the leaves of an expression tree. Note that a condition is a Boolean (sub)expression.”

In mathematical logic, Chilenski's conditions are known as simple, or atomic, propositions. Propositions can be simple or compound, where a compound proposition contains at least one logical connective. Propositions are also called predicates, the term that E.F. Miller uses.

8.3.3.3 Coupled Conditions (per Chilenski)

Two (or more) conditions are *coupled* if changing one also changes the other(s).

When conditions are coupled, it may not be possible to vary individual conditions, because the coupled condition(s) might also change. Chelinski notes that conditions can be strongly or weakly coupled. In a strongly coupled pair, changing one condition always changes the other. In a weakly coupled triplet, changing one condition may change one other coupled condition, but not the third one. Chelinski offers these examples:

In $((x = 0) \text{ AND } A) \text{ OR } ((x \neq 0) \text{ AND } B)$, the conditions $(x = 0)$ and $(x \neq 0)$ are strongly coupled.

In $((x = 1) \text{ OR } (x = 2) \text{ OR } (x = 3))$, the three conditions are weakly coupled.

8.3.3.4 Masking Conditions (per Chilenski)

“The process *masking conditions* involves of setting the one operand of an operator to a value such that changing the other operand of that operator does not change the value of the operator.

Referring to Chapter 3.4.3, masking uses the Domination Laws. For an AND operator, masking of one operand can be achieved by holding the other operand False.

$(X \text{ AND } \text{False} = \text{False} \text{ AND } X = \text{False} \text{ no matter what the value of } X \text{ is.})$

For an OR operator, masking of one operand can be achieved by holding the other operand True.

$(X \text{ OR } \text{True} = \text{True} \text{ OR } X = \text{True} \text{ no matter what the value of } X \text{ is.})$

8.3.3.5 Modified Condition Decision Coverage

MCDC is required for “Level A” software by testing standard DO-178B. MCDC has three variations: Masking MCDC, Unique-Cause MCDC, and Unique-Cause + Masking MCDC. These are explained in exhaustive detail in Chilenski (2001), which concludes that Masking MCDC, while demonstrably the weakest form of the three, is recommended for compliance with DO-178B. The definitions below are quoted from Chilenski.

Definition

MCDC requires

1. Every statement must be executed at least once.
2. Every program entry point and exit point must be invoked at least once.
3. All possible outcomes of every control statement are taken at least once.
4. Every nonconstant Boolean expression has been evaluated to both true and false outcomes.
5. Every nonconstant condition in a Boolean expression has been evaluated to both true and false outcomes.
6. Every nonconstant condition in a Boolean expression has been shown to independently affect the outcomes (of the expression).

The basic definition of MCDC needs some explanation. Control statements are those that make decisions, such as If statements, Case/Switch statements, and looping statements. In a program graph, control statements have an outdegree greater than 1. Constant Boolean expressions are those that always evaluate to the same end value. For example, the Boolean expression $(p \vee \sim p)$ always evaluates to True, as does the condition $(a = a)$. Similarly, $(p \wedge \sim p)$ and $(a \neq a)$ are constant expressions (that evaluate to False). In terms of program graphs, MCDC requirements 1 and 2 translate to node coverage, and MCDC requirements 3 and 4 translate to edge coverage. MCDC requirements 5 and 6 get to the complex part of MCDC testing. In the following, the three variations discussed by Chilenski are intended to clarify the meaning of point 6 of the general definition, namely, the exact meaning of “independence.”

Definition (per Chilenski)

"Unique-Cause MCDC [requires] a unique cause (toggle a single condition and change the expression result) for all possible (uncoupled) conditions."

Definition (per Chilenski)

"Unique-Cause + Masking MCDC [requires] a unique cause (toggle a single condition and change the expression result) for all possible (uncoupled) conditions. In the case of strongly coupled conditions, masking [is allowed] for that condition only, i.e., all other (uncoupled) conditions will remain fixed."

Definition (per Chilenski)

"Masking MCDC allows masking for all conditions, coupled and uncoupled (toggle a single condition and change the expression result) for all possible (uncoupled) conditions. In the case of strongly coupled conditions, masking [is allowed] for that condition only (i.e., all other (uncoupled) conditions will remain fixed.)"

Chilenski comments: "In the case of strongly coupled conditions, no coverage set is possible as DO-178B provides no guidance on how such conditions should be covered."

8.3.4 Examples

The examples in this section are directed at the variations of testing code with compound conditions.

8.3.4.1 Condition with Two Simple Conditions

Consider the program fragment in Figure 8.7. It is deceptively simple, with a cyclomatic complexity of 2.

The decision table (see Chapter 7) for the condition (a AND (b OR c)) is in Table 8.2. Decision coverage is attained by exercising any pair of rules such that each action is executed at least once. Test cases corresponding to rules 3 and 4 provide decision coverage, as do rules 1 and 8. Condition coverage is attained by exercising a set of rules such that each condition is evaluated to both true and false. Test cases corresponding to rules 1 and 8 provide condition coverage, as do rules 4 and 5.

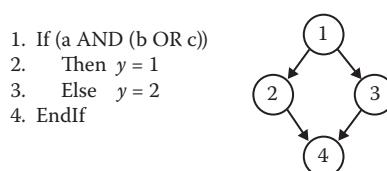


Figure 8.7 Compound condition and its program graph.

Table 8.2 Decision Table for Example Program Fragment

<i>Conditions</i>	<i>Rule 1</i>	<i>Rule 2</i>	<i>Rule 3</i>	<i>Rule 4</i>	<i>Rule 5</i>	<i>Rule 6</i>	<i>Rule 7</i>	<i>Rule 8</i>
a	T	T	T	T	F	F	F	F
b	T	T	F	F	T	T	F	F
c	T	F	T	F	T	F	T	F
a AND (b OR c)	True	True	True	False	False	False	False	False
Actions								
$y = 1$	x	x	x	—	—	—	—	—
$y = 2$	—	—	—	x	x	x	x	x

To attain MCDC, each condition must be evaluated to both true and false while holding the other conditions constant, and the change must be visible at the outcome. Rules 1 and 5 toggle condition a; rules 2 and 4 toggle condition b; and rules 3 and 4 toggle condition c.

In the Chelinski (2001) paper (p. 9), it happens that the Boolean expression used is

$$(a \text{ AND } (b \text{ OR } c))$$

In its expanded form, $(a \text{ AND } b) \text{ OR } (a \text{ AND } c)$, the Boolean variable a cannot be subjected to unique cause MCDC testing because it appears in both AND expressions.

Given all the complexities here (see Chelinski [2001] for much, much more) the best practical solution is to just make a decision table of the actual code, and look for impossible rules. Any dependencies will typically generate an impossible rule.

8.3.4.2 Compound Condition from NextDate

In our continuing NextDate problem, suppose we have some code checking for valid inputs of the day, month, and year variables. A code fragment for this and its program graph are in Figure 8.8. Table 8.3 is a decision table for the NextDate code fragment. Since the day, month, and year variables are all independent, each can be either true or false. The cyclomatic complexity of the program graph in Figure 8.8 is 5.

Decision coverage is attained by exercising any pair of rules such that each action is executed at least once. Test cases corresponding to rule 1 and any one of rules 2–8 provide decision coverage.

Multiple condition coverage requires exercising a set of rules such that each condition is evaluated to both True and False. The eight test cases corresponding to all eight rules are necessary to provide decision coverage.

To attain MCDC, each condition must be evaluated to both true and false while holding the other conditions constant, and the change must be visible at the outcome. Rules 1 and 2 toggle condition yearOK; rules 1 and 3 toggle condition monthOK, and rules 1 and 5 toggle condition dayOK.

Since the three variables are truly independent, multiple condition coverage will be needed.

```

1 NextDate Fragment
2 Dim day, month, year As Integer
3 Dim dayOK, monthOK, yearOK As Boolean
4 Do
5   Input(day, month, year)
6   If 0 < day < 32
7     Then dayOK = True
8     Else dayOK = False
9   EndIf
10  If 0 < month < 13
11    Then monthOK = True
12    Else monthOK = False
13  EndIf
14  If 1811 < year < 2013
15    Then yearOK = True
16    Else yearOK = False
17  EndIf
18 Until (dayOK AND monthOK AND yearOK)
19 End Fragment

```

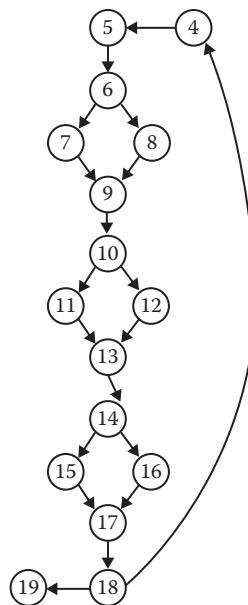


Figure 8.8 NextDate fragment and its program graph.

Table 8.3 Decision Table for NextDate Fragment

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
dayOK	T	T	T	T	F	F	F	F
monthOK	T	T	F	F	T	T	F	F
YearOK	T	F	T	F	T	F	T	F
The Until condition	True	False						

Actions								
Leave the loop	x	—	—	—	—	—	—	—
Repeat the loop	—	x	x	x	x	x	x	x

8.3.4.3 Compound Condition from the Triangle Program

This example is included to show important differences between it and the first two examples. The code fragment in Figure 8.9 is the part of the triangle program that checks to see if the values of sides a, b, and c constitute a triangle. The test incorporates the definition that each side must be strictly less than the sum of the other two sides. Notice that the program graphs in Figures 8.7 and 8.9 are identical. The NextDate fragment and the triangle program fragment are both functions of three variables. The second difference is that a, b, and c in the triangle program are dependent, whereas dayOK, monthOK, and yearOK in the NextDate fragment are truly independent variables.

1. If $(a < b + c) \text{ AND } (a < b + c) \text{ AND } (a < b + c)$
2. Then IsA Triangle = True
3. Else IsA Triangle = False
4. EndIf

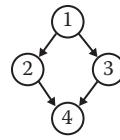


Figure 8.9 Triangle program fragment and its program graph.

The dependence among a , b , and c is the cause of the four impossible rules in the decision table for the fragment in Table 8.4; this is proved next.

Fact: It is numerically impossible to have two of the conditions false.

Proof (by contradiction): Assume any pair of conditions can both be true. Arbitrarily choosing the first two conditions that could both be true, we can write the two inequalities

$$a \geq (b + c)$$

$$b \geq (a + c)$$

Adding them together, we have

$$(a + b) \geq (b + c) + (a + c)$$

and rearranging the right side, we have

$$(a + b) \geq (a + b) + 2c$$

But a , b , and c are all > 0 , so we have a contradiction. QED.

Decision coverage is attained by exercising any pair of rules such that each action is executed at least once. Test cases corresponding to rules 1 and 2 provide decision coverage, as do rules 1 and 3, and rules 1 and 5. Rules 4, 6, 7, and 8 cannot be used owing to their numerical impossibility.

Condition coverage is attained by exercising a set of rules such that each condition is evaluated to both true and false. Test cases corresponding to rules 1 and 2 toggle the $(c < a + b)$ condition, rules 1 and 3 toggle the $(b < a + c)$ condition, and 1 and 5 toggle the $(a < b + c)$ condition.

MCDC is complicated by the numerical (and hence logical) impossibilities among the three conditions. The three pairs (rules 1 and 2, rules 1 and 3, and rules 1 and 5) constitute MCDC.

Table 8.4 Decision Table for Triangle Program Fragment

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
$(a < b + c)$	T	T	T	T	F	F	F	F
$(b < a + c)$	T	T	F	F	T	T	F	F
$(c < a + b)$	T	F	T	F	T	F	T	F
IsATriangle = True	x	—	—	—	—	—	—	—
IsATriangle = False	—	x	x	—	x	—	—	—
Impossible	—	—	—	x		x	x	x

In complex situations such as these examples, falling back on decision tables is an answer that will always work. Rewriting the compound condition with nested If logic, we will have (preserving the original statement numbers)

```

1.1      If (a < b + c)
1.2          Then If (b < a + c)
1.3              Then If (c < a + b)
2                  Then IsATriangle = True
3.1                  Else IsATriangle = False
3.2              End If
3.3          Else IsATriangle = False
3.4      End If
3.5      Else IsATriangle = False
4      EndIf

```

This code fragment avoids the numerically impossible combinations of a, b, and c. There are four distinct paths through its program graph, and these correspond to rules 1, 2, 3, and 5 in the decision table.

8.3.5 Test Coverage Analyzers

Coverage analyzers are a class of test tools that offer automated support for this approach to testing management. With a coverage analyzer, the tester runs a set of test cases on a program that has been “instrumented” by the coverage analyzer. The analyzer then uses information produced by the instrumentation code to generate a coverage report. In the common case of DD-path coverage, for example, the instrumentation identifies and labels all DD-paths in an original program. When the instrumented program is executed with test cases, the analyzer tabulates the DD-paths traversed by each test case. In this way, the tester can experiment with different sets of test cases to determine the coverage of each set. Mr. Tilo Linz maintains a website with excellent test tool information at www.testtoolreview.com.

8.4 Basis Path Testing

The mathematical notion of a “basis” has attractive possibilities for structural testing. Certain sets can have a basis; and when they do, the basis has very important properties with respect to the entire set. Mathematicians usually define a basis in terms of a structure called a “vector space,” which is a set of elements (called vectors) as well as operations that correspond to multiplication and addition defined for the vectors. If a half dozen other criteria apply, the structure is said to be a vector space, and all vector spaces have a basis (in fact they may have several bases). The basis of a vector space is a set of vectors that are independent of each other and “span” the entire vector space in the sense that any other vector in the space can be expressed in terms of the basis vectors. Thus, a set of basis vectors somehow represents “the essence” of the full vector space: everything else in the space can be expressed in terms of the basis, and if one basis element is deleted, this spanning property is lost. The potential application of this theory for testing is that, if we can view a program as a vector space, then the basis for such a space would be a very interesting set of elements to test. If the basis is okay, we could hope that everything that can be expressed in terms of the basis is also okay. In this section, we examine the early work of Thomas McCabe, who recognized this possibility in the mid-1970s.

8.4.1 McCabe's Basis Path Method

Figure 8.10 is taken from McCabe (1982). It is a directed graph that we might take to be the program graph (or the DD-path graph) of some program. For the convenience of readers who have encountered this example elsewhere (McCabe, 1987; Perry, 1987), the original notation for nodes and edges is repeated here. (Notice that this is not a graph derived from a structured program: nodes B and C are a loop with two exits, and the edge from B to E is a branch into the if–then statement in nodes D, E, and F.) The program does have a single entry (A) and a single exit (G). McCabe based his view of testing on a major result from graph theory, which states that the cyclomatic number (see Chapter 4) of a strongly connected graph is the number of linearly independent circuits in the graph. (A circuit is similar to a chain: no internal loops or decisions occur, but the initial node is the terminal node. A circuit is a set of 3-connected nodes.)

We can always create a strongly connected graph by adding an edge from the (every) sink node to the (every) source node. (Notice that, if the single-entry, single-exit precept is violated, we greatly increase the cyclomatic number because we need to add edges from each sink node to each source node.) The right side of Figure 8.10 shows the result of doing this; it also contains edge labels that are used in the discussion that follows.

Some confusion exists in the literature about the correct formula for cyclomatic complexity. Some sources give the formula as $V(G) = e - n + p$, while others use the formula $V(G) = e - n + 2p$; everyone agrees that e is the number of edges, n is the number of nodes, and p is the number of connected regions. The confusion apparently comes from the transformation of an arbitrary directed graph (such as the one in Figure 8.10, left side) to a strongly connected, directed graph obtained by adding one edge from the sink to the source node (as in Figure 8.10, right side). Adding an edge clearly affects value computed by the formula, but it should not affect the number of circuits. Counting or not counting the added edge accounts for the change to the coefficient of p , the number of connected regions. Since p is usually 1, adding the extra edge means we move from $2p$ to p . Here is a way to resolve the apparent inconsistency. The number of linearly independent paths from the source node to the sink node of the graph on the left side of Figure 8.10 is

$$\begin{aligned} V(G) &= e - n + 2p \\ &= 10 - 7 + 2(1) = 5 \end{aligned}$$

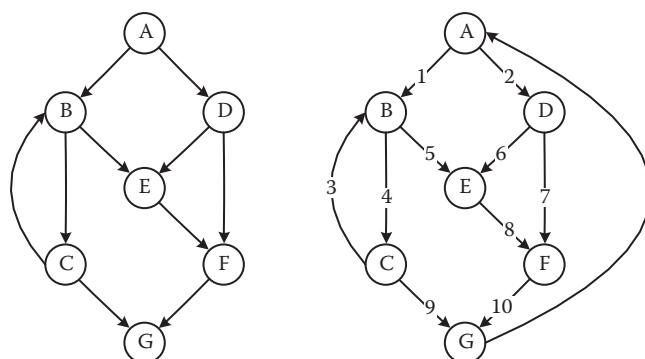


Figure 8.10 McCabe's control graph and derived strongly connected graph.

The number of linearly independent circuits of the graph on the right side of the graph in Figure 8.10 is

$$\begin{aligned}V(G) &= e - n + p \\&= 11 - 7 + 1 = 5\end{aligned}$$

The cyclomatic complexity of the strongly connected graph in Figure 8.10 is 5; thus, there are five linearly independent circuits. If we now delete the added edge from node G to node A, these five circuits become five linearly independent paths from node A to node G. In small graphs, we can visually identify independent paths. Here, we identify paths as sequences of nodes:

- p1: A, B, C, G
- p2: A, B, C, B, C, G
- p3: A, B, E, F, G
- p4: A, D, E, F, G
- p5: A, D, F, G

Table 8.5 shows the edges traversed by each path, and also the number of times an edge is traversed. We can force this to begin to look like a vector space by defining notions of addition and scalar multiplication: path addition is simply one path followed by another path, and multiplication corresponds to repetitions of a path. With this formulation, McCabe arrives at a vector space of program paths. His illustration of the basis part of this framework is that the path A, B, C, B, E, F, G is the basis sum $p_2 + p_3 - p_1$, and the path A, B, C, B, C, B, C, G is the linear combination $2p_2 - p_1$. It is easier to see this addition with an incidence matrix (see Chapter 4) in which rows correspond to paths, and columns correspond to edges, as in Table 8.5. The entries in this table are obtained by following a path and noting which edges are traversed. Path p1, for example, traverses edges 1, 4, and 9, while path p2 traverses the following edge sequence: 1, 4, 3, 4, 9. Because edge 4 is traversed twice by path p2, that is the entry for the edge 4 column.

We can check the independence of paths p1 – p5 by examining the first five rows of this incidence matrix. The bold entries show edges that appear in exactly one path, so paths p2 – p5 must

Table 8.5 Path/Edge Traversal

Path/Edges Traversed	1	2	3	4	5	6	7	8	9	10
p1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
p2: A, B, C, B, C, G	1	0	1	2	0	0	0	0	1	0
p3: A, B, E, F, G	1	0	0	0	1	0	0	1	0	1
p4: A, D, E, F, G	0	1	0	0	0	1	0	1	0	1
p5: A, D, F, G	0	1	0	0	0	0	1	0	0	1
ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

Table 8.6 Basis Paths in Figure 8.5

Original	p1: A–B–C–E–F–H–J–K–M–N–O–Last	Scalene
Flip p1 at B	p2: A–B–D–E–F–H–J–K–M–N–O–Last	Infeasible
Flip p1 at F	p3: A–B–C–E–F–G–O–Last	Infeasible
Flip p1 at H	p4: A–B–C–E–F–H–I–N–O–Last	Equilateral
Flip p1 at J	p5: A–B–C–E–F–H–J–L–M–N–O–Last	Isosceles

be independent. Path p1 is independent of all of these, because any attempt to express p1 in terms of the others introduces unwanted edges. None can be deleted, and these five paths span the set of all paths from node A to node G. At this point, you might check the linear combinations of the two example paths. (The addition and multiplication are performed on the column entries.)

McCabe next develops an algorithmic procedure (called the baseline method) to determine a set of basis paths. The method begins with the selection of a baseline path, which should correspond to some “normal case” program execution. This can be somewhat arbitrary; McCabe advises choosing a path with as many decision nodes as possible. Next, the baseline path is retraced, and in turn each decision is “flipped”; that is, when a node of $\text{outdegree} \geq 2$ is reached, a different edge must be taken. Here we follow McCabe’s example, in which he first postulates the path through nodes A, B, C, B, E, F, G as the baseline. (This was expressed in terms of paths p1 – p5 earlier.) The first decision node ($\text{outdegree} \geq 2$) in this path is node A; thus, for the next basis path, we traverse edge 2 instead of edge 1. We get the path A, D, E, F, G, where we retrace nodes E, F, G in path 1 to be as minimally different as possible. For the next path, we can follow the second path, and take the other decision outcome of node D, which gives us the path A, D, F, G. Now, only decision nodes B and C have not been flipped; doing so yields the last two basis paths, A, B, E, F, G and A, B, C, G. Notice that this set of basis paths is distinct from the one in Table 8.6: this is not problematic because a unique basis is not required.

8.4.2 Observations on McCabe’s Basis Path Method

If you had trouble following some of the discussion on basis paths and sums and products of these, you may have felt a haunting skepticism—something along the lines of, “Here’s another academic oversimplification of a real-world problem.” Rightly so, because two major soft spots occur in the McCabe view: one is that testing the set of basis paths is sufficient (it is not), and the other has to do with the yoga-like contortions we went through to make program paths look like a vector space. McCabe’s example that the path A, B, C, B, C, B, C, G is the linear combination $2p2 - p1$ is very unsatisfactory. What does the $2p2$ part mean? Execute path p2 twice? (Yes, according to the math.) Even worse, what does the $-p1$ part mean? Execute path p1 backward? Undo the most recent execution of p1? Do not do p1 next time? Mathematical sophistries like this are a real turnoff to practitioners looking for solutions to their very real problems. To get a better understanding of these problems, we will go back to the triangle program example.

Start with the DD-path graph of the triangle program in Figure 8.5. We begin with a baseline path that corresponds to a scalene triangle, for example, with sides 3, 4, 5. This test case

will traverse the path p1 (see Table 8.5). Now, if we flip the decision at node B, we get path p2. Continuing the procedure, we flip the decision at node F, which yields the path p3. Now, we continue to flip decision nodes in the baseline path p1; the next node with outdegree = 2 is node H. When we flip node H, we get the path p4. Next, we flip node J to get p5. We know we are done because there are only five basis paths; they are shown in Table 8.5.

Time for a reality check: if you follow paths p2 and p3, you find that they are both infeasible. Path p2 is infeasible because passing through node D means the sides are not a triangle; so the outcome of the decision at node F must be node G. Similarly, in p3, passing through node C means the sides do form a triangle; so node G cannot be traversed. Paths p4 and p5 are both feasible and correspond to equilateral and isosceles triangles, respectively. Notice that we do not have a basis path for the NotATriangle case.

Recall that dependencies in the input data domain caused difficulties for boundary value testing and that we resolved these by going to decision table-based specification-based testing, where we addressed data dependencies in the decision table. Here, we are dealing with code-level dependencies, which are absolutely incompatible with the latent assumption that basis paths are independent. McCabe's procedure successfully identifies basis paths that are topologically independent; however, when these contradict semantic dependencies, topologically possible paths are seen to be logically infeasible. One solution to this problem is to always require that flipping a decision results in a semantically feasible path. Another is to reason about logical dependencies. If we think about this problem, we can identify two rules:

If node C is traversed, then we must traverse node H.

If node D is traversed, then we must traverse node G.

Taken together, these rules, in conjunction with McCabe's baseline method, will yield the following feasible basis path set. Notice that logical dependencies reduce the size of a basis set when basis paths must be feasible.

p1: A–B–C–E–F–H–J–K–M–N–O–Last	Scalene
p6: A–B–D–E–F–G–O–Last	Not a triangle
p4: A–B–C–E–F–H–I–N–O–Last	Equilateral
p5: A–B–C–E–F–H–J–L–M–N–O–Last	Isosceles

The triangle problem is atypical in that no loops occur. The program has only eight topologically possible paths; and of these, only the four basis paths listed above are feasible. Thus, for this special case, we arrive at the same test cases as we did with special value testing and output range testing.

For a more positive observation, basis path coverage guarantees DD-path coverage: the process of flipping decisions guarantees that every decision outcome is traversed, which is the same as DD-path coverage. We see this by example from the incidence matrix description of basis paths and in our triangle program feasible basis paths. We could push this a step further and observe that the set of DD-paths acts like a basis because any program path can be expressed as a linear combination of DD-paths.

8.4.3 Essential Complexity

Part of McCabe's work on cyclomatic complexity does more to improve programming than testing. In this section, we take a quick look at this elegant blend of graph theory, structured programming, and the implications these have for testing. This whole package centers on the notion of essential complexity (McCabe, 1982), which is only the cyclomatic complexity of yet another form of condensation graph. Recall that condensation graphs are a way of simplifying an existing graph; thus far, our simplifications have been based on removing either strong components or DD-paths. Here, we condense around the structured programming constructs, which are repeated as Figure 8.11.

The basic idea is to look for the graph of one of the structured programming constructs, collapse it into a single node, and repeat until no more structured programming constructs can be found. This process is followed in Figure 8.12, which starts with the DD-path graph of the pseudocode triangle program. The if–then–else construct involving nodes B, C, D, and E is condensed into node a, and then the three if–then constructs are condensed onto nodes b, c, and d. The remaining if–then–else (which corresponds to the IF IsATriangle statement) is condensed into node e, resulting in a condensed graph with cyclomatic complexity $V(G) = 1$. In general, when a program is well structured (i.e., is composed solely of the structured programming constructs), it can always be reduced to a graph with one path.

The graph in Figure 8.10 cannot be reduced in this way (try it!). The loop with nodes B and C cannot be condensed because of the edge from B to E. Similarly, nodes D, E, and F look like an if–then construct, but the edge from B to E violates the structure. McCabe (1976) went on to find elemental “unstructures” that violate the precepts of structured programming. These are shown in Figure 8.13. Each of these violations contains three distinct paths, as opposed to the two paths present in the corresponding structured programming constructs;

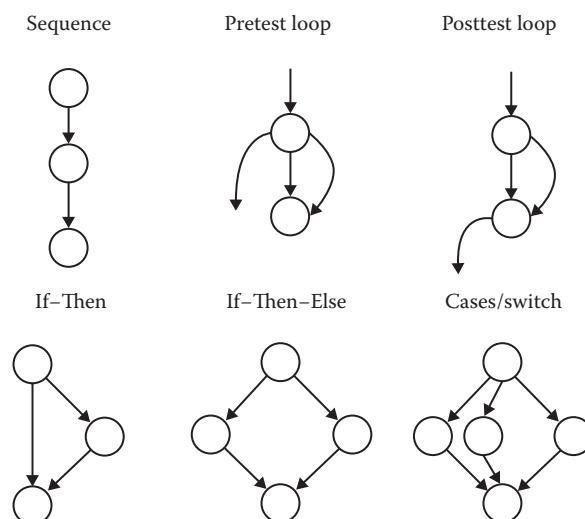


Figure 8.11 Structured programming constructs.

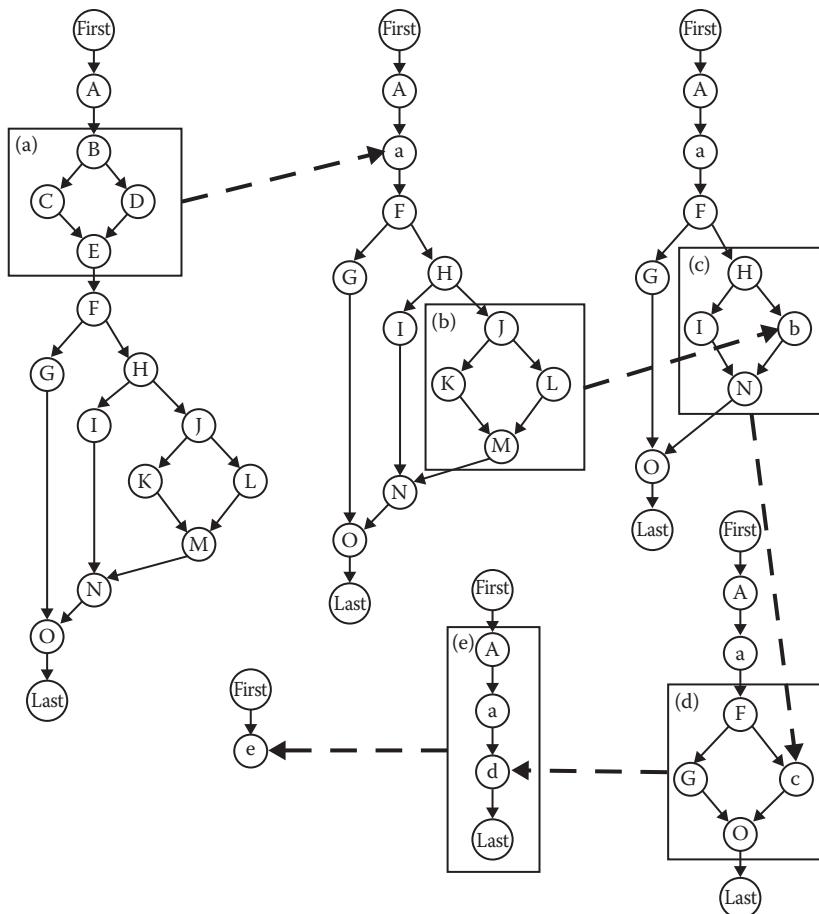


Figure 8.12 Condensing with respect to structured programming constructs.

so one conclusion is that such violations increase cyclomatic complexity. The *pièce de resistance* of McCabe's analysis is that these violations cannot occur by themselves: if one occurs in a program, there must be at least one more, so a program cannot be only slightly unstructured. Because these increase cyclomatic complexity, the minimum number of test cases is thereby increased. In the next chapter, we will see that the violations have interesting implications for data flow testing.

The bottom line for testers is this: programs with high cyclomatic complexity require more testing. Of the organizations that use the cyclomatic complexity metric, most set some guideline for maximum acceptable complexity; $V(G) = 10$ is a common choice. What happens if a unit has a higher complexity? Two possibilities: either simplify the unit or plan to do more testing. If the unit is well structured, its essential complexity is 1; so it can be simplified easily. If the unit has an essential complexity greater than 1, often the best choice is to eliminate the violations.

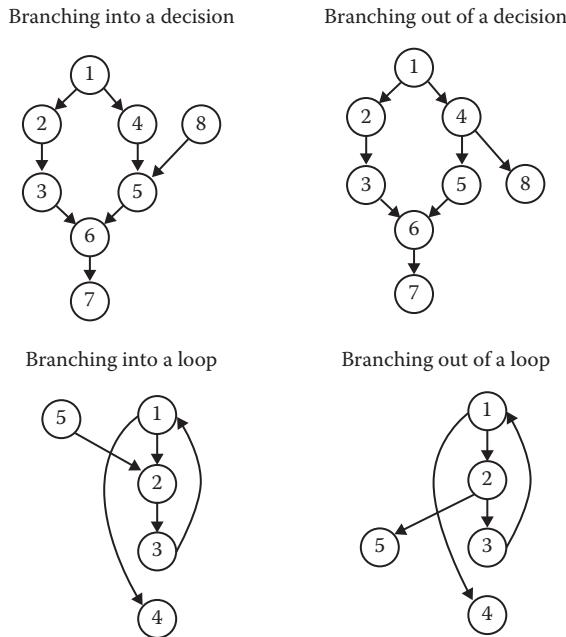


Figure 8.13 Violations of structured programming constructs.

8.5 Guidelines and Observations

In our study of specification-based testing, we observed that gaps and redundancies can both exist and, at the same time, cannot be recognized. The problem was that specification-based testing removes us too far from the code. The path testing approaches to code-based testing represent the case where the pendulum has swung too far the other way: moving from code to directed graph representations and program path formulations obscures important information that is present in the code, in particular the distinction between feasible and infeasible paths. Also, no form of code-based testing can reveal missing functionality that is specified in the requirements. In the next chapter, we look at data flow-based testing. These techniques move closer to the code, so the pendulum will swing back from the path analysis extreme.

McCabe (1982) was partly right when he observed, “It is important to understand that these are purely criteria that measure the quality of testing, and not a procedure to identify test cases.” He was referring to the DD-path coverage metric and his basis path heuristic based on cyclomatic complexity metric. Basis path testing therefore gives us a lower boundary on how much testing is necessary.

Path-based testing also provides us with a set of metrics that act as crosschecks on specification-based testing. We can use these metrics to resolve the gaps and redundancies question. When we find that the same program path is traversed by several functional test cases, we suspect that this redundancy is not revealing new faults. When we fail to attain DD-path coverage, we know that there are gaps in the functional test cases. As an example, suppose we have a program that contains extensive error handling, and we test it with boundary value test cases (min, min+, nom, max-, and max). Because these are all permissible values, DD-paths corresponding to the error-handling code will not be traversed. If we add test cases derived from robustness testing

or traditional equivalence class testing, the DD-path coverage will improve. Beyond this rather obvious use of coverage metrics, an opportunity exists for real testing craftsmanship. Any of the coverage metrics in Section 8.3 can operate in two ways: either as a blanket-mandated standard (e.g., all units shall be tested to attain full DD-path coverage) or as a mechanism to selectively test portions of code more rigorously than others. We might choose multiple-condition coverage for modules with complex logic, while those with extensive iteration might be tested in terms of the loop coverage techniques. This is probably the best view of structural testing: use the properties of the source code to identify appropriate coverage metrics, and then use these as a crosscheck on functional test cases. When the desired coverage is not attained, follow interesting paths to identify additional (special value) test cases.

EXERCISES

1. Find the cyclomatic complexity of the graph in Figure 8.3.
2. Identify a set of basis paths for the graph in Figure 8.3.
3. Discuss McCabe's concept of "flipping" for nodes with outdegree ≥ 3 .
4. Suppose we take Figure 8.3 as the DD-path graph of some program. Develop sets of paths (which would be test cases) for the C_0 , C_1 , and C_2 metrics.
5. Develop multiple-condition coverage test cases for the pseudocode triangle program. (Pay attention to the dependency between statement fragments 14 and 16 with the expression $(a = b) \text{ AND } (b = c)$.)
6. Rewrite the program segment 14–20 such that the compound conditions are replaced by nested if–then–else statements. Compare the cyclomatic complexity of your program with that of the existing version.
 14. If $(a = b) \text{ AND } (b = c)$
 15. Then Output ("Equilateral")
 16. Else If $(a \neq b) \text{ AND } (a \neq c) \text{ AND } (b \neq c)$
 17. Then Output ("Scalene")
 18. Else Output ("Isosceles")
 19. EndIf
 20. EndIf
7. Look carefully at the original statement fragments 14–20. What happens with a test case (e.g., $a = 3$, $b = 4$, $c = 3$) in which $a = c$? The condition in line 14 uses the transitivity of equality to eliminate the $a = c$ condition. Is this a problem?
8. The codeBasedTesting.xls Excel spreadsheet at the CRC website (www.crcpress.com/product/isbn/9781466560680) contains instrumented VBA implementations of the triangle, NextDate, and commission problems that you may have analyzed with the specBasedTesting.xls spreadsheet. The output shows the DD-path coverage of individual test cases and an indication of any faults revealed by a failing test case. Experiment with various sets of test cases to see if you can devise a set of test cases that has full DD-path coverage yet does not reveal the known faults.
9. (For mathematicians only.) For a set V to be a vector space, two operations (addition and scalar multiplication) must be defined for elements in the set. In addition, the following criteria must hold for all vectors x , y , and $z \in V$, and for all scalars k , l , 0, and 1:
 - a. If $x, y \in V$, the vector $x + y \in V$.
 - b. $x + y = y + x$.
 - c. $(x + y) + z = x + (y + z)$.
 - d. There is a vector $0 \in V$ such that $x + 0 = x$.

- e. For any $x \in V$, there is a vector $-x \in V$ such that $x + (-x) = 0$.
- f. For any $x \in V$, the vector $kx \in V$, where k is a scalar constant.
- g. $k(x + y) = kx + ky$.
- h. $(k + l)x = kx + lx$.
- i. $k(lx) = (kl)x$.
- j. $1x = x$.

How many of these 10 criteria hold for the “vector space” of paths in a program?

References

- Beizer, B., *Software Testing Techniques*, Van Nostrand, New York, 1984.
- Chilenski, J.J., *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*, DOT/FAA/AR-01/18, April 2001.
http://www.faa.gov/about/office_org/headquarters_offices/ang/offices/tc/library/ (see actlibrary.tc.faa.gov).
- Huang, J.C., Detection of dataflow anomaly through program instrumentation, *IEEE Transactions on Software Engineering*, Vol. SE-5, 1979, pp. 226–236.
- Miller, E.F. Jr., *Tutorial: Program Testing Techniques*, COMPSAC '77, IEEE Computer Society, 1977.
- Miller, E.F. Jr., Automated software testing: a technical perspective, *American Programmer*, Vol. 4, No. 4, April 1991, pp. 38–43.
- McCabe, T. J., A complexity metric, *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976, pp. 308–320.
- McCabe, T.J., *Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*, National Bureau of Standards (Now NIST), Special Publication 500-99, Washington, DC, 1982.
- McCabe, T.J., *Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*, McCabe and Associates, Baltimore, 1987.
- Perry, W.E., *A Structured Approach to Systems Testing*, QED Information Systems, Inc., Wellesley, MA, 1987.
- Schach, S.R., *Software Engineering*, 2nd ed., Richard D. Irwin, Inc. and Aksen Associates, Inc., Homewood, IL, 1993.

Chapter 9

Data Flow Testing

Data flow testing is an unfortunate term because it suggests some connection with data flow diagrams; no connection exists. Data flow testing refers to forms of structural testing that focus on the points at which variables receive values and the points at which these values are used (or referenced). We will see that data flow testing serves as a “reality check” on path testing; indeed, many of the data flow testing proponents (and researchers) see this approach as a form of path testing. While dataflow and slice-based testing are cumbersome at the unit level; they are well suited for object-oriented code. We will look at two mainline forms of data flow testing: one provides a set of basic definitions and a unifying structure of test coverage metrics, while the other is based on a concept called a “program slice.” Both of these formalize intuitive behaviors (and analyses) of testers; and although they both start with a program graph, both move back in the direction of functional testing. Also, both of these methods are difficult to perform manually, and unfortunately, few commercial tools exist to make life easier for the data flow and slicing testers. On the positive side, both techniques are helpful for coding and debugging.

Most programs deliver functionality in terms of data. Variables that represent data somehow receive values, and these values are used to compute values for other variables. Since the early 1960s, programmers have analyzed source code in terms of the points (statements and statement fragments) at which variables receive values and points at which these values are used. Many times, their analyses were based on concordances that list statement numbers in which variable names occur. Concordances were popular features of second-generation language compilers (they are still popular with COBOL programmers). Early data flow analyses often centered on a set of faults that are now known as define/reference anomalies:

- A variable that is defined but never used (referenced)
- A variable that is used before it is defined
- A variable that is defined twice before it is used

Each of these anomalies can be recognized from the concordance of a program. Because the concordance information is compiler generated, these anomalies can be discovered by what is known as static analysis: finding faults in source code without executing it.

9.1 Define/Use Testing

Much of the formalization of define/use testing was done in the early 1980s (Rapps and Weyuker, 1985); the definitions in this section are compatible with those in Clarke et al. (1989), which summarizes most define/use testing theory. This body of research is very compatible with the formulation we developed in Chapters 4 and 8. It presumes a program graph in which nodes are statement fragments (a fragment may be an entire statement) and programs that follow the structured programming precepts.

The following definitions refer to a program P that has a program graph $G(P)$ and a set of program variables V . The program graph $G(P)$ is constructed as in Chapter 4, with statement fragments as nodes and edges that represent node sequences. $G(P)$ has a single-entry node and a single-exit node. We also disallow edges from a node to itself. Paths, subpaths, and cycles are as they were in Chapter 4. The set of all paths in P is $\text{PATHS}(P)$.

Definition

Node $n \in G(P)$ is a *defining node* of the variable $v \in V$, written as $\text{DEF}(v, n)$, if and only if the value of variable v is defined as the statement fragment corresponding to node n .

Input statements, assignment statements, loop control statements, and procedure calls are all examples of statements that are defining nodes. When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables are changed.

Definition

Node $n \in G(P)$ is a *usage node* of the variable $v \in V$, written as $\text{USE}(v, n)$, if and only if the value of the variable v is used as the statement fragment corresponding to node n .

Output statements, assignment statements, conditional statements, loop control statements, and procedure calls are all examples of statements that are usage nodes. When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables remain unchanged.

Definition

A usage node $\text{USE}(v, n)$ is a *predicate use* (denoted as P-use) if and only if the statement n is a predicate statement; otherwise, $\text{USE}(v, n)$ is a computation use (denoted C-use).

The nodes corresponding to predicate uses always have an outdegree ≥ 2 , and nodes corresponding to computation uses always have an outdegree ≤ 1 .

Definition

A *definition/use path* with respect to a variable v (denoted du-path) is a path in $\text{PATHS}(P)$ such that, for some $v \in V$, there are define and usage nodes $\text{DEF}(v, m)$ and $\text{USE}(v, n)$ such that m and n are the initial and final nodes of the path.

Definition

A *definition-clear path* with respect to a variable v (denoted dc-path) is a definition/use path in $\text{PATHS}(P)$ with initial and final nodes $\text{DEF}(v, m)$ and $\text{USE}(v, n)$ such that no other node in the path is a defining node of v .

Testers should notice how these definitions capture the essence of computing with stored data values. Du-paths and dc-paths describe the flow of data across source statements from points at which the values are defined to points at which the values are used. Du-paths that are not definition clear are potential trouble spots. One of the main values of du-paths is they identify points for variable “watches” and breakpoints when code is developed in an Integrated Development Environment. Figure 9.3 illustrates this very well later in the chapter.

9.1.1 Example

We will use the commission problem and its program graph to illustrate these definitions. The numbered pseudocode and its corresponding program graph are shown in Figure 9.1. This program computes the commission on the sales of the total numbers of locks, stocks, and barrels sold. The while loop is a classic sentinel controlled loop in which a value of -1 for locks signifies the end of the sales data. The totals are accumulated as the data values are read in the while loop. After printing this preliminary information, the sales value is computed, using the constant item prices defined at the beginning of the program. The sales value is then used to compute the commission in the conditional portion of the program.

Figure 9.2 shows the decision-to-decision path (DD-path) graph of the program graph in Figure 9.1. More compression exists in this DD-path graph because of the increased computation in the commission problem. Table 9.1 details the statement fragments associated with DD-paths. Some DD-paths (per the definition in Chapter 8) are combined to simplify the graph. We will need this figure later to help visualize the differences among DD-paths, du-paths, and program slices.

Table 9.2 lists the define and usage nodes for the variables in the commission problem. We use this information in conjunction with the program graph in Figure 9.1 to identify various definition/use and definition-clear paths. It is a judgment call whether nonexecutable statements such as constant and variable declaration statements should be considered as defining nodes. Such nodes are not very interesting when we follow what happens along their du-paths; but if something is wrong, it can be helpful to include them. Take your pick. We will refer to the various paths as sequences of node numbers.

Tables 9.3 and 9.4 present some of the du-paths in the commission problem; they are named by their beginning and ending nodes (from Figure 9.1). The third column in Table 9.3 indicates whether the du-paths are definition clear. Some of the du-paths are trivial—for example, those for lockPrice, stockPrice, and barrelPrice. Others are more complex: the while loop (node sequence $<14, 15, 16, 17, 18, 19, 20>$) inputs and accumulated values for totalLocks, totalStocks, and totalBarrels. Table 9.3 only shows the details for the totalStocks variable. The initial value definition for totalStocks occurs at node 11, and it is first used at node 17. Thus, the path $(11, 17)$, which consists of the node sequence $<11, 12, 13, 14, 15, 16, 17>$, is definition clear. The path $(11, 22)$, which consists of the node sequence $<11, 12, 13, (14, 15, 16, 17, 18, 19, 20)^*, 21, 22>$, is not definition clear because values of totalStocks are defined at node 11 and (possibly several times at) node 17. (The asterisk after the while loop is the Kleene Star notation used both in formal logic and regular expressions to denote zero or more repetitions.)

```

1 Program Commission (INPUT,OUTPUT)
2 Dim locks, stocks, barrels As Integer
3 Dim lockPrice, stockPrice, barrelPrice As Real
4 Dim totalLocks, totalStocks, totalBarrels As Integer
5 Dim lockSales, stockSales, barrelSales As Real
6 Dim sales, commission As Real
7 lockPrice = 45.0
8 stockPrice = 30.0
9 barrelPrice = 25.0
10 totalBarrels = 0
11 totalStocks = 0
12 totalBarrels = 0
13 Input(locks)
14 While NOT(locks = -1) "locks = -1 signals end of data
15 Input(stocks, barrels)
16 totalLocks = totalLocks + locks
17 totalStocks = totalStocks + stocks
18 totalBarrels = totalBarrels + barrels
19 Input(locks)
20 EndWhile
21 Output("Locks sold;," totalLocks)
22 Output("Stocks sold;," totalStocks)
23 Output("Barrels sold;," totalBarrels)
24 lockSales = lockPrice*totalLocks
25 stockSales = stockPrice*totalStocks
26 barrelsSales = barrelPrice * totalBarrels
27 sales = lockSales + stockSales + barrelSales
28 Output("Total sales: ", sales)
29 If (sales > 1800.0)
30 Then
31 commission = 0.10 * 1000.0
32 commission = commission + 0.15 * 800.0
33 commission = commission + 0.20*(sales-1800.0)
34 Else If (sales > 1000.0)
35 Then
36 commission = 0.10 * 1000.0
37 commission = commission + 0.15*(sales-1000.0)
38 Else
39 commission = 0.10 * sales
40 Endif
41 EndIf
42 Output("Commission is $", commission)
43 End Commission

```

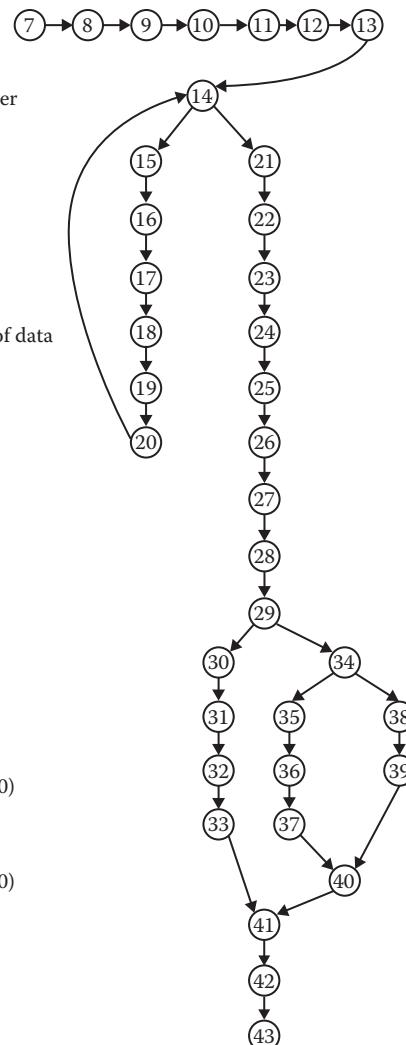


Figure 9.1 Commission problem and its program graph.

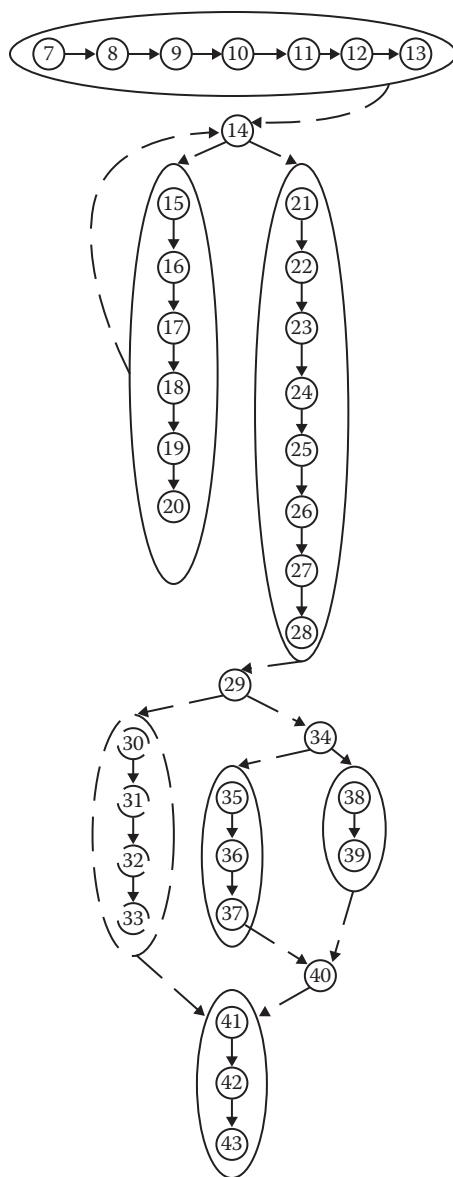


Figure 9.2 DD-path graph of commission problem pseudocode (in Figure 9.1).

9.1.2 Du-paths for Stocks

First, let us look at a simple path: the du-path for the variable stocks. We have DEF(stocks, 15) and USE(stocks, 17), so the path <15, 17> is a du-path with respect to stocks. No other defining nodes are used for stocks; therefore, this path is also definition clear.

9.1.3 Du-paths for Locks

Two defining and two usage nodes make the locks variable more interesting: we have DEF(locks, 13), DEF(locks, 19), USE(locks, 14), and USE(locks, 16). These yield four du-paths; they are shown in Figure 9.3.

```
p1 = <13, 14>
p2 = <13, 14, 15, 16>
p3 = <19, 20, 14>
p4 = <19, 20, 14, 15, 16>
```

Note: du-paths p1 and p2 refer to the priming value of locks, which is read at node 13. The locks variable has a predicate use in the while statement (node 14), and if the condition is true (as in path p2), a computation use at statement 16. The other two du-paths start near the end of the while loop and occur when the loop repeats. These paths provide the loop coverage discussed in Chapter 8—bypass the loop, begin the loop, repeat the loop, and exit the loop. All these du-paths are definition clear.

```
13 Input(locks)
14 While NOT(locks = -1) 'locks = -1 signals end of data
15 Input(stocks, barrels)
16 totalLocks = totalLocks + locks
17 totalStocks = totalStocks + stocks
18 totalBarrels = totalBarrels + barrels
19 Input(locks)
20 EndWhile
```

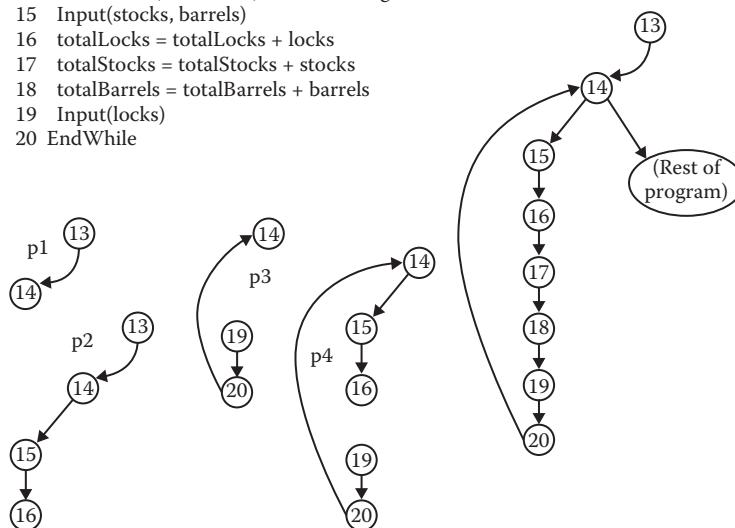


Figure 9.3 Du-paths for locks.

Table 9.1 DD-paths in Figure 9.1

<i>DD-path</i>	<i>Nodes</i>
A	7, 8, 9, 10, 11, 12, 13
B	14
C	15, 16, 17, 18, 19, 20
D	21, 22, 23, 24, 25, 26, 27, 28
E	29
F	30, 31, 32, 33
G	34
H	35, 36, 37
I	38
J	39
K	40
L	41, 42

Table 9.2 Define/Use Nodes for Variables in Commission Problem

<i>Variable</i>	<i>Defined at Node</i>	<i>Used at Node</i>
lockPrice	7	24
stockPrice	8	25
barrelPrice	9	26
totalLocks	10, 16	16, 21, 24
totalStocks	11, 17	17, 22, 25
totalBarrels	12, 18	18, 23, 26
Locks	13, 19	14, 16
Stocks	15	17
Barrels	15	18
lockSales	24	27
stockSales	25	27
barrelSales	26	27
Sales	27	28, 29, 33, 34, 37, 38
Commission	31, 32, 33, 36, 37, 38	32, 33, 37, 41

Table 9.3 Selected Define/Use Paths

<i>Variable</i>	<i>Path (Beginning, End) Nodes</i>	<i>Definition Clear?</i>
lockPrice	7, 24	Yes
stockPrice	8, 25	Yes
barrelPrice	9, 26	Yes
totalStocks	11, 17	Yes
totalStocks	11, 22	No
totalStocks	11, 25	No
totalStocks	17, 17	Yes
totalStocks	17, 22	No
totalStocks	17, 25	No
Locks	13, 14	Yes
Locks	13, 16	Yes
Locks	19, 14	Yes
Locks	19, 16	Yes
Sales	27, 28	Yes
Sales	27, 29	Yes
Sales	27, 33	Yes
Sales	27, 34	Yes
Sales	27, 37	Yes
Sales	27, 38	Yes

Table 9.4 Define/Use Paths for Commission

<i>Variable</i>	<i>Path (Beginning, End) Nodes</i>	<i>Feasible?</i>	<i>Definition Clear?</i>
Commission	31, 32	Yes	Yes
Commission	31, 33	Yes	No
Commission	31, 37	No	N/A
Commission	31, 41	Yes	No
Commission	32, 32	Yes	Yes
Commission	32, 33	Yes	Yes
Commission	32, 37	No	N/A
Commission	32, 41	Yes	No
Commission	33, 32	No	N/A
Commission	33, 33	Yes	Yes
Commission	33, 37	No	N/A
Commission	33, 41	Yes	Yes
Commission	36, 32	No	N/A
Commission	36, 33	No	N/A
Commission	36, 37	Yes	Yes
Commission	36, 41	Yes	No
Commission	37, 32	No	N/A
Commission	37, 33	No	N/A
Commission	37, 37	Yes	Yes
Commission	37, 41	Yes	Yes
Commission	38, 32	No	N/A
Commission	38, 33	No	N/A
Commission	38, 37	No	N/A
Commission	38, 41	Yes	Yes

9.1.4 Du-paths for totalLocks

The du-paths for totalLocks will lead us to typical test cases for computations. With two defining nodes (DEF(totalLocks, 10) and DEF(totalLocks, 16)) and three usage nodes (USE(totalLocks, 16), USE(totalLocks, 21), USE(totalLocks, 24)), we might expect six du-paths. Let us take a closer look.

Path p5 = <10, 11, 12, 13, 14, 15, 16> is a du-path in which the initial value of totalLocks (0) has a computation use. This path is definition clear. The next path is problematic:

p6 = <10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 14, 21>

```

10 totalLocks = 0
11 totalStocks = 0
12 totalBarrels = 0
13 Input(locks)
  locks = -1 signals end of data
14 While NOT(locks = -1)
15   Input(stocks, barrels)
16   totalLocks = totalLocks + locks
17   totalStocks = totalStocks + stocks
18   totalBarrels = totalBarrels + barrels
19   Input(locks)
20 EndWhile
21 Output("Locks sold;" totalLocks)
22 Output("Stocks sold;" totalStocks)
23 Output("Barrels sold;" totalBarrels)
24 lockSales = lockPrice*totalLocks

```

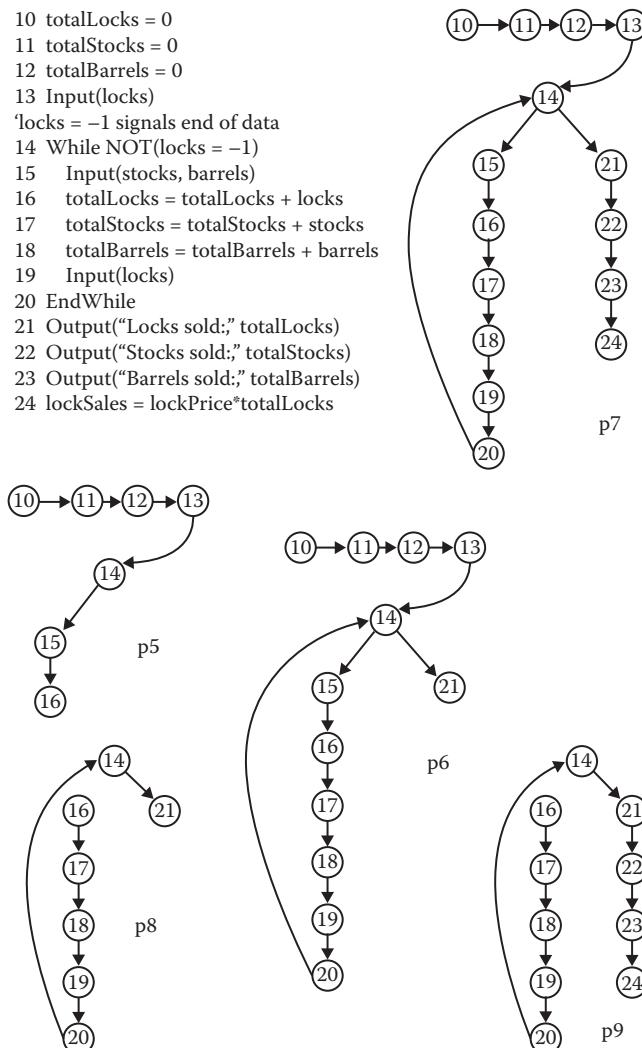


Figure 9.4 Du-paths for totalLocks.

Path p6 ignores the possible repetition of the while loop. We could highlight this by noting that the subpath $\langle 16, 17, 18, 19, 20, 14, 15 \rangle$ might be traversed several times. Ignoring this for now, we still have a du-path that fails to be definition clear. If a problem occurs with the value of totalLocks at node 21 (the Output statement), we should look at the intervening DEF(totalLocks, 16) node.

The next path contains p6; we can show this by using a path name in place of its corresponding node sequence:

```
p7 = <10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 14, 21, 22, 23, 24>
p7 = <p6, 22, 23, 24>
```

Du-path p7 is not definition clear because it includes node 16. Subpaths that begin with node 16 (an assignment statement) are interesting. The first, $\langle 16, 16 \rangle$, seems degenerate. If we “expanded” it into machine code, we would be able to separate the define and usage portions. We will disallow these as du-paths. Technically, the usage on the right-hand side of the assignment refers to a value defined at node 10 (see path p5). The remaining two du-paths are both subpaths of p7:

```
p8 = <16, 17, 18, 19, 20, 14, 21>
p9 = <16, 17, 18, 19, 20, 14, 21, 22, 23, 24>
```

Both are definition clear, and both have the loop iteration problem we discussed before. The du-paths for totalLocks are shown in Figure 9.4.

9.1.5 Du-paths for Sales

There is one defining node for sales; therefore, all the du-paths with respect to sales must be definition clear. They are interesting because they illustrate predicate and computation uses. The first three du-paths are easy:

```
p10 = <27, 28>
p11 = <27, 28, 29>
p12 = <27, 28, 29, 30, 31, 32, 33>
```

Notice that p12 is a definition-clear path with three usage nodes; it also contains paths p10 and p11. If we were testing with p12, we know we would also have covered the other two paths. We will revisit this toward the end of the chapter.

The IF, ELSE IF logic in statements 29 through 40 highlights an ambiguity in the original research. Two choices for du-paths begin with path p11: one choice is the path $\langle 27, 28, 29, 30, 31, 32, 33 \rangle$, and the other is the path $\langle 27, 28, 29, 34 \rangle$. The remaining du-paths for sales are

```
p13 = <27, 28, 29, 34>
p14 = <27, 28, 29, 34, 35, 36, 37>
p15 = <27, 28, 29, 34, 38>
```

Note that the dynamic view is very compatible with the kind of thinking we used for DD-paths in Chapter 8.

9.1.6 Du-paths for Commission

If you have followed this discussion carefully, you are probably dreading the analysis of du-paths with respect to commission. You are right—it is time for a change of pace. In statements 29 through 41, the calculation of commission is controlled by ranges of the variable sales. Statements 31 to 33 build up the value of commission by using the memory location to hold intermediate values. This is a common programming practice, and it is desirable because it shows how the final value is computed. (We could replace these lines with the statement “commission: = 220 + 0.20 * (sales -1800),” where 220 is the value of $0.10 * 1000 + 0.15 * 800$, but this would be hard for a maintainer to understand.) The “built-up” version uses intermediate values, and these will appear as define and usage nodes in the du-path analysis. We decided to disallow du-paths from assignment statements like 31 and 32, so we will just consider du-paths that begin with the three “real” defining nodes: DEF(commission, 33), DEF(commission, 37), and DEF(commission, 39). Only one usage node is used: USE(commission, 41).

9.1.7 Define/Use Test Coverage Metrics

The whole point of analyzing a program with definition/use paths is to define a set of test coverage metrics known as the Rapps–Weyuker data flow metrics (Rapps and Weyuker, 1985). The first three of these are equivalent to three of E.F. Miller’s metrics in Chapter 8: All-Paths, All-Edges, and All-Nodes. The others presume that define and usage nodes have been identified for all program variables, and that du-paths have been identified with respect to each variable. In the following definitions, T is a set of paths in the program graph $G(P)$ of a program P , with the set V of variables. It is not enough to take the cross product of the set of DEF nodes with the set of USE nodes for a variable to define du-paths. This mechanical approach can result in infeasible paths. In the next definitions, we assume that the define/use paths are all feasible.

Definition

The set T satisfies the *All-Defs criterion* for the program P if and only if for every variable $v \in V$, T contains definition-clear paths from every defining node of v to a use of v .

Definition

The set T satisfies the *All-Uses criterion* for the program P if and only if for every variable $v \in V$, T contains definition-clear paths from every defining node of v to every use of v , and to the successor node of each $\text{USE}(v, n)$.

Definition

The set T satisfies the *All-P-Uses/Some C-Uses criterion* for the program P if and only if for every variable $v \in V$, T contains definition-clear paths from every defining node of v to every predicate use of v ; and if a definition of v has no P-uses, a definition-clear path leads to at least one computation use.

Definition

The set T satisfies the *All-C-Uses/Some P-Uses criterion* for the program P if and only if for every variable $v \in V$, T contains definition clear paths from every defining node of v to every computation use of v ; and if a definition of v has no C-uses, a definition-clear path leads to at least one predicate use.

Definition

The set T satisfies the *All-DU-paths criterion* for the program P if and only if for every variable $v \in V$, T contains definition-clear paths from every defining node of v to every use of v and to the successor node of each $\text{USE}(v, n)$, and that these paths are either single loop traversals or they are cycle free.

These test coverage metrics have several set-theory-based relationships, which are referred to as “subsumption” in Rapps and Weyuker (1985). These relationships are shown in Figure 9.5. We now have a more refined view of structural testing possibilities between the extremes of the (typically unattainable) All-Paths metric and the generally accepted minimum, All-Edges. What good is all this? Define/use testing provides a rigorous, systematic way to examine points at which faults may occur.

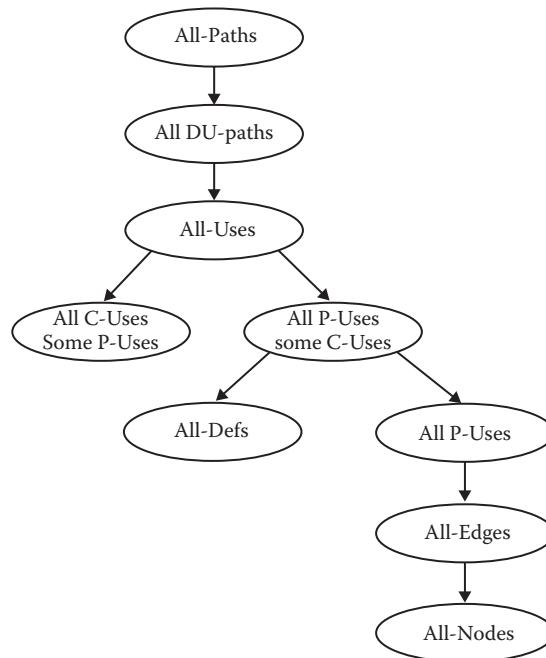


Figure 9.5 Rapps–Weyuker hierarchy of data flow coverage metrics.

9.1.8 Define/Use Testing for Object-Oriented Code

All of the define/use definitions thus far make no mention of where the variable is defined and where it is used. In a procedural code, this is usually assumed to be within a unit, but it can involve procedure calls to improperly coupled units. We might make this distinction by referring to these definitions as “context free”; that is, the places where variables are defined and used are independent. The object-oriented paradigm changes this—we must now consider the define and use locations with respect to class aggregation, inheritance, dynamic binding, and polymorphism. The bottom line is that data flow testing for object-oriented code moves from the unit level to the integration level.

9.2 Slice-Based Testing

Program slices have surfaced and submerged in software engineering literature since the early 1980s. They were proposed in Mark Weiser’s dissertation in 1979 (Weiser, 1979), made more generally available in Weiser (1985), used as an approach to software maintenance in Gallagher and Lyle (1991), and more recently used to quantify functional cohesion in Bieman (1994). During the early 1990s, there was a flurry of published activity on slices, including a paper (Ball and Eick, 1994) describing a program to visualize program slices. This latter paper describes a tool used in industry. (Note that it took about 20 years to move a seminal idea into industrial practice.)

Part of the utility and versatility of program slices is due to the natural, intuitively clear intent of the concept. Informally, a program slice is a set of program statements that contributes to, or affects the value of, a variable at some point in a program. This notion of slice corresponds to other disciplines as well. We might study history in terms of slices—US history, European history, Russian history, Far East history, Roman history, and so on. The way such historical slices interact turns out to be very analogous to the way program slices interact.

We will start by growing our working definition of a program slice. We continue with the notation we used for define/use paths: a program P that has a program graph $G(P)$ and a set of program variables V . The first try refines the definition in Gallagher and Lyle (1991) to allow nodes in $P(G)$ to refer to statement fragments.

Definition

Given a program P and a set V of variables in P , a *slice on the variable set V at statement n* , written $S(V, n)$, is the set of all statement fragments in P that contribute to the values of variables in V at node n .

One simplifying notion—in our discussion, the set V of variables consists of a single variable, v . Extending this to sets of more than one variable is both obvious and cumbersome. For sets V with more than one variable, we just take the union of all the slices on the individual variables of V . There are two basic questions about program slices, whether they are backward or forward slices, and whether they are static or dynamic. Backward slices refer to statement fragments that contribute to the value of v at statement n . Forward slices refer to all the program statements that are affected by the value of v and statement n . This is one place where the define/use notions are helpful. In a backward slice $S(v, n)$, statement n is nicely understood as a Use node of the variable v , that is, $\text{Use}(v, n)$. Forward slices are not as easily described, but they certainly depend on predicate uses and computation uses of the variable v .

The static/dynamic dichotomy is more complex. We borrow two terms from database technology to help explain the difference. In database parlance, we can refer to the intension and extensions of a database. The intension (it is unique) is the fundamental database structure, presumably expressed in a data modeling language. Populating a database creates an extension, and changes to a populated database all result in new extensions. With this in mind, a static backward slice $S(v, n)$ consists of all the statements in a program that determine the value of variable v at statement n , independent of values used in the statements. Dynamic slices refer to execution-time execution of portions of a static slice with specific values of all variables in $S(v, n)$. This is illustrated in Figures 9.6 and 9.7.

Listing elements of a slice $S(V, n)$ will be cumbersome because, technically, the elements are program statement fragments. It is much simpler to list the statement fragment numbers in $P(G)$, so we make the following trivial change.

Definition

Given a program P and a program graph $G(P)$ in which statements and statement fragments are numbered, and a set V of variables in P , the static, backward slice on the variable set V at statement fragment n , written $S(V, n)$, is the set of node numbers of all statement fragments in P that contribute to the values of variables in V at statement fragment n .

The idea of program slicing is to separate a program into components that have some useful (functional) meaning. Another refinement is whether or not a program slice is executable. Adding all the data declaration statements and other syntactically necessary statements clearly increases the size of a slice, but the full version can be compiled and separately executed and tested. Further, such compilable slices can be “spliced” together (Gallagher and Lyle, 1991) as a bottom-up way to develop a program. As a test of clear diction, Gallagher and Lyle suggest the term “slice splicing.” In a sense, this is a precursor to agile programming. The alternative is to just consider program fragments, which we do here for space and clarity considerations. Eventually, we will develop a

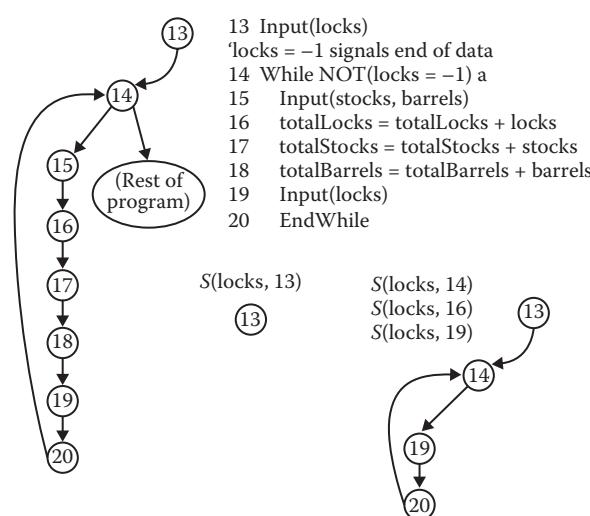
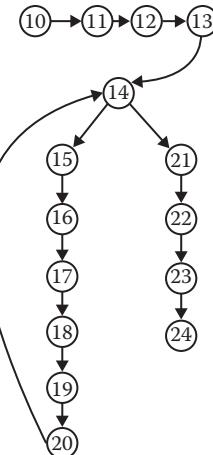


Figure 9.6 Selected slices on locks.

```

10 totalLocks = 0
11 totalStocks = 0
12 totalBarrels = 0
13 Input(locks)
'locks = -1 signals end of data
14 While NOT(locks = -1)
15   Input(stocks, barrels)
16   totalLocks = totalStocks + stocks
17   totalStocks = totalStocks + stocks
18   totalBarrels = totalBarrels + barrels
19   Input(locks)
20 EndWhile
21 Output("Locks sold:", totalLocks)
22 Output("Stocks sold:", totalStocks)
23 Output("Barrels sold:", totalBarrels)
24 lockSales = lockPrice*totalLocks

```



$S(\text{totalLocks}, 10)$ $S(\text{totalLocks}, 16)$ $S(\text{totalStocks}, 17)$

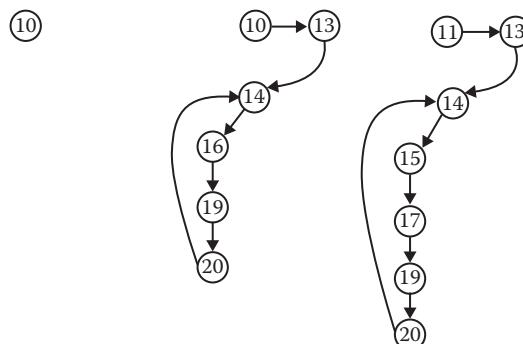


Figure 9.7 Selected slices in a loop.

lattice (a directed, acyclic graph) of static slices, in which nodes are slices and edges correspond to the subset relationship.

The “contribute” part is more complex. In a sense, data declaration statements have an effect on the value of a variable. For now, we only include all executable statements. The notion of contribution is partially clarified by the predicate (P-use) and computation (C-use) usage distinction of Rapps and Weyuker (1985), but we need to refine these forms of variable usage. Specifically, the USE relationship pertains to five forms of usage:

P-use	used in a predicate (decision)
C-use	used in computation
O-use	used for output
L-use	used for location (pointers, subscripts)
I-use	iteration (internal counters, loop indices)

Most of the literature on program slices just uses P-uses and C-uses. While we are at it, we identify two forms of definition nodes:

I-def	defined by input
A-def	defined by assignment

Recall our simplification that the slice $S(V, n)$ is a slice on one variable; that is, the set V consists of a single variable, v . If statement fragment n is a defining node for v , then n is included in the slice. If statement fragment n is a usage node for v , then n is not included in the slice. If a statement is both a defining and a usage node, then it is included in the slice. In a static slice, P-uses and C-uses of other variables (not the v in the slice set V) are included to the extent that their execution affects the value of the variable v . As a guideline, if the value of v is the same whether a statement fragment is included or excluded, exclude the statement fragment.

L-use and I-use variables are typically invisible outside their units, but this hardly precludes the problems such variables often create. Another judgment call: here (with some peril), we choose to exclude these from the intent of “contribute.” Thus, O-use, L-use, and I-use nodes are excluded from slices.

9.2.1 Example

The commission problem is used in this book because it contains interesting data flow properties, and these are not present in the triangle problem (nor in NextDate). In the following, except where specifically noted, we are speaking of static backward slices and we only include nodes corresponding to executable statement fragments. The examples refer to the source code for the commission problem in Figure 9.1. There are 42 “interesting” static backward slices in our example. They are named in Table 9.5. We will take a selective look at some interesting slices.

The first six slices are the simplest—they are the nodes where variables are initialized.

Table 9.5 Slices in Commission Problem

$S_1: S(lockPrice, 7)$	$S_{15}: S(barrels, 18)$	$S_{29}: S(barrelSales, 26)$
$S_2: S(stockPrice, 8)$	$S_{16}: S(totalBarrels, 18)$	$S_{30}: S(sales, 27)$
$S_3: S(barrelPrice, 9)$	$S_{17}: S(locks, 19)$	$S_{31}: S(sales, 28)$
$S_4: S(totalLocks, 10)$	$S_{18}: S(totalLocks, 21)$	$S_{32}: S(sales, 29)$
$S_5: S(totalStocks, 11)$	$S_{19}: S(totalStocks, 22)$	$S_{33}: S(sales, 33)$
$S_6: S(totalBarrels, 12)$	$S_{20}: S(totalBarrels, 23)$	$S_{34}: S(sales, 34)$
$S_7: S(locks, 13)$	$S_{21}: S(lockPrice, 24)$	$S_{35}: S(sales, 37)$
$S_8: S(locks, 14)$	$S_{22}: S(totalLocks, 24)$	$S_{36}: S(sales, 39)$
$S_9: S(stocks, 15)$	$S_{23}: S(lockSales, 24)$	$S_{37}: S(commission, 31)$
$S_{10}: S(barrels, 15)$	$S_{24}: S(stockPrice, 25)$	$S_{38}: S(commission, 32)$
$S_{11}: S(locks, 16)$	$S_{25}: S(totalStocks, 25)$	$S_{39}: S(commission, 33)$
$S_{12}: S(totalLocks, 16)$	$S_{26}: S(stockSales, 25)$	$S_{40}: S(commission, 36)$
$S_{13}: S(stocks, 17)$	$S_{27}: S(barrelPrice, 26)$	$S_{41}: S(commission, 37)$
$S_{14}: S(totalStocks, 17)$	$S_{28}: S(totalBarrels, 26)$	$S_{42}: S(commission, 39)$

$S_1: S(\text{lockPrice}, 7)$	$= \{7\}$
$S_2: S(\text{stockPrice}, 8)$	$= \{8\}$
$S_3: S(\text{barrelPrice}, 9)$	$= \{9\}$
$S_4: S(\text{totalLocks}, 10)$	$= \{10\}$
$S_5: S(\text{totalStocks}, 11)$	$= \{11\}$
$S_6: S(\text{totalBarrels}, 12)$	$= \{12\}$

Slices 7 through 17 focus on the sentinel controlled while loop in which the totals for locks, stocks, and barrels are accumulated. The locks variable has two uses in this loop: a P-use at fragment 14 and C-use at statement 16. It also has two defining nodes, at statements 13 and 19. The stocks and barrels variables have a defining node at 15, and computation uses at nodes 17 and 18, respectively. Notice the presence of all relevant statement fragments in slice 8. The slices on locks are shown in Figure 9.6.

$S_7: S(\text{locks}, 13)$	$= \{13\}$
$S_8: S(\text{locks}, 14)$	$= \{13, 14, 19, 20\}$
$S_9: S(\text{stocks}, 15)$	$= \{13, 14, 15, 19, 20\}$
$S_{10}: S(\text{barrels}, 15)$	$= \{13, 14, 15, 19, 20\}$
$S_{11}: S(\text{locks}, 16)$	$= \{13, 14, 19, 20\}$
$S_{12}: S(\text{totalLocks}, 16)$	$= \{10, 13, 14, 16, 19, 20\}$
$S_{13}: S(\text{stocks}, 17)$	$= \{13, 14, 15, 19, 20\}$
$S_{14}: S(\text{totalStocks}, 17)$	$= \{11, 13, 14, 15, 17, 19, 20\}$
$S_{15}: S(\text{barrels}, 18)$	$= \{12, 13, 14, 15, 19, 20\}$
$S_{16}: S(\text{totalBarrels}, 18)$	$= \{12, 13, 14, 15, 18, 19, 20\}$
$S_{17}: S(\text{locks}, 19)$	$= \{13, 14, 19, 20\}$

Slices 18, 19, and 20 are output statements, and none of the variables is defined; hence, the corresponding statements are not included in these slices.

$S_{18}: S(\text{totalLocks}, 21)$	$= \{10, 13, 14, 16, 19, 20\}$
$S_{19}: S(\text{totalStocks}, 22)$	$= \{11, 13, 14, 15, 17, 19, 20\}$
$S_{20}: S(\text{totalBarrels}, 23)$	$= \{12, 13, 14, 15, 18, 19, 20\}$

Slices 21 through 30 deal with the calculation of the variable sales. As an aside, we could simply write $S_{30}: S(\text{sales}, 27) = S_{23} \cup S_{26} \cup S_{29} \cup \{27\}$. This is more like the form that Weiser (1979) refers to in his dissertation—a natural way to think about program fragments. Gallagher and Lyle (1991) echo this as a thought pattern among maintenance programmers. This also leads to Gallagher's "slice splicing" concept. Slice S_{23} computes the total lock sales, S_{25} the total stock sales, and S_{28} the total barrel sales. In a bottom-up way, these slices could be separately coded and tested, and later spliced together. "Splicing" is actually an apt metaphor—anyone who has ever spliced a twisted rope line knows that splicing involves carefully merging individual strands at just the right places. (See Figure 9.7 for the effect of looping on a slice.)

$S_{21}: S(\text{lockPrice}, 24)$	$= \{7\}$
$S_{22}: S(\text{totalLocks}, 24)$	$= \{10, 13, 14, 16, 19, 20\}$
$S_{23}: S(\text{lockSales}, 24)$	$= \{7, 10, 13, 14, 16, 19, 20, 24\}$
$S_{24}: S(\text{stockPrice}, 25)$	$= \{8\}$

$S_{25}: S(\text{totalStocks}, 25) = \{11, 13, 14, 15, 17, 19, 20\}$
 $S_{26}: S(\text{stockSales}, 25) = \{8, 11, 13, 14, 15, 17, 19, 20, 25\}$
 $S_{27}: S(\text{barrelPrice}, 26) = \{9\}$
 $S_{28}: S(\text{totalBarrels}, 26) = \{12, 13, 14, 15, 18, 19, 20\}$
 $S_{29}: S(\text{barrelSales}, 26) = \{9, 12, 13, 14, 15, 18, 19, 20, 26\}$
 $S_{30}: S(\text{sales}, 27) = \{7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27\}$

Slices 31 through 36 are identical. Slice S_{31} is an O-use of sales; the others are all C-uses. Since none of these changes the value of sales defined at S_{30} , we only show one set of statement fragment numbers here.

$S_{31}: S(\text{sales}, 28) = \{7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27\}$

The last seven slices deal with the calculation of commission from the value of sales. This is literally where it all comes together.

$S_{37}: S(\text{commission}, 31) = \{7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31\}$
 $S_{38}: S(\text{commission}, 32) = \{7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31, 32\}$
 $S_{39}: S(\text{commission}, 33) = \{7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31, 32, 33\}$
 $S_{40}: S(\text{commission}, 36) = \{7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 34, 35, 36\}$
 $S_{41}: S(\text{commission}, 37) = \{7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 34, 35, 36, 37\}$
 $S_{42}: S(\text{commission}, 39) = \{7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 34, 38, 39\}$
 $S_{43}: S(\text{commission}, 41) = \{7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 37, 39\}$

Looking at slices as sets of fragment numbers (Figure 9.8) is correct in terms of our definition, but it is also helpful to see how slices are composed of sets of previous slices. We do this next, and show the final lattice in Figure 9.9.

$S_1: S(\text{lockPrice}, 7) = \{7\}$
 $S_2: S(\text{stockPrice}, 8) = \{8\}$
 $S_3: S(\text{barrelPrice}, 9) = \{9\}$
 $S_4: S(\text{totalLocks}, 10) = \{10\}$
 $S_5: S(\text{totalStocks}, 11) = \{11\}$
 $S_6: S(\text{totalBarrels}, 12) = \{12\}$
 $S_7: S(\text{locks}, 13) = \{13\}$
 $S_8: S(\text{locks}, 14) = S_7 \cup \{14, 19, 20\}$
 $S_9: S(\text{stocks}, 15) = S_8 \cup \{15\}$
 $S_{10}: S(\text{barrels}, 15) = S_8$
 $S_{11}: S(\text{locks}, 16) = S_8$
 $S_{12}: S(\text{totalLocks}, 16) = S_4 \cup S_{11} \cup \{16\}$
 $S_{13}: S(\text{stocks}, 17) = S_9 = \{13, 14, 19, 20\}$

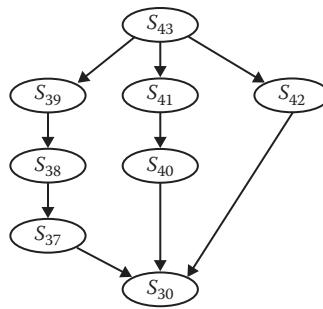


Figure 9.8 Partial lattice of slices on commission.

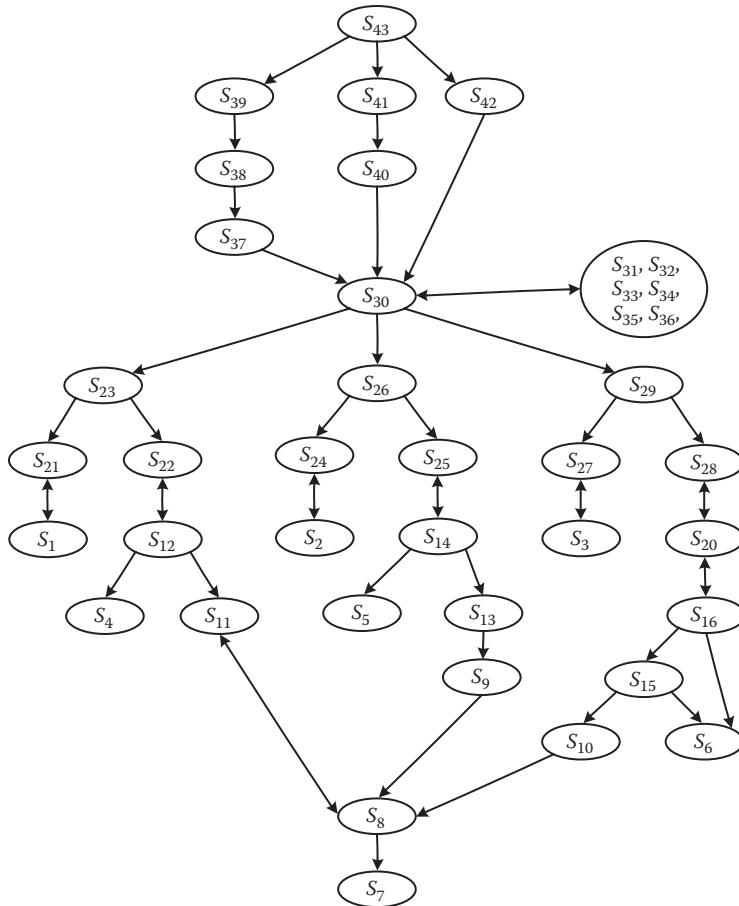


Figure 9.9 Full lattice on commission.

$$S_{14}: S(\text{totalStocks}, 17) = S_5 \cup S_{13} \cup \{17\}$$

$$S_{15}: S(\text{barrels}, 18) = S_6 \cup S_{10}$$

$$S_{16}: S(\text{totalBarrels}, 18) = S_6 \cup S_{15} \cup \{18\}$$

$$S_{18}: S(\text{totalLocks}, 21) = S_{12}$$

$$\begin{aligned}
S_{19}: S(\text{totalStocks}, 22) &= S_{14} \\
S_{20}: S(\text{totalBarrels}, 23) &= S_{16} \\
S_{21}: S(\text{lockPrice}, 24) &= S_1 \\
S_{22}: S(\text{totalLocks}, 24) &= S_{12} \\
S_{23}: S(\text{lockSales}, 24) &= S_{21} \cup S_{22} \cup \{24\} \\
S_{24}: S(\text{stockPrice}, 25) &= S_2 \\
S_{25}: S(\text{totalStocks}, 25) &= S_{14} \\
S_{26}: S(\text{stockSales}, 25) &= S_{24} \cup S_{25} \cup \{25\} \\
S_{27}: S(\text{barrelPrice}, 26) &= S_3 \\
S_{28}: S(\text{totalBarrels}, 26) &= S_{20} \\
S_{29}: S(\text{barrelSales}, 26) &= S_{27} \cup S_{28} \cup \{26\} \\
S_{30}: S(\text{sales}, 27) &= S_{23} \cup S_{26} \cup S_{29} \cup \{27\} \\
S_{31}: S(\text{sales}, 28) &= S_{30} \\
S_{32}: S(\text{sales}, 29) &= S_{30} \\
S_{33}: S(\text{sales}, 33) &= S_{30} \\
S_{34}: S(\text{sales}, 34) &= S_{30} \\
S_{35}: S(\text{sales}, 37) &= S_{30} \\
S_{36}: S(\text{sales}, 39) &= S_{30} \\
S_{37}: S(\text{commission}, 31) &= S_{30} \cup \{29, 30, 31\} \\
S_{38}: S(\text{commission}, 32) &= S_{37} \cup \{32\} \\
S_{39}: S(\text{commission}, 33) &= S_{38} \cup \{33\} \\
S_{40}: S(\text{commission}, 36) &= S_{30} \cup \{29, 34, 35, 36\} \\
S_{41}: S(\text{commission}, 37) &= S_{40} \cup \{37\} \\
S_{42}: S(\text{commission}, 39) &= S_{30} \cup \{29, 34, 38, 39\} \\
S_{43}: S(\text{commission}, 41) &= S_{39} \cup S_{41} \cup S_{42}
\end{aligned}$$

Several of the connections in Figure 9.9 are double-headed arrows indicating set equivalence. (Recall from Chapter 3 that if $A \subseteq B$ and $B \subseteq A$, then $A = B$.) We can clean up Figure 9.9 by removing these, and thereby get a better lattice. The result of doing this is in Figure 9.10.

9.2.2 Style and Technique

When we analyze a program in terms of interesting slices, we can focus on parts of interest while disregarding unrelated parts. We could not do this with du-paths—they are sequences that include statements and variables that may not be of interest. Before discussing some analytic techniques, we will first look at “good style.” We could have built these stylistic precepts into the definitions, but then the definitions are more restrictive than necessary.

1. Never make a slice $S(V, n)$ for which variables v of V do not appear in statement fragment n . This possibility is permitted by the definition of a slice, but it is bad practice. As an example, suppose we defined a slice on the locks variable at node 27. Defining such slices necessitates tracking the values of all variables at all points in the program.
2. Make slices on one variable. The set V in slice $S(V, n)$ can contain several variables, and sometimes such slices are useful. The slice $S(V, 27)$ where

$$V = \{\text{lockSales}, \text{stockSales}, \text{barrelSales}\}$$

contains all the elements of the slice $S_{30}: S(\text{sales}, 27)$ except statement 27.

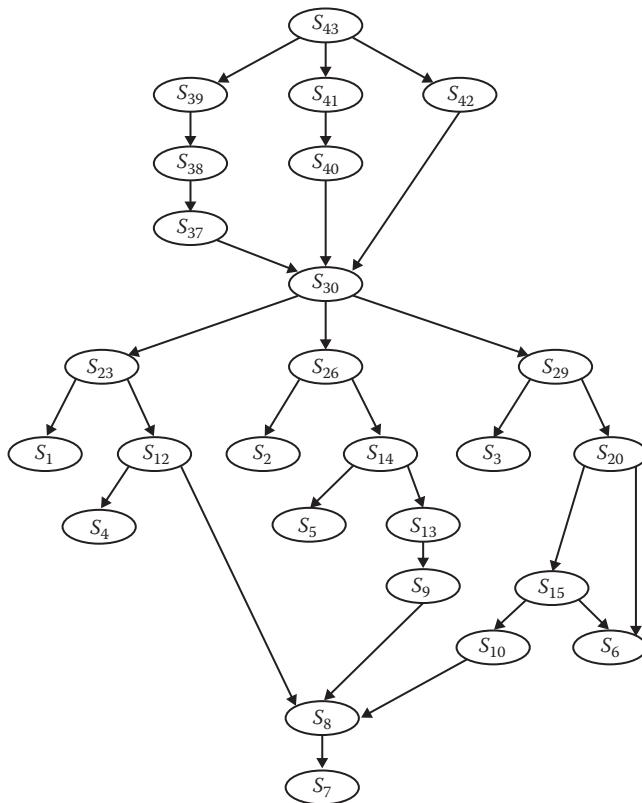


Figure 9.10 Simplified lattice on commission.

3. Make slices for all A-def nodes. When a variable is computed by an assignment statement, a slice on the variable at that statement will include (portions of) all du-paths of the variables used in the computation. Slice S_{30} : $S(\text{sales}, 27)$ is a good example of an A-def slice. Similarly for variables defined by input statements (I-def nodes), such as S_{10} : $S(\text{barrels}, 15)$.
4. There is not much reason to make slices on variables that occur in output statements. Slices on O-use variables can always be expressed as unions of slices on all the A-defs (and I-defs) of the O-use variable.
5. Make slices for P-use nodes. When a variable is used in a predicate, the slice on that variable at the decision statement shows how the predicate variable got its value. This is very useful in decision-intensive programs such as the triangle program and NextDate.
6. Consider making slices compilable. Nothing in the definition of a slice requires that the set of statements is compilable; however, if we make this choice, it means that a set of compiler directive and data declaration statements is a subset of every slice. If we added this same set of statements to all the slices we made for the commission program, our lattices remain undisturbed; however, each slice is separately compilable (and therefore executable). In Chapter 1, we suggested that good testing practices lead to better programming practices. Here, we have a good example. Think about developing programs in terms of compilable slices. If we did this, we could code a slice and immediately test it. We can then code and test other slices and merge them (sometimes called “slice splicing”) into a fairly solid program. This is done in Section 9.2.3.

9.2.3 Slice Splicing

The commission program is deliberately small, yet it suffices to illustrate the idea of “slice splicing.” In Figures 9.11 through 9.14, the commission program is split into four slices. Statement fragment numbers and the program graphs are as they were in Figure 9.1. Slice 1 contains the input while loop controlled by the locks variable. This is a good starting point because both Slice 2 and Slice 3 use the loop to get input values for stocks and barrels, respectively. Slices 1, 2, and 3 each culminate in a value of sales, which is the starting point for Slice 4, which computes the commission bases on the value of sales.

This is overkill for this small example; however, the idea extends perfectly to larger programs. It also illustrates the basis for program comprehension needed in software maintenance. Slices allow the maintenance programmer to focus on the issues at hand and avoid the extraneous information that would be in du-paths.

```

1 Program Slice1 (INPUT,OUTPUT)
2 Dim locks As Integer
3 Dim lockPrice As Real
4 Dim totalLocks As Integer
5 Dim lockSales As Real
6 Dim sales As Real
7 lockPrice = 45.0
13 Input(locks)
'locks = -1 signals end of data
14 While NOT(locks = -1)
16 totalLocks = totalLocks + locks
19 Input(locks)
20 EndWhile
21 Output("Locks sold: ", totalLocks)
24 lockSales = lockPrice*totalLocks
27 sales = lockSales
28 Output("Total sales: ", sales)

```

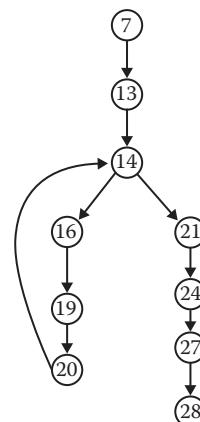


Figure 9.11 Slice 1.

```

1 Program Slice2 (INPUT,OUTPUT)
2 Dim locks, stocks As Integer
3 Dim stockPrice As Real
4 Dim totalStocks As Integer
5 Dim stockSales As Real
6 Dim sales As Real
8 stockPrice = 30.0
11 totalStocks = 0
13 Input(locks)
'locks = -1 signals end of data
14 While NOT(locks = -1)
15 Input(stocks)
17 totalStocks = totalStocks + stocks
19 Input(locks)
20 EndWhile
22 Output("Stocks sold: ", totalStocks)
25 stockSales = stockPrice*totalStocks
27 sales = stockSales
28 Output("Total sales: ", sales)

```

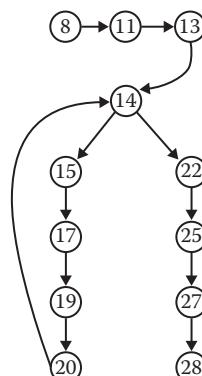


Figure 9.12 Slice 2.

```

1 Program Slice3 (INPUT,OUTPUT)
2 Dim locks, barrels As integer
3 Dim barrelPrice As Real
4 Dim totalBarrels As Integer
5 Dim barrelSales As Real
6 Dim sales As Real
9 barrelPrice = 25.0
12 totalBarrels = 0
13 Input(locks)
'locks = -1 signals end of data
14 While NOT(locks = -1)
15 Input(barrels)
18 totalBarrels = totalBarrels + barrels
19 Input(locks)
20 EndWhile
23 Output("Barrels sold;" totalBarrels)
26 barrelSales = barrelsPrice * totalBarrels
27 sales = barrelSales
28 Output("Total sales;" sales)

```

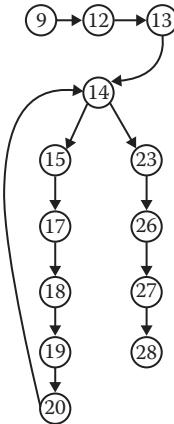


Figure 9.13 Slice 3.

```

1 Program Slice4 (INPUT,OUTPUT)
6 Dim sales, commission As Real
29 If (sales>1800.0)
30 Then
31 commission = 0.10 * 1000.0
32 commission = commission + 0.15*800.0
33 commission = commission + 0.20*(sales-1800.0)
34 Else If (sales > 1000.0)
35 Then
36 commission = 0.10 * 1000.0
37 commission = commission + 0.15*(sales-1000.0)
38 Else
39 commission = 0.10 * sales
40 EndIf
41 EndIf
42 Output("Commission is $" commission)
43 End Commission

```

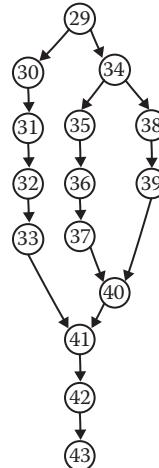


Figure 9.14 Slice 4.

9.3 Program Slicing Tools

Any reader who has gone carefully through the preceding section will agree that program slicing is not a viable manual approach. I hesitate assigning a slicing exercise to my university students because the actual learning is only marginal in terms of the time spent with good tools; however, program slicing has its place. There are a few program slicing tools; most are academic or experimental, but there are a very few commercial tools. (See Hoffner [1995] for a dated comparison.)

The more elaborate tools feature interprocedural slicing, something clearly useful for large systems. Much of the market uses program slicing to improve the program comprehension that maintenance programmers need. One, JSlice, will be appropriate for object-oriented software. Table 9.6 summarizes a few program slicing tools.

Table 9.6 Selected Program Slicing Tools

<i>Tool/Product</i>	<i>Language</i>	<i>Static/Dynamic?</i>
Kamkar	Pascal	Dynamic
Spyder	ANSI C	Dynamic
Unravel	ANSI C	Static
CodeSonar®	C, C++	Static
Indus/Kaveri	Java	Static
JSlice	Java	Dynamic
SeeSlice	C	Dynamic

EXERCISES

1. Think about the static versus dynamic ambiguity of du-paths in terms of DD-paths. As a start, what DD-paths are found in the du-paths p12, p13, and p14 for sales?
2. Try to merge some of the DD-path-based test coverage metrics into the Rapps–Weyuker hierarchy shown in Figure 9.5.
3. List the du-paths for the commission variable.
4. Our discussion of slices in this chapter has actually been about “backward slices” in the sense that we are always concerned with parts of a program that contribute to the value of a variable at a certain point in the program. We could also consider “forward slices” that refer to parts of the program where the variable is used. Compare and contrast forward slices with du-paths.

References

- Bieman, J.M. and Ott, L.M., Measuring functional cohesion, *IEEE Transactions on Software Engineering*, Vol. SE-20, No. 8, August 1994, pp. 644–657.
- Ball, T. and Eick, S.G., Visualizing program slices, *Proceedings of the 1994 IEEE Symposium on Visual Languages*, St. Louis, MO, October 1994, pp. 288–295.
- Clarke, L.A. et al., A formal evaluation of dataflow path selection criteria, *IEEE Transactions on Software Engineering*, Vol. SE-15, No. 11, November 1989, pp. 1318–1332.
- Gallagher, K.B. and Lyle, J.R., Using program slicing in software maintenance, *IEEE Transactions on Software Engineering*, Vol. SE-17, No. 8, August 1991, pp. 751–761.
- Hoffner, T., *Evaluation and Comparison of Program Slicing Tools*, Technical Report, Dept. of Computer and Information Science, Linkoping University, Sweden, 1995.
- Rapps, S. and Weyuker, E.J., Selecting software test data using dataflow information, *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, April 1985, pp. 367–375.
- Weiser, M., *Program Slices: Formal Psychological and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, Ann Arbor, MI. 1979.
- Weiser, M.D., Program slicing, *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, April 1988, pp. 352–357.

CHAPTER 6

SOFTWARE VERIFICATION AND VALIDATION



OBJECTIVES

This chapter aims to give an understanding of verification and validation and the different stages of software development under these two domains.



6.1 INTRODUCTION

We have seen two views of quality, viz. 'Conformance to requirements' which is defined as a developers view of quality and 'Fitness for use' which is defined as a customers view of quality. The first part may be termed 'verification' of the process used for developing a product while the second part

may be called 'validation' of a product which is created by testers and given to customers for acceptance testing/using it in production.

Quality planning includes planning for the processes used to build and test software to create a right product. It may include software certification which may be termed 'validation', as well as finding conformance to the requirements defined by the processes and standards called 'verification'. Quality is not an accident but an outcome of planned activities carried out in various phases of Software Development Life Cycle (SDLC).

Quality planning in an organisation includes the following stages.

- Definition of processes, procedures, standards, and guidelines for developing and testing software. These may be refined and redefined as defects are found in the application, so that processes become more and more matured.
- Performing quality operations such as verification and validation as per process definition at different phases of SDLC.
- Auditing work products to ensure that processes are followed correctly and work products match exit criteria for the phase under consideration.
- Collecting metrics and defining actions for lacunae found in processes and opportunities for improvements on the basis of quantitative data.

Verification and validation activities are two branches of software testing. They are complementary to each other, and not substitutes of each other. Verification and validation are completely dependent on each other. They are not even mutually exclusive. In addition to auditing, which works on sampling basis, verification and validation ensure that processes are followed correctly, the process definition is also appropriate and process capability is ensured by optimising processes through feedback loop. Moreover, the work product also matches with exit criteria as defined by various stages of development. Many organisational metrics related to processes as well as product are defined by collecting data through verification and validation. Proper verification and validation together can ensure quality of work product as well as quality of processes used for developing and testing software which will finally help in achieving business objectives. Only proper verification without proper validation, and vice versa cannot ensure a good software product. Rather, it needs a combination of both.

6.2 VERIFICATION

Verification is a disciplined approach to evaluate whether a software product fulfills the requirements or conditions imposed on them by the standards or processes. It is done to ensure that the processes and procedures defined by the customer and/or organisation for development and testing are followed correctly.

Verification is also called 'static technique' as it does not involve execution of any code, program or work product. It is done by systematically reading and assessing the contents of work product with an intention of detecting defect with respect to standards or processes. It helps in identification of defect and its location which then assists in correction. Some people follow a methodology of finding and fixing verification defects then and there only. One must do a root cause analysis and take corrective as well as preventive actions on the defects found.

6.2.1 ADVANTAGES OF VERIFICATION

- Verification can confirm that the work product has followed the processes correctly as defined by an organisation or customer (as the case may be). There is higher probability of success due to following processes religiously.
- It can find defects—in terms of deviations from standards—easily and also, the location of the defects. This helps in fixing the defects easily as well as conducting root cause analysis so that the same defects are not repeated.
- It can reduce the cost of finding and fixing defects as each work product is reviewed and corrected faster. Sometimes, defects are fixed as and when they are found.
- It can locate the defect easily as the work product under review is yet to be integrated with other work products. Cost of fixing defect is less as there is no/minimum impact on other parts of software.
- It can be used effectively for training people about processes and standards. Typically peer review, superior reviews, and walkthrough can be excellent tools for training.

6.2.2 DISADVANTAGES OF VERIFICATION

- It cannot show whether the developed software is correct or not. Rather, it shows whether the processes have been followed or not. If processes are not capable, then the outcome may not be good.

- Actual working software may not be assessed by verification as it does not cover any kind of execution of a work product. 'Fit for use' cannot be assessed in verification.

6.2.3 PREREQUISITES FOR VERIFICATION

Depending upon the method followed and the work product under verification, there may be different inputs required for verification. Figure 6.1 is an indicative workbench definition for verification. It may differ significantly from one work product to another, for different customers and organisations.

For any process to happen, there must be some inputs coming to the work bench from the previous work bench, and some outputs going out from the work bench to the next work bench. Inputs for the verification process may include work product, quality plan or verification plan, and output from the verification work bench may include verification report, and defects. Some tools, checklists, guidelines, process definitions of verification, and standards are required for successful verification activity.

Prerequisites for verification may include the following

- Training required by people for conducting verification
- Standards, guidelines, and tools to be used during verification
- Verification Do and Check process definition

6.3 VERIFICATION WORK BENCH

A verification work bench is where verification activities are conducted, and may be a physical or virtual entity. For every work bench, the following basic things are required.

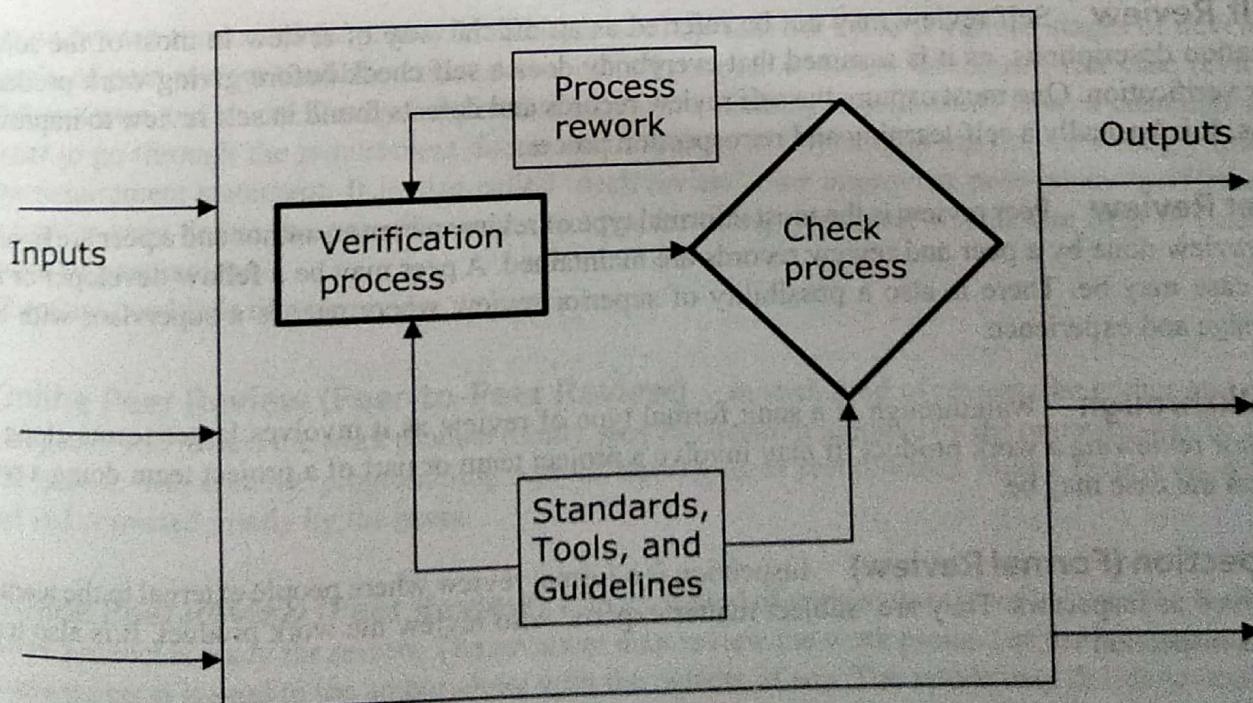


Fig. 6.1

Verification workbench

Inputs There must be some entry criteria definition when inputs are entering the work bench. This definition should match with the output criteria of the earlier work bench. For example, if we are verifying a code file, then we must have a written and compilable code as an input to the work bench along with verification plan, verification artifacts, etc.

Outputs Similarly to the entry criteria, there must be some exit criteria from work bench which should match with input criteria for the next work bench. Outputs may include review comments and the work product with defects if any.

Verification Process Verification process must describe step-by-step activities to be conducted in a work bench. It must also describe the activities done while verifying the work product under review.

Check Process Check process must describe how the verification process has been checked. It is not a verification of the work product but a verification of the process used for verification. Quality plan must define the objectives to be achieved, and check processes must verify that the objectives have been really achieved.

Standards, Tools, and Guidelines These may be termed 'the tools' available for conducting verification activities. There may be coding guidelines or testing standards available for verifications. Sometimes checklists are used for doing verification.

6.4 METHODS OF VERIFICATION

There are many methods available for verification of software work products. Some of them are listed below.

- **Self Review** Self review may not be referred as an official way of review in most of the software verification descriptions, as it is assumed that everybody does a self check before giving work product for further verification. One must capture the self review records and defects found in self review to improve the process. It is basically a self-learning and retrospection process.
- **Peer Review** Peer review is the most informal type of review where an author and a peer are involved. It is a review done by a peer and review records are maintained. A peer may be a fellow developer or tester as the case may be. There is also a possibility of superior review where peer is a supervisor with better knowledge and experience.
- **Walkthrough** Walkthrough is a semi formal type of review as it involves larger teams along with the author reviewing a work product. It may involve a project team or part of a project team doing a review jointly as the case may be.
- **Inspection (Formal Review)** Inspection is a formal review where people external to the team may be involved as inspectors. They are 'subject matter experts' who review the work product. It is also termed 'Fagan's inspection'.
- **Audits** Audit is a formal review based on samples. Audits are conducted by auditors who may or may not be experts in the given work product.

6.4.1 SELF REVIEW

Self review is not considered as an official review in software development as it is expected to be done by the author or creator of an artifact by default. It is a primary expectation that no one must say that his work is done before doing a self review and being satisfied that there are no problems in the output. Self review is excellent in defect prevention through self learning or retrospection. Defects found in self review can help in self education and self improvement.

Advantages of Self Review

- Self reviews are highly flexible with respect to time and defect finding, as one need not take an appointment or allocate specific time while doing it. It may be an online activity.
- Self review is an excellent tool for self learning, as learning from mistakes is a good way of improvement.
- There is no ego involved in self review as the findings need not be shared with anybody.

Disadvantages of Self Review

- Approach- or understanding-related defects may not be found in self review as it is difficult to think in some other way.
- People involved in self review may not conduct a review in reality due to time and focus issues. For example, developers may concentrate more on creating code rather than reviewing the code developed by them.
- Something which has been implemented correctly in initial development may get changed due to personal issues. Some people with less self confidence may change the correct implementation.

6.4.2 PEER REVIEW

Peer reviews are conducted frequently in software development life cycle at various stages of development. Example of peer review could be code review done by a peer or fellow developer, test case review done by fellow tester and so on. It may happen that a person collecting requirements from a customer may ask his peer to go through the requirement document to find out if anything is missing or wrongly interpreted in the requirement statement. It is also called 'desk review'. For improving peer-review performance, an organisation may define the processes and checklist for doing peer review. This can help in tracking the outcome of peer reviews.

There are two kinds of peer review

• **Online Peer Review (Peer-to-Peer Review)** In such kind of review, the author and reviewer meet together and review the work product jointly. Any explanation required by the reviewer may be provided by the author. The extreme programming way of development recommends online review as defects are found and corrected jointly by the peers.

• **Offline Peer Review (Peer Review)** In such kind of review, the author informs the reviewer that the work product is ready for review. The reviewer may review the work product as per his time availability. The review report is send to the author along with the defects, if any. The author may decide to accept/reject the review comments or defects identified. This review is prone to mistakes as there is no joint review, and the author and the reviewer do not discuss the defects. It is more on an opinion basis.

Advantages of Peer Review

- Peer reviews are highly informal and unplanned in nature. It can happen at any time during development/testing life cycle. Sometimes, peer review is done jointly by the author and the peer, while at other times, the peer may complete the review of the artifact on his own and give feedback to the author.
- There are no (or less) egos or pride attached between the peers, and defects are accepted easily. Peer-to-peer relationships are important in such reviews.
- Mostly, defects are discussed and decision is reached very informally. This helps in finding and fixing the defects fast.
- Peer review is an excellent tool for educating peers about the artifact. While reviewing the artifacts, a reviewer must have some basic knowledge about the contents and background information. Thus, more number of people can be aware about the contents and approach followed in each artifact in addition to an author.

Disadvantages of Peer Review

- The other person doing the review may or may not be an expert on the artifacts under review, and his/her suggestions may not be valid always. Sometimes, a peer may change the work product without informing the author, or without noting down the defects in review reports. This may happen in offline reviews where a peer may make changes without telling the author.
- Peers may fix the defects without recording them, as they may not like to show defects of their partners/friends. Root causes of the defects may not be analysed as defect fixes may not be known.
- Sometimes, the correct implementation may get replaced by the wrong one as reviewer may update the artifact without recording a defect. There is no discussion, and hence, updation may go unnoticed.
- Approach-related defects may not be fixed, if both the author and the reviewer are at the same status and understand things in a similar way.

6.4.3 SUPERIOR REVIEW

Sometimes, superior review is also considered as peer review, as a superior may be considered as peer of the author with some more experience and visibility. Superior reviews are conducted to avoid some limitations of peer review related to approach, visibility, etc. A superior such as team leader or module leader may have to review the artifacts made by his subordinates to find its suitability with reference to expected outcome.

Advantages of Superior Review

- Approach-related defects can be found easily as superiors are expected to have a better view of the entire project. A superior may have better understanding of overall requirements and designs than the author.
- Superiors can share better ways of doing things—learned from their experience—with their subordinates. Thus, superior review is also used as a tool for training.

Disadvantages of Superior Review

- Superior review can become a bottleneck if everything developed by a team needs to be reviewed by the superior. For example, if there are 10 people in a team of a module leader, then reviewing their work product may hamper his own work.
- It does not work efficiently when the superior is also new to the domain/approach, or if the project is in research and development kind of environment where exact solution or even approach may not be known to anybody.

6.4.4 WALKTHROUGH

Walkthrough is more formal than peer review but less formal than inspection. It is termed 'semi formal review' as only related people are involved in walkthrough. In a typical walkthrough, some members of a project team are involved in examining an artifact under review. Author of an artifact presents it, and the entire team discusses various aspects of the artifact. The comments/minutes of the discussion are documented and actions are tracked till closure. Walkthroughs can be effectively used for communication between the team.

Advantages of Walkthrough

- Walkthrough is an excellent tool for collaboration where joint decisions can be made by the team. Each and every member present must be involved in making decisions.
- Walkthrough is also useful for training the entire team at one time. Test scenarios and designs are generally subjected to walkthrough.
- People understand what they can expect from the document/approach under consideration.
- Problems can get recorded and suggestions can be received from the team for improving the work product further.

Disadvantages of Walkthrough

- Availability of people can be an issue when teams are large. Multilocations and distributed teams are major challenges in walkthrough. It is more formal than peer and procuring people can be a challenge.
- Team members may not be experts in giving comments and may need some training and basic knowledge about the project, artifact under walkthrough and so on.
- Time can be a constraint where schedules are very tight.

6.4.5 INSPECTION

Inspection is a very formal way of reviewing the work product. Agenda of an inspection is decided in advance. Moreover, people attending the inspection are given inputs and background documents beforehand. Generally, the author of the work product is not allowed/expected to be present during inspection. A presenter presents a work product, and a recorder makes notes of comments given by the inspectors. The actions identified during inspection are tracked till closure. The people present in inspection are experts in the domain under consideration. Sometimes, inspectors may be external to the organisation. These people are experts in their fields but may not understand customer requirements exactly. There can be issues like non-disclosure agreement which may affect inspection by outsiders.

Advantages of Inspection

- Experts opinion is available as they are present during review. Subject-matter experts can clarify many issues.
- Inspection must be time effective as availability of experts is limited. There is always a time pressure for inspection.
- Defects are recorded but solutions are not given by subject-matter experts. This helps the organisation to initiate own action plan for fixing the defects.

Disadvantages of Inspection

- Experts may not be ready with review comments before inspection and time may not be utilised properly. It is a common scenario that people start reading documents at the time of meeting.

- Experts' opinion may vary from realities as it may be derived from their judgments and experiences. Experts may not understand exactly what the needs of the customer are.
- Facilitator's job becomes critical in case there is a difference of opinion between two inspectors, as the final aim of the organisation/project is to get inputs from inspectors.

Guidelines for Successful Inspection Inspection is a very formal method of verifying software work products with respect to expectation, standards, and guidelines. Sometimes, the customer may participate in inspection process. A defined methodology of inspection must be followed for performing inspection. The processes for inspection must be defined in organisational process database. Some generic guidelines for inspection are given below.

- Organisation/project must plan for inspection well in advance so that all the stakeholders are aware of their roles, responsibilities, time schedule available, etc. There must be a process definition of how to do inspection. It becomes more important when inspectors are subject-matter experts from outside the organisation including the customer. Effectiveness of the inspection process must be compared with expected level of expectations, and deviations must be taken for process improvement.
- Allocation of trained resources for facilitating inspection process, recording the opinions of experts, and presenting artifacts are essential to maintain the agenda and inspection flow. Inspectors, typically outsiders, are expected to be subject-matter experts in their field. They carry knowledge—and also, prejudice and ego with them. If two inspectors have a conflict of opinion, their opinions need to be handled carefully. Wastage of time can prove to be very costly.
- Expertise in terms of logistics arrangement is also needed, if inspectors are external to the organisation. There must be venue bookings and other arrangements for the inspection.
- Moderator/leader must circulate the work product to be inspected to the inspection team in advance, so that inspectors come prepared and maximum can be achieved in the given time. Inspector cost is very high, and output has to be measured against these costs.
- Product must be clear, compiled, checked for grammar, formatting, etc. Inspectors are expected to give inputs about approach of doing things and contents, and not on language or grammar.
- Inspection team may be of 3–5 people, as it is difficult to handle a larger group. Less number of inspectors would mean bias in the observations.
- Maximum time allocated for the inspection must not exceed 1–1.5 hours, as people may not be able to concentrate for a very long time. It works as per the law of diminishing returns.
- It is the moderator's responsibility to maintain the agenda of inspection. It is expected that inspection must concentrate on the work product and not a person/author of the work product.
- Manager or superior of the author may not be involved in the inspection as poor remarks about a work product, if any, can create a bad impression of the author on the superior. It may affect the morale of the author in a negative manner. People may not like to receive criticism of their work products in front of their superiors.
- The presenter presents the work product while inspectors give their comments. Many organisations do not allow the author to be present during inspection as they may become defensive while facing criticism about their work product.
- During inspection, one must concentrate on locating defects and not on fixing them. How to fix these defects is a responsibility of an organisation/project/author. One can note down what is wrong but need not ask any explanation to the inspector about how to fix it.
- Classify and record defects in different categories. This can help when root causes of the defect are analysed and taken for further action. It can show the weak parts of the processes so that efforts can be concentrated in the defined areas.

- Review inspection process for continuous improvement. Stronger parts may be noted, and weaker parts may be taken for improvements to make a process successful.

Phases of Inspection

A typical inspection has the following phases

- *Planning for Inspection* Planning for inspection involves selecting people for inspection, allocating roles to other people (such as facilitator, recorder, and presenter); defining the entry and exit criteria for inspection, and selecting which parts of artifacts are to be looked at. The artifact must be delivered to concerned people well in advance so that they are ready with their comments.
- *Kick-off Inspection* Kick-off inspection may start by distributing artifacts, explaining the objectives of inspection, process to be followed for inspection, and checking entry criteria for the artifact as well as inspection process. Entire process must be known to people taking part in actual inspection and to the stakeholders of inspection.
- *Individual Preparation* It is expected that participants must come prepared for the inspection. Work must be reviewed and comments (including potential defects and questions) by each of the participants must be ready before the inspection meeting. Participants must have read the work product and have clarity about the problems and issues.
- *Inspection Meeting* Proceeds of the meeting such as discussions done or logging of comments, and preparation of minutes of meeting must be completed in defined time frame. The participants of the meeting may simply note defects, make recommendations for handling the defects or make decisions about the defects. Recorder shall note these comments.
- *Decision on Comments* An organisation has to decide the fate of comments once the inspection is over. It is not necessary that all comments will be accepted. Comments may be rejected, deferred or may undergo another iteration of inspection. For accepted comments, the fixing of defects is done by the author.
- *Follow Up* Completing the actions as identified in minutes of meeting like checking that the defects have been addressed and whether exit criteria has been met or not is required. The findings can be used to gather statistics about work product, project or process.

Roles and Responsibilities

A typical inspection may include the following roles.

- *Manager* Manager is the person responsible for getting the work product inspected. For a project, he may be the project manager. Manager decides on the execution of inspection, defines the schedule, allocates time, defines the objectives to be met, and determines if the inspection objectives have been met or not at the end of the inspection process.
- *Moderator* Moderator is the person who leads the inspection of the artifacts, including planning the inspection, running the meeting and follow-up after the meeting. If necessary, moderator may mediate between the various points of view of different inspectors, and is often the person upon whom the success of the inspection rests. Moderator is also called 'facilitator' as he facilitates the entire process.
- *Author* Author is the writer or person with chief responsibility of the artifact to be inspected. He is the person who has created the artifact, and will be taking action based on the outcome of the inspection.
- *Reviewers (Checkers/Inspectors)* Individuals with specific technical or business background who, after necessary preparation, identify and describe findings in the work product under inspection in the form of comments. Reviewers must be chosen to represent different perspectives and roles in the inspection process and must take part in meetings.

- **Recorder** Recorder is the person who documents all the issues, problems and open points that are identified during the meeting. The presence of a recorder is a must because if reviewers or inspectors are discussing something, they have to decide collectively about the comment. Recorder is also expected to write minutes of meeting.

Success Factors for Inspection

- Each inspection must have a clear, predefined objective that has to be achieved. An inspection can be called 'successful' only if these objectives are met.
- The right people must be involved in the inspection process. For success in an inspection, people with adequate knowledge, expertise, etc. must be involved.
- Defects found should be welcomed and expressed objectively. Inspection must tell clearly what is wrong and why it is wrong.
- People issues and psychological aspects are not to be dealt with in an inspection process. One must stick to the artifact only (e.g. making it a positive experience for the author).
- Inspection techniques applied should be suitable to the type and level of software work products and inspectors.
- Checklist or role playing is to be used if appropriate to increase effectiveness of defect identification.
- Training in inspection techniques is to be given in case inspectors are internal to the organisation.
- Management support is essential for a good inspection process (e.g. by incorporating adequate time for inspection activities in project schedules).
- There must be an emphasis on learning lessons and process improvement.

6.4.6 AUDIT

Audit is defined as an independent assessment of the process or product under progress. Historically, audit process was applied to financial transactions to know and understand the financial health of an organisation. Audit may be categorised into quality assurance activity or quality control activity depending upon its nature, breadth and depth. 100% auditing may be considered as quality control activity while sampling may be considered as quality assurance activity. Some people define auditing as a separate branch than quality control or quality assurance, as it involves combination of quality assurance activities and quality control activities at a time.

Software development and testing process audit is an examination of product to ensure that the products as well as processes used to build them meet predefined criteria. Audits are conducted by auditors—people with expertise in auditing processes and also some knowledge about work products (CMMi organisations may refer to such people as Software Quality Analysts (SQA)). Auditors may not be experts in subject matter but may have a fair amount of experience in the field as well as in auditing techniques. The outcome of audit is an audit report comprising the following.

- **Major and Minor Non-Conformances** Non-conformances are deviations from the established criteria used for developing a work product. They indicate a gap between what is required and what actually exists. Minor non-conformances indicate that there is some deviation at some places, but in other places, the processes have been followed correctly. Major non-conformances indicate a big issue where something required is missing completely.
- **Observations** Observations are findings which may get converted into future non-conformances, if proper care is not taken. They may be termed as conformances just on the verge of breakage.

- **Achievement or Good Practices** These are good achievements by areas under audit which can be used by others. There is no possibility of existence of any project where there is not a single positive aspect which can be shared across the platform.

In software development and testing phases, various audits are conducted, as mentioned below,

- **Kick-Off Audit** Kick-off audit is conducted at the start of the project. This audit covers the areas that ensure whether all the processes required at the start of the project are covered or not. It may start from proposal, contract, risk analysis, scope definition, team size and skill requirements, authorities and responsibilities of various roles in the team. Kick-off audits cover checking documentation and compliances required at the start of the project.
- **Periodic Software Quality Assurance Audit** Software quality assurance audit is famous by the name 'SQA audit' or 'SQA review'. It is conducted for a product under development and processes used as defined by quality assurance process definition for these work products. Auditor is expected to check whether different work products meet the defined exit criteria or not, and the processes used for building these work products are correctly implemented or not.
- **Phase-End Audit** Phase-end audit checks whether the phase defined in the development life cycle achieves its expected outcome or not. It is also used as a gate to decide whether the next phase can be started or not. These are also termed 'gate reviews'.
- **Predelivery Audit** Predelivery audit checks whether all the requisite processes of delivery are followed or not, and whether the work product meets the expected delivery criteria or not. Many organisations follow a delivery checklist approach for conducting predelivery audits. Only those work products which are successful in predelivery audits can be given to a customer.
- **Product Audit** Product audits can be covered in SQA audit. It is done by executing sample test cases to find whether the product meets its defined exit criteria or not. Sometimes, this is also considered as 'smoke testing'.

6.5 TYPES OF REVIEW ON THE BASIS OF STAGE/PHASE

The reviews may be categorised on the basis of their happening at particular phase or timeline during development life cycle, as given below.

6.5.1 IN-PROCESS REVIEW

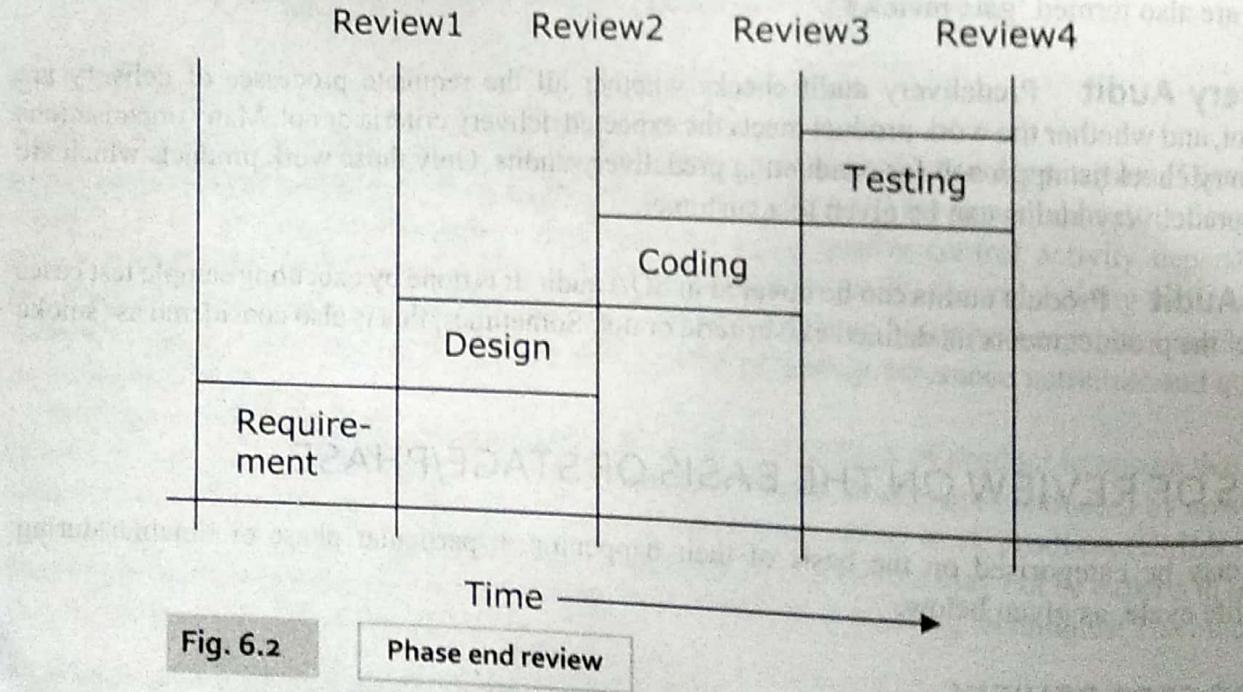
Reviews conducted while different phases of software development life cycle are going on are defined as 'in-process review'. They are intended to check whether inputs to the phase are correct or not, and whether all the applicable processes/procedures during a phase are followed correctly or not. Process reviews help in correcting things before the phase ends, so that the objectives of the phase can be achieved.

Milestone Review Milestone reviews are conducted on periodic basis depending on the completion of a particular phase or a time frame defined or a milestone achieved during a software development life cycle. Examples of such reviews may be a requirement review at the end of requirement phase, design

review at the end of design phase, weekly status review at the end of a week, or review conducted as per percentage completion (say 5% project completion review). These reviews confirm that the output from the phase matches with predefined quality goals and input requirements of the next phase, and development life cycle can go to the next phase. Three types of milestone reviews are explained below.

- **Phase-End Review** Phase-end review is conducted at the end of a development phase under review such as requirements phase, design phase, coding, and testing. It can be applied when there are distinct phases of development as defined in waterfall development model such as requirements gathering, design creation (high level and low level designs), coding, testing at various levels, deployment, etc. and also, input and output criteria for the phases are clearly defined. Typical waterfall cycle is suitable for doing a phase-end review where distinction between different phases is clearly defined. It may be used during iterative development where one phase completion may be reviewed. As required by iterative development methodology, reviewed phase may undergo change when there are some changes coming in. Changes may be required due to some other phase of development.

Phase-end reviews are also termed 'gate reviews' by some organisations as the gate at phase end opens only when the output criteria of the phase are achieved by the work product. Phase-end reviews are used extensively in agile development as one phase completion is marked by review before the next phase is started. Figure 6.2 explains phase-end review in a typical waterfall development approach.



- **Periodic Review** Waterfall is a classical model and applied very rarely in software development. Most of the times, we may not have a clear definition of phase end as there may be considerable overlap between different phases of development. Also, there is no clear definition of input and output criteria for each phase but they get defined as one progresses through software development life cycle. When one phase ends, in reality another phase may be half way through. In such cases, we may have review on some periodic basis.

such as weekly, monthly, quarterly, etc. Project plan must define various periods when review of project-related activities will be conducted. Stakeholders may be required to review the progress and take actions on any impediments. Many models such as CMMi mandate that various activities must be reviewed with stakeholders and senior management. Figure 6.3 shows a monthly review for a project where there are considerable overlaps between different phases.

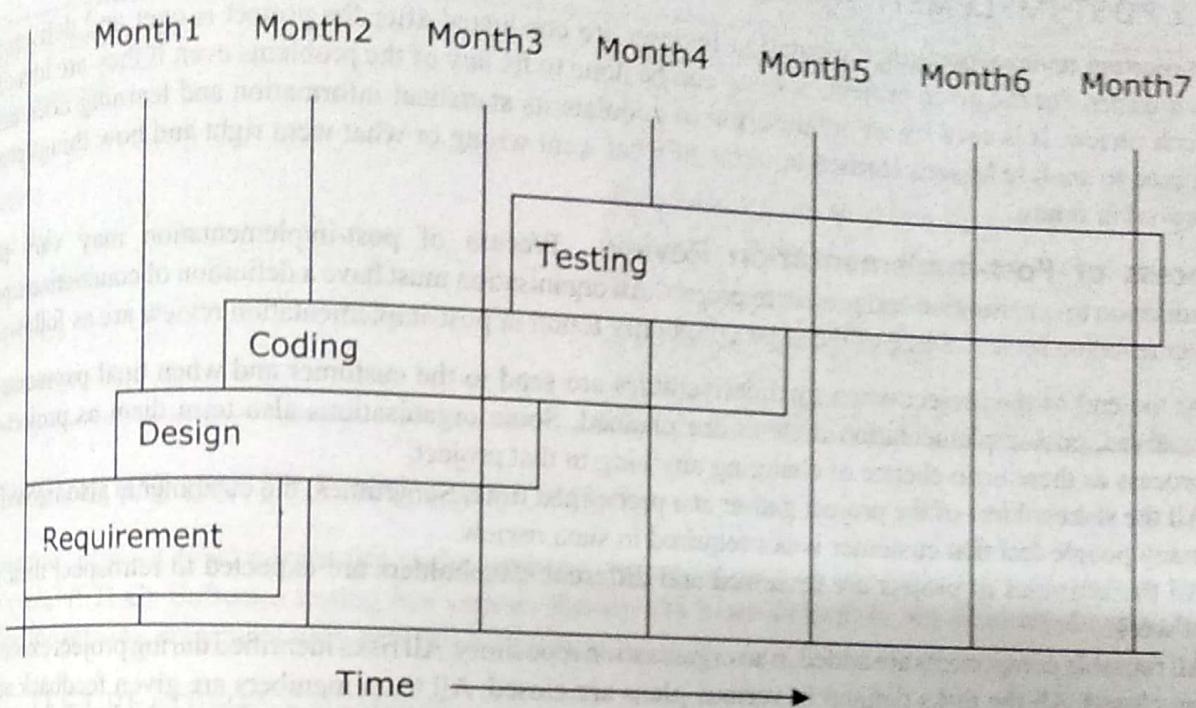


Fig. 6.3

Periodic review

Percent Completion Percent completion review is a combination of periodic review and phase-end review where the project activities or product development activities are assessed on the basis of percent completion e.g. 5% completion or similar way. The review is conducted as the project progresses and the part of the project defined in percentage gets completed. At high activity level, month may be a very long duration for review as lot of things may change in a month. In such cases, percentage review is a better option. There may be some phases where development/testing activities may be very low and reviewing every month is of not much value as there is not much change. Here also, percentage is a better way of doing review.

Process of In-Process Review An organisation/project must have a definition of in-process review. It must have a list of stakeholders attending various reviews, and the work products under reviews. Sometimes, the customer is also invited for such reviews. Commonly found steps of in-process reviews are given below.

- Work product, and metrics undergoing review are created and submitted to the stakeholders well in advance.
- Inputs are received from stakeholders to initiate various actions.
- Risks identified by the project are reviewed and actions may be initiated as required. Risks which no longer exist may be closed and risks which cannot affect the project are retired.

- Any issue related to any stakeholder is noted and follow-up actions are initiated. Issues may be like software availability, hardware availability, changing requirements, trainings required, etc.
- Review reports are generated at the end of review which acts as a foundation for the next review. In each review, actions and issues of earlier reviews are visited and any open items are taken for further actions.

6.5.2 POST-IMPLEMENTATION REVIEW

Post-mortem reviews/post-implementation reviews are conducted after the project is over and delivered to the customer. For the given project, nothing can be done to fix any of the problems even if they are identified in such review. It is used by an organisation to populate its statistical information and learning documents. It is used to analyse lessons learned in terms of what went wrong or what went right and how things can be improved in future.

Process of Post-Implementation Review Process of post-implementation may vary from organisation to organisation and project to project. An organisation must have a definition of conducting a post-implementation review. Some of the steps commonly found in post-implementation review are as follows.

- At the end of the project when final deliverables are sent to the customer and when final payments are received, post-implementation reviews are planned. Some organisations also term them as project-end process as there is no chance of changing anything in that project.
- All the stakeholders of the project gather at a predefined time. Sometimes, the customer is also invited but many people feel that customer is not required in such review.
- All the activities of project are reviewed and different stakeholders are expected to retrospect their part of work.
- All reusable components are added in an organisation repository. All risks identified during project execution are closed. All the tasks defined in various plans are closed. All team members are given feedback about their performance in a project. Skill sets of all team members are updated in an organisation database.
- Each and every activity is listed to identify if something went exceptionally good or exceptionally bad when the project was executed. All project-related metrics are reviewed.

6.6 EXAMPLES OF ENTITIES INVOLVED IN VERIFICATION

The following Table 6.1 illustrates the artifacts prepared during development and testing life cycle and the entities reviewing them. These are indicative entities and may change as per specific circumstances, organisation and customer requirements.

Table 6.1

Entities performing verification

Verification	Entities performing verification
Requirement review	Business analysts, System analysts, Project team including architects, developer and customer/User
Design review	Project team, Customer/user
Code review	Development team, Customer/user
Project plan review	Project team, Customer/user, Suppliers
Test artifacts review	Test team, Development team, Customer/user

All the verification techniques discussed above may not be used for all phases and work products. Verification techniques generally used in different phases of development lifecycle are shown in Table 6.2.

Table 6.2

SDLC phase and verification technique used

Phase	Verification technique used
Planning documents	Inspection, Walkthrough, Peer review
Requirements	Inspection, Peer review
Design	Walkthrough, Peer review
Coding	Peer review, Superior review
Test plan	Inspection, Walkthrough, Peer review
Test scenario	Walkthrough, Peer review
Test cases	Peer review, Superior review
Test results	Walkthrough, peer review

6.7 REVIEWS IN TESTING LIFECYCLE

Similar to software development life cycle, software testing has its own life cycle termed ‘software testing life cycle’ (STLC). Software testing has various phases which are defined in test planning domain. Let us define the reviews associated with execution of testing.

6.7.1 TEST READINESS REVIEW

Test readiness reviews are generally conducted by test managers, test leads or senior testers with project manager, module leaders or project leaders to ensure that product under development is ready for testing as per the phase of testing applicable. Test readiness review is done at each stage of development such as unit testing, integration testing, interface testing, system testing, and acceptance testing. Test readiness review for system testing may include the following stages,

- Review comments for different work products indicate the possible weak areas in software development, and possible problematic areas where defects can be found. All comments may not be defects but decision about comments (accept/reject/defer/change) must be finalised and actions must be taken according to decisions. Test manager must review all review comments of the artifacts such as peer review comments, walkthrough or inspection comments to check whether all comments are closed or not. If review comments are open, it is likely that defects would be found in the respective areas.
- Review of unit test report to check whether all unit testing defects are closed or not. If unit testing defects are open, the same defects can be found in system testing also. Sometimes, it may not be possible to close all unit testing defects but then, release note given to system testing must record such defects as known issues and their possible impact on the working of a product under testing.
- Most of the softwares need installation before they can be used. One needs to install software to check whether installation testing has been successful or not. Installation testing may include installation through CD or other media (like pen drives), installation from network, or remote installation as defined by requirement statement. If software cannot be installed, then it cannot be used by the users. Sometimes installation defects are called ‘showstopper’ as they can stop usage of software.

Prerequisites Testing Software installation has some prerequisites such as operating system, database, and reporting services as the case may be. If requirement statement specifies that installation must check for the availability of such prerequisites, then it must be tested accordingly. Sometimes, if prerequisites are not available, installation may prompt, or prerequisites may be installed directly during installation. Also, installation can be manual, autorun or semi-auto depending upon the requirements.

Updations Testing Sometimes, it is expected that while installing software, there must be a check whether the same or similar software already exists there or not. One may experience that while installing an operating system, it checks whether some operating system already exists and what is the version of the existing system. Generally, advance versions may not be overwritten by the older versions. Sometimes, it also prompts for a repair or new installation, if the versions available are older than the one being installed. The options must be guided by requirements.

Un-Installation Testing Un-installation testing is done when the developing organisation wishes to check that un-installation is clean. Any file existing before installation must not get replaced while installing the application. Similarly, when an application is un-installed, all the files installed must be removed from the disk. Applications running before installation must be running after installation and also once un-installation is complete.

- Smoke testing is done to check whether the application is alive or not. It refers to testing the application to understand if a user can work with it or not. Any living application must occupy some disc space, and must be shown in the programs available on a computer. Launching of an application and executing very high-level test cases must be possible. The purpose of smoke testing is to check whether the application can be worked with or not. It does not establish any functionality or correct behavior of the application. Some people refer to smoke testing as smell testing.
- Sanity testing, also called as Build Verification Testing (BVT) is done to check whether the major functionalities of an application are available to the user or not. In sanity testing, one works with software at macro level to understand its ability to produce consistent results and availability of major functionalities or features. Many people consider smoke and sanity testing at the same testing level. Build verification testing ensures that the results produced by executing test cases are repeatable and reproducible. If an application fails in sanity testing, there is no point in going ahead with system testing as the defects found may not be reproducible and correct/wrong behavior cannot be proved.

Test readiness review also includes review of all background arrangements including help files, release notes, and supporting documentation as defined in test plan document. Test readiness review may also include whether the test team has all information and hardware/software/licenses required for creation of test bed, availability of tools including regression testing tools, configuration management tools, defect tracking tools, training, and other resources for performing testing.

6.7.2 TEST COMPLETION REVIEW

Test completion review is conducted when a testing cycle is completed and test results are created. Outcome of such reviews includes analysis of testing process, number and type of defects found, and number of iterations completed. Test completion review must check the coverage of testing in terms of requirement coverage, functionality coverage, and feature coverage as the case may be (as intended in test plan), and the number of defects found as against the targeted number of defects. Final outcome of test completion is a recommendation

about the status of an application. It must tell the development team whether the product can go to the next phase or needs any repairing, retesting and regression testing before it can be declared as successful.

6.8 COVERAGE IN VERIFICATION (TEST DESIGNING)

Different instances of verification offer different ways of measuring coverage achieved in verification. Definition of coverage must be in sync with the document or artifact under verification. Let us consider few coverage considerations in coding as an activity under verification (the same can be applied to other areas of development and testing).

- Statement coverage
- Path coverage
- Decision coverage
- Decision-to-decision path coverage

6.8.1 STATEMENT COVERAGE

Programs include numerous 'statements'. Definition of 'statement' must be done by an organisation before calculating any coverage. Some people consider ';' as a sign of statement while some people consider a code line with or without ';' as a statement. Definition must be used consistently atleast for the given project. Sometimes, non-executable lines are also considered as statement as they are very important from maintainability point of view, for example, comments.

When a verification activity is conducted, how many lines are verified against the total number of lines available in a given unit will represent the statement coverage. 100% may be targeted for a unit but for complete application, it may not be possible to verify each and every line in the application. One may concentrate on the important part of code which will be mostly used. In such cases, coverage may be less than 100%.

6.8.2 PATH COVERAGE

Path represents the sequence of control flow from entry to exit in a given code. Application may have several paths in which the flow may progress while executing the application. If there are no decision loops in the program, it will have a single path in which it can progress. As soon as decisions are introduced in the application, number of paths increase accordingly. When decisions are mutually exclusive, number of paths increase as per number of decisions. But if decisions are dependent on each other, increase in number of paths is exponential.

In verification activity, if each path is considered for verification, it may give 100% path coverage. Each path has different probability of happening and some paths with lesser probability of getting executed than others may not be verified, if time available for verification is a constraint. In such cases, path coverage may be less than 100%.

When the decision gives a range like 'for' or 'do while' loops, each outcome may represent a different path. One may apply techniques like 'equivalence partitioning', and 'boundary value analysis' to decide which path needs to be verified so that it would be equivalent to many other paths. In such cases, path coverage may be less than 100%.

Path Predicate Path prediction may be done depending upon the decision points available in the domain. Each decision gives several paths depending upon the type of decision included. Decision, depending upon the previous decisions, adds to number of paths exponentially while mutually independent decisions add the number of paths arithmetically.

Generally, decisions like 'if and else' give two paths while decisions like '< or >' may give many more paths. Also, when outcome of one decision is an input to another decision, it multiplies the paths.

Achievable (Feasible) Path Achievable paths are the feasible paths when a user is actually using the application. If there is any path where the control would never go, it is termed 'infeasible path' or 'redundant path'. One must remove such path from the application being developed. Redundant path indicates a possible defect in verification.

Path Sensitising The basic strategy in path sensitising is to pick a defect or bug to be tested and sensitise paths; some of which go through that defect. A path is sensitised if the associated test checks for faults along the path; therefore the main objective is to find a set of tests which will determine if the defect has occurred or not. Path sensitising method uses less time and is more efficient in locating defects.

Path Profiling (Instrumentation) Path profiling is intended to find heavily-used paths from other paths present in the application where the control goes rarely. Heavily-used paths are more important for the application, and hence must be correct and deliver good performance to generate adequate confidence in users. Generally, verification must target to cover such paths for better product.

6.8.3 DECISION COVERAGE

When an application is getting executed at several places, it may have to make decisions depending upon the situation it faces. Each decision may have several paths. One may not be able to verify all paths and hence may concentrate on the decision part only. If one outcome is verified with an assumption that all other outcomes will be correct if the one selected is correct, it gives decision coverage less than 100%.

When the decision gives a range like 'for' or 'do while' loops, one may apply techniques like 'equivalence partitioning', and 'boundary value analysis' to decide which path needs to be verified. One may need to decide more probable paths for verification. In such cases, path coverage may be less than 100%.

6.8.4 DECISION-TO-DECISION PATH COVERAGE (DD PATH COVERAGE)

There may be a path between two decisions. If decisions are mutually exclusive, then there may be a single path between two decisions. If latter decisions are depending upon earlier decisions, then there may be several paths possible between different decisions.

Verification of path between two decisions will be used for statement coverage in addition to decision coverage. Decision coverage may be restricted to the decision part, while flow happening after the decision till next decision is captured by decision-to-decision path coverage.

6.8.5 NICE DOMAIN AND UGLY DOMAIN

During verification, it is considered that the domain under testing is a nice domain. The characteristics of nice domain are that the domain is linear, complete, systematic, orthogonal, consistently closed, simply connected and convex. This assumption makes an application or code review easier, as one set of transaction can be considered to represent all the sets available in that domain.

On the other hand, there can be some ugly domains which do not follow the assumptions given above. Such domains are nonlinear, less or not systematic, inconsistent and so on. If we consider an example of division of two numbers as ' a/b ' then ' a ' can assume any number and is a nice domain. But ' b ' can have any value other than 'zero'.

KV Charts A KV map (Karnaugh map or Veitch map) may contain number of boolean variables. Generally, more than 6 variables make the chart very complicated. Being a boolean in nature, each variable can contribute maximum two possibilities indicated as '0' and '1'. When we take multiple variables, one can plot all system states by considering different states of an individual variable. Once the variables are defined, the output possibilities are transcribed according to the grid location provided by the variables. Thus, for every possibility of a boolean input, the output possibility is defined.

When the KV map is completed, to derive a minimised function, the '1's or desired outputs are grouped into the largest possible rectangular group in which the number of grids boxes (output possibilities) in the groups must be equal to a power of 2. The boxes can be used more than once only if it generates the least number of groups. All '1's or desired output possibilities must be contained within a grouping.

The groups generated are then converted to boolean expression by locating and transcribing the variable possibility attributed to that box, and by the axiom laws of boolean algebra in which if the (initial) variable possibility and its inverse are contained within the same group, the variable term is removed. Each group provides a 'product' to create a 'sum-of-products' in the boolean expression. To determine the inverse of the KV map, the '0's are grouped instead of the '1's. The two expressions are non-complementary.

6.9 CONCERNS OF VERIFICATION

There are few concerns associated with verification activities. In general, verification is expected to find about 60% of the total number of defects. The defects found in verification must have some probability of impacting final users if they go undetected. Verification defects must not be only of academic nature but must also affect software negatively, if they are not removed. Effects may be in terms of not achieving some quality factors important from software usage perspective.

6.9.1 USE OF RIGHT VERIFICATION TECHNIQUE

Every verification technique has some advantages and disadvantages. One may have to use verification techniques judiciously. For test cases and code files, peer-to-peer review may be more advantageous from cost perspective while for requirement statement, inspection may be selected as a technique. Project manager/organisation may decide the technique on the basis of various factors of evaluation such as cost-benefit analysis, importance from user's perspective and so on.

6.9.2 INTEGRATION OF VERIFICATION ACTIVITIES IN SDLC

As seen in V & V model, every phase of software development life cycle has associated phases of verification as well as validation. Selection of verification technique must be such that it must find the defect as early as possible. This can prevent stage contamination. The 'check' part must be very close to the 'do' part of the process so that defects in the 'do' process are found and removed. This improves development process capability.

6.9.3 RESOURCES AND SKILLS AVAILABLE FOR VERIFICATION

Very important inputs for verification are time and people with required skills. If reviewer/inspector is capable with sufficient knowledge and ability, verification can be very effective. Otherwise, it becomes ineffective as people may not find much value addition by conducting such a review. Defects found in verification must have some possibility of affecting final users, if they go undetected.

6.10 VALIDATION

Validation is a disciplined approach to evaluate whether the final built software product fulfills its specific intended use. It is meant to validate the requirements as defined in requirement specification, ensure that the application as developed matches with the requirements defined, and is fit for the intended use. Real-life scenarios are achieved in test lab to perform validation of the application under testing. Validation steps may involve test bed preparation, test scenario definitions, test case definitions, test data definitions, test case execution, defect identification, retesting, and regression testing.

Validation is also called 'dynamic testing' as the application is executed during validation, with the intention to find defects. Validation must be done by independent users, functional experts, and black box testers to ensure independence of testing from development activities. It helps in analysing whether the software product meets the requirements as specified in requirement statement, and can give a user required level of confidence that the application will not fail in normal as well as abnormal circumstances.

6.10.1 ADVANTAGES OF VALIDATION

- Black box testing approach is used for system, integration and acceptance testing. It represents actual user interaction with the system without any consideration of internal structures or how the system is built. It is generally independent of the platform of development, database or any other technical aspect related to software development.
- Validation is the only way to show that software developed by following all processes of development is actually functioning as desired and is usable. It can answer the question—'Can we use the software?'

6.10.2 DISADVANTAGES OF VALIDATION

- No amount of testing can prove that software does not have defects. If defects are not found by conducting the defined test cases, we can only conclude that there is no defect in the set of transactions actually executed. There can be many more defects not captured by these test cases.
- For unit testing and module testing, stubs and drivers are needed to be designed and used during testing. Stubs and drivers need additional efforts of development and testing before they can be used.
- Sometimes, it may result into redundant testing as the tester is not aware of internal structure, and same part of the code gets executed again and again. This is a common scenario when we have some library functions which are called several times during execution, and they are tested each time since the tester does not know that the same function is being called again and again.

6.10.3 PREREQUISITES FOR VALIDATION

Depending upon the method followed for validation and work product under validation, there may be different inputs required for validation. Any process would need some inputs coming to the work bench and some outputs going out from the system. Inputs for the validation process may include work product, test plan, and test cases, whereas output from the validation work bench may include test report and defects.

Prerequisites for validation may include the following.

- Training required for conducting validation. Training may include domain knowledge and knowledge about testing and various test tools
- Standards, guidelines, and tools to be used during validation
- Validation 'do and check' process definition

6.11 VALIDATION WORK BENCH

A validation work bench is a place where validation activities are conducted on the work products, and may be a physical or virtual entity. For every work bench, the following basic things are required. Figure 6.4 shows a validation workbench.

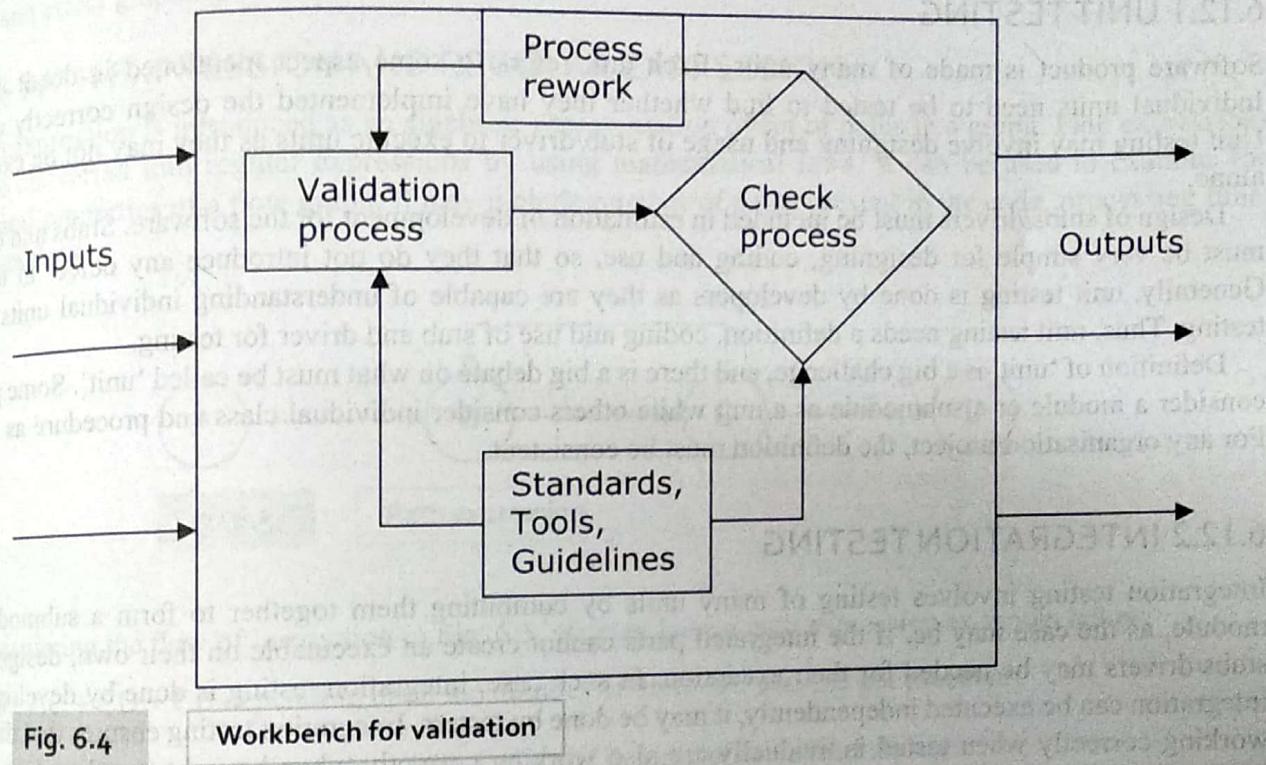


Fig. 6.4

Workbench for validation

Inputs There must be some entry criteria definition when inputs are entering the work bench. This definition should match with the output criteria of the earlier work bench. For example, if we are validating an application or part of it, then we must have a written and compilable code as an input to the work bench along with test plan and test cases/test data as per definition of input.

Outputs Similarly, there must be some exit criteria from work bench which should match with input criteria for the next work bench. Outputs may include validated work products, defects, and test logs.

Validation Process Validation process must describe step-by-step activities to be conducted in a work bench. It must also describe the activities done while validating the work product under testing.

Check Process Check process must describe how the validation process has been checked. It is not a validation of the work product but a process. Test plan must define the objectives to be achieved during validation, and check processes must verify that the objectives have been really achieved.

Standards, Tools, and Guidelines These may be termed 'the tools' available for validation. There may be testing guidelines or testing standards available. Sometimes checklists are used for doing validation.

6.12 LEVELS OF VALIDATION

Similar to verification at different stages, validation can be applied at different stages of software development. Some of these stages are explained below.

6.12.1 UNIT TESTING

Software product is made of many units. Each unit refers to some aspect mentioned in detail design. Individual units need to be tested to find whether they have implemented the design correctly or not. Unit testing may involve designing and usage of stub/driver to execute units as they may not be executed alone.

Design of stubs/drivers must be included in estimation of development for the software. Stubs and drivers must be very simple for designing, coding and use, so that they do not introduce any defect in testing. Generally, unit testing is done by developers as they are capable of understanding individual units under testing. Thus, unit testing needs a definition, coding and use of stub and driver for testing.

Definition of 'unit' is a big challenge, and there is a big debate on what must be called 'unit'. Some people consider a module or a submodule as a unit while others consider individual class and procedure as a unit. For any organisation/project, the definition must be consistent.

6.12.2 INTEGRATION TESTING

Integration testing involves testing of many units by combining them together to form a submodule or module, as the case may be. If the integrated parts cannot create an executable on their own, designing of stubs/drivers may be needed for their execution. In such case, integration testing is done by developers. If integration can be executed independently, it may be done by testers. Integration testing ensures that the units working correctly when tested individually are also working correctly when brought together. Integration testing helps in identifying if there are any issues of parameter passing as different units are interacting with each other. Test cases are defined by referring to low-level design. Integration testing mainly refers to detail design or low-level design.

6.12.3 INTERFACE TESTING

Interface testing involves testing of software with the environmental factors (such as database and operating system) where the application is supposed to work. If an application is supposed to work with some other applications, third-party components (such as communication software) combining all these applications and then conducting testing is termed 'interface testing'. Parameter passing is the most important criterion for interface testing. Test cases are defined by referring to high-level design or architectural design.

6.12.4 SYSTEM TESTING

System testing involves end-to-end testing of a system to find the behavior of a system with respect to expectations. It needs to be done by testers as if they are the users of the system. The testing process involves defining test scenario, test cases and test data for testing. System testing is actually a set of processes which may include functionality, user interface, performance, and security testing. Test cases are defined by referring to requirements and designs.

6.12.5 CAUSE AND EFFECT GRAPHING

While using the system, there may be several inputs given to the system, and the system is expected to respond to those inputs. Equivalence partitioning is used extensively along with boundary value analysis for plotting the causes and their effects on the system. ‘When somebody clicks a login button with valid login and password, the system allows that user to enter the system as a valid user’—this can be an example of cause and effect graphing.

6.12.6 PATH EXPRESSION AND REGULAR EXPRESSION

A path expression is introduced as an algebraic representation of set of paths in a graph. Path expressions can be converted into regular expressions by using mathematical laws. It can be used to examine the structural properties of a flow graph. It may include number of paths present in the code, processing time, and data flow.

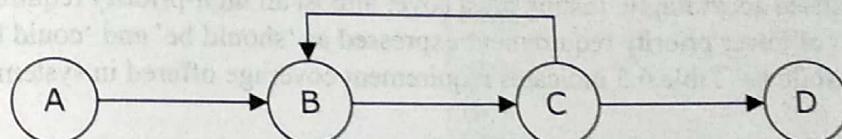


Fig. 6.5

Path expression

Considering the flow of instruction in Fig. 6.5, we may have a path expression as shown below.

$AB + BC + CD + CB$ depending upon the number of times BC and CB get executed.

Path Sum Path sum denotes the number of paths in parallel as defined above.

Path Product When a larger path is made of two successive paths, the total number of paths would be a multiple of number of path in each instance. Thus, such paths are in a series to each other.

Path Loops Loops are infinite (or very large) number of parallel paths. It may be expressed in terms of definition of exponential values.

6.13 COVERAGE IN VALIDATION (PRIORITISATION/ SLICE BASED TESTING)

Different instances of validation may offer different coverages depending upon the test plan and definition of coverage. It has a relationship with requirement traceability matrix of an organisation and the customer. Validation mainly talks about black box testing methodology where we may not know how things are structured in an application. It mainly covers functionalities and requirements as defined in requirement statement.

When there is a time constraint and testing is expected to give results faster, 100% testing is not feasible. System may be released by testing the critical parts with respect to user requirements. This is also termed ‘prioritisation technique’ where primary usage parts of the system are tested fully so that the system will not

have defects under normal circumstances. The slices are obtained on the basis of prioritisation as defined below.

Following are some of the famous coverages offered during validation.

- Requirement coverage
- Functionality coverage
- Feature coverage

Requirement Coverage Requirements are defined in 'requirement specifications' documents. Traceability matrix starts with requirements as defined in requirement statement and goes forward upto test results. Theoretically, all the requirements as defined in requirement statement must have corresponding test cases and test results. Passing or failure of such test cases can define whether the requirements have been achieved or not.

All requirements are not mandatory. They can be put in different classes such as 'must', 'should be', and 'could be', and prioritised accordingly. Testing must cover almost all high-priority requirements expressed as 'must' and some part of lower priority requirement expressed as 'should be' and 'could be'. Thus, coverage may be expressed as follows. Table 6.3 indicates requirement coverage offered in system testing.

Table 6.3

Requirement coverage definition

Priority of requirement	Coverage offered
P ₁ (Must)	100%
P ₂ (Should be)	50%
P ₃ (Could be)	25%

Functionality Coverage Sometimes, requirements are expressed in terms of functionality required for the application to work successfully. Functionalities are also defined at different priorities as per their importance to the users. All functionalities are not mandatory from the user's perspective. Test cases representing high-priority functionality must be tested to a larger extent than the functionalities with lower priorities. Thus, coverage may be expressed as follows. Table 6.4 indicates functionality coverage offered in system testing.

Table 6.4

Functionality coverage definition

Priority of functionality	Coverage offered
P ₁ (Must)	100%
P ₂ (Should be)	50%
P ₃ (Could be)	25%

Functionality coverage cannot help in identifying the coverage established by non-functional requirements such as performance and user interface.

Feature Coverage An application may need some features as defined in requirements. Features are a group of functionalities doing same/similar things. The same feature may have different ways of doing things, and each of them is a functionality for the application. Accomplishing atleast one of the functionalities may help in achieving the user requirements as a feature. If we consider saving a file in Microsoft Word, then 'saving' represents the feature while 'different ways to save the file' may represent different functionalities. All may not be equally important and one can perform a probability analysis on the basis of user profile.

Feature coverage talks about covering a feature required by the user. It may mean covering atleast one of the functionalities which represents a feature provided, even if there are multiple ways of doing things. Thus coverage may be expressed as follows. Table 6.5 shows a feature coverage offered in system testing.

Table 6.5

Feature coverage definition

Priority of feature	Coverage offered
P ₁ (Must)	100%
P ₂ (Should be)	50%
P ₃ (Could be)	25%

Specification-Based Testing Specification-based testing refers to the process of testing a program based on its specification, as defined in requirement specification, design specifications or user manual. Generally, it states how the applications behavior should be. In particular, we can develop test cases based on the specification, without seeing an implementation of the program. Furthermore, we can develop test cases before the programming on the basis of behavioral expectations.

Without writing a program to implement requirements based on the specification given and generating a set of test cases and test data that one thinks would be sufficient to test a program may be considered as specification testing. If one thinks that the specifications are incomplete in any way, then one has to state what assumptions one is making about how it should be completed or clarified.

Remember that a test case consists of the input data and the expected result for that input data. This testing generally concentrates on external functions, and does not test technical, system, and legal requirements completely.

6.14 ACCEPTANCE TESTING

Acceptance testing is generally done by the users and/or customers to understand whether the software satisfies their requirements or not, and to ascertain whether it is fit for use. There are some predefined methodologies, conditions and test cases in the contract which will be used for acceptance testing. Users/customers execute these test cases to show if the acceptance criterion for the application has been met or not. There are two (sometimes three) levels of acceptance testing.

- Alpha Testing** Alpha testing represents the testing done by the customer in development environment in front of the development team. The testing is done at development site with dummy data, either created by developers or shared by customer. In reality, alpha testing may be done by testers in front of the customer to show that software is working. It can be used as a tool for training key users on the application. Any major

problems in terms of requirement understanding, functioning of software, etc. can be cleared during alpha testing before software is formally delivered. Data used during testing may be shared by customer as a live business data or dummy data created by customer or tester. Test data created by testers must be reviewed by business experts/customer/users. There is no direct impact of such testing on customer business as testing is done off-site. This is also termed as 'dry run' by some people.

- **Beta Testing** Beta testing represents a business pilot where testing is actually conducted by customer in production/semi-production environment. It is done at selected customer locations by representatives from the customer side. Testers/developers may be appointed to help the customer in testing the application, and recording the problems, if any. Beta testing is extensively used as parallel testing to give confidence to the users that the software really works as per established existing system at customer's place.

- **Gamma Testing** Gamma testing is used for limited liability testing at selected places. Gamma testing is generally done by the product development organisation that wishes to have limited market demonstration and piloting of an application in the market. The application is given to few people for using it in production environment, and feedback is obtained from them about the features incorporated as well as something missing or wrongly interpreted. This is used by product organisations to understand customer reaction to a new or enhanced product.

6.15 MANAGEMENT OF VERIFICATION AND VALIDATION (V & V)

Verification and validation techniques are complementary to each other, and not the replacement of each other. A project must estimate time and efforts required for preparing and executing verification and validation activities as per project/quality plan and test plan definitions. The steps involved in verification/validation are as follows.

6.15.1 DEFINING THE PROCESSES FOR VERIFICATION AND VALIDATION

An organisation must define the processes applied for verification and validation activities during the development life cycle phase of the project. The processes involved may be as follows.

- *Software Quality Assurance Process* These processes concentrate on developing the techniques/procedures for development and testing of the project. They may be more generic, indicating overall approach of handling verification and validation activities, or may be very specific for the project/customer. The processes may be inherited from the organisational process database, or may be tailored as required for a particular situation. Generally, quality assurance process forms a part of prevention cost, where care is taken so that minimum (zero) defects are introduced in the product.

- *Software Quality Control Process* These processes concentrate on developing the approach for verification and validation activities during software development and testing. They can be more specific for the project under development. The process definition must have the details of types of activities and stakeholders involved at each stage. Generally, quality control process forms a part of appraisal and failure cost. Quality control process must ensure that defects are found as near to its origin as possible, so that stage contamination can be reduced.

- *Software Development Process* These processes concentrate on the approach of developing software where minimum (zero) defects will be introduced. They may cover software quality control processes along with other SDLC processes. They define usage of development process, and templates, formats, guidelines

and standards to be used during development and testing. These processes may be derived from organisational process database or may be acquired from external sources such as national or international standards. Sometimes, customer defined software processes are adopted by projects.

- **Software Life Cycle Definition** Software life cycle definition is essential to establish development and testing processes for the software. It depends on the life cycle definition of software. One may select a waterfall model or iterative development model as per specific requirements of customer.

6.15.2 PREPARE PLANS FOR EXECUTION OF PROCESS

When a project proposal is made, it must contain a definition of what is meant by a successful delivery, and how quality of deliverables will be achieved, ensured and tested during life cycle of the project. The contract must also cover the acceptance criteria and deliverable definitions. The plans involved are as follows.

- Software development plan and schedule which include responsibilities and time schedule for verification/validation activities must be defined at the start of the project. It must define entry and exit criteria for each activity. These plans must be refined as and when required.
- Software quality plan and software test plan must define the use of different methodologies for ensuring quality of deliverables, verification and validation activities associated with development. Software quality plan and software test plan must be refined if required.
- Software acceptance testing plan must be defined where acceptance criteria is finalised. Software acceptance test plan must cover the entry and exit criteria at each stage of development. Exit criteria at the earlier stage must match with entry criteria of the next stage.

6.15.3 INITIATE IMPLEMENTATION PLAN

The plans made at the time of proposal/contract must be implemented during development life cycle of a project. There must be formal records of requirements review, design review, code reviews, unit testing, integration testing, interface testing, system testing, and acceptance testing. The results must be recorded and corrective/preventive actions must be initiated from the results of verification and validation. Whenever required, verification and validation plans must be changed to accommodate the changes in development activities, scope, customer expectations and so on.

6.15.4 MONITOR EXECUTION PLAN

The verification and validation activities must be monitored during development life cycle execution. Project manager and test manager must oversee that the activities defined in various plans are being executed and the results are being logged in test log. The time required for various activities, number of test cases executed, and number of defects found must be monitored to find out their sufficiency and effectiveness for the purpose. Corrective/preventive actions must be planned when discrepancies are observed with respective planned arrangements, or defects are logged, or nonconformances are observed in the processes/work products.

6.15.5 ANALYSE PROBLEMS DISCOVERED DURING EXECUTION

Execution of verification and validation processes may bring out many problems in the product as well as processes associated with development and testing. Root cause analysis of problems and planning for improvement actions are essential parts of continuous improvement. An organisation is expected to perform a root cause analysis, initiate corrective actions to remove the chances of recurrence of the same problem, and take preventive actions along with correction.

6.15.6 REPORT PROGRESS OF THE PROCESSES

During the life cycle execution of a project, there are milestone or period dependent reporting mechanisms established by plans and schedules. The outcome of verification and validation activities as planned must be formally reported to management, customer, and development team to make them aware of the project progress and problems faced by the product/processes during project life cycle.

6.15.7 ENSURE PRODUCT SATISFIES REQUIREMENTS

The ultimate aim of verification and validation activities undertaken during project execution is to achieve customer satisfaction. The requirements specified by the customer and defined by the organisation must be tested fully to ensure that these have been achieved in the product offered to the customer. Requirement traceability matrix helps project management in understanding the successful implementation of requirements in software on the basis of test execution log. Results of verification and validation must be logged in requirement traceability matrix. Defects found will show that which requirements have not been implemented successfully or are missing in the application under testing.

6.16 SOFTWARE DEVELOPMENT VERIFICATION AND VALIDATION ACTIVITIES

Verification and validation activities are spread over the life cycle of software development. The cycle of verification and validation may include the following.

Conceptualisation Conceptualisation is the first phase of developing a product or a project. For the customer, Conceptualisation means converting the thoughts or concepts into reality. It may be through 'proof of concept' approach or 'prototyping' approach. During proposal, the supplier may give some approach or solution, and the customer may have to evaluate the feasibility of such an approach or solution from product perspective. The supplier must understand what is expected by the customer, and whether it can be provided or not by an organisation. In Conceptualisation, the main stress of verification and validation activity is to determine feasibility of an approach for the project. Feasibility may be depending upon various factors such as technical feasibility, economic feasibility, and skill availability. Verification and validation ensure that an organisation is able to undertake a project and will be able to achieve its mission through the project. Verification and validation by the customer is intended to check the technical capabilities of a supplier to complete the project.

Requirement Analysis Requirement phase may start from conceptualisation. Requirements clarification happens through dialog with users/customer. It can be done by various approaches such as joint application development and customer survey. Requirement analysis verification and validation shows the feasibility of requirements and gives inputs to design approach. It can help in finding the gaps between proposal and requirements so that further clarifications can be achieved. Requirement reviews help in understanding different aspects of customer needs as well as various trade-offs done. Requirement validation can help in identifying any assumption as well as implied requirements in application development.

Design Requirements are implemented through design. Verification and validation of design include understanding that design is complete in all respects and matches with requirements. Requirement traceability ensures that all requirements are converted into design. Design validation involves dummy execution of a

product design through dataflow diagrams or prototyping, to find whether the design reflects the requirements correctly along with its feasibility. When a flow is complete, design is assumed to be complete.

Coding Coding involves code review and testing of the units, as part of verification and validation, to make sure that requirements and design are correctly implemented. Units must be traceable to requirements through design.

Integration Integration validation and verification show that the individually tested units work correctly when brought together to form a module or submodule. It involves testing of modules/submodules to ensure proper working with respect to requirements and designs. Integration must satisfy low-level design definition such as parameter passing and communication between different units. Integration with other hardware/software is defined in architectural design. Interface testing must satisfy architectural design declared earlier.

Testing Test artifacts such as test plan, test scenario, test bed, test cases, and test data must be subjected to verification and validation activities. Test plan must be complete, covering all aspects of testing expected by the customer. Test bed must mimic real life scenario. Any simulators or any known issues related to test-environment creation must be noted in the risk-management section in test plan. Test scenario must be complete and feasible. Test cases must cover the application completely. Test cases for acceptance testing must be validated by customer/user/business analyst as the case may be.

Installation Application must be tested for installation, if installation is required by the customer. The documentation giving instructions for installation must be complete and sufficient to install the application. Verification and validation must ensure that there is adequate support and help available to common users for installation of an application. User must be guided through all prerequisites. Installation may even be interactive, if requirement specifies so.

Documentation There are many documents given along with a software product such as installation guide and user manual. They must be complete, detailed and informative, so that it can be referred by a common user. The list of documents must be mentioned in contract or statement of work so that compliance can be checked. The documents must undergo validation to ensure that documents and product are in sync with each other.



Tips for verification and validation activities

- Software testing includes verification and validation activities at each stage of development. It starts from proposal and ends only when the project crosses acceptance testing.
- Verification and validation activities cannot replace each other, rather they support each other.
- Maximum (about 60%) defects can be found in verification. Verification can reduce stage contamination as defect percolations are reduced.
- Verification has three different types, viz. reviews, walkthrough and inspection. Different artifacts undergo different review techniques. Test artifacts also need verification.
- Validation also occurs at different phases of development life cycle. About 25% of the defects can be found in unit testing. Over-reliance on system testing is not an economic solution of finding defects.

Summary

This chapter provides a clear exposition of verification and validation activities. It covers various methods of verification such as reviews, walkthrough and inspection and various stages of verification such as requirement verification, design verification, and test artifacts verification. Advantages of different levels of reviews such as self review, peer review, walkthrough, and inspection have been dealt in detail. It also presents audits as an independent way than quality control and quality assurance. In-process reviews and post implementation reviews have also been elucidated.

The Second half of the chapter focusses on validation techniques such as unit testing, integration testing, interface testing and system testing. The chapter concludes with a clear overview of acceptance testing.

- 1) Discuss the advantages and disadvantages of verification.
- 2) Describe different types of verification on the basis of parties involved in verification.
- 3) Describe the advantages and disadvantages of self review.
- 4) Describe the advantages and disadvantages of peer review.
- 5) Describe the advantages and disadvantages of walkthrough review.
- 6) Describe the advantages and disadvantages of inspection.
- 7) Describe the process of inspection.
- 8) Describe the auditing process.
- 9) What are the different audits planned during development life cycle?
- 10) Explain various types of in-process review.
- 11) Explain the concept of post-mortem review. Why it is essential for a learning organisation?
- 12) Explain test readiness review.
- 13) Explain different ways of assessing coverage in verification.
- 14) What are the concerns of verification?
- 15) Discuss the advantages of validation.
- 16) How coverage is measured in case of verification?
- 17) How coverage is measured in case of validation?
- 18) Discuss the different levels of validation.
- 19) Explain different levels of acceptance testing.



CHAPTER

7

V - TEST MODEL



OBJECTIVES

This chapter establishes 'V model' (validation model) and 'VV model' (verification and validation model). It also defines roles and responsibilities of three critical entities in software development.



7.1 INTRODUCTION

Testing is a lifecycle activity. It starts when the proposal of software development is made to a prospect, and ends only when application is finally delivered and accepted by the customer/end user. For product development, we may define each iteration of development as a separate project, and

several projects may come together to make a complete product. In case of maintenance, the cycle may get repeated for every instance of change in present system. For a customer, it starts from a problem statement or conceptualisation of a new product, and ends with satisfactory product receipt, acceptance and usage. For every development activity, there is a testing activity associated with it, so that the phase achieves its milestone deliverable with minimum problem (theoretically, no problem). This is also termed 'certification approach of testing' or 'gate approach of testing', where the gate of a phase opens for delivery, only when the deliverable carries a certification that it meets exit criteria defined for that phase which is entry criteria for the next phase.

Each phase of software development activities must consider corresponding testing activity associated with it. Project plan and estimations in terms of time and effort for a project must consider all the activities of testing (verification and validation) along with development activities corresponding to different phases. The following diagram 7.1 shows the details of software lifecycle development activities, and software lifecycle testing activities associated with them.

7.2 V MODEL FOR SOFTWARE

Validation model describes the validation activities associated with different phases of software development.

- At the requirement phase, there is system testing and acceptance testing, where system testers and users confirm that requirements have been really met or not.

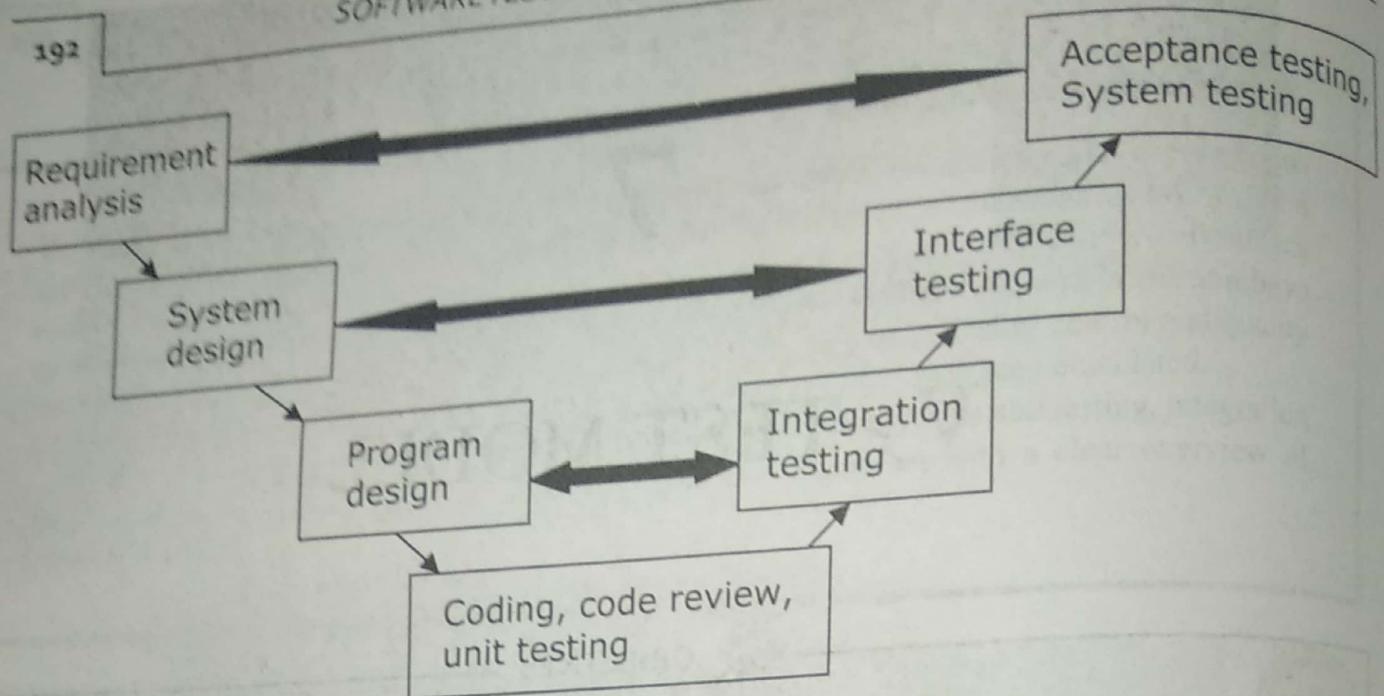


Fig. 7.1

V model for testing (Validation model)

- Design phase is associated with interface testing which covers design specification testing as well as structural testing.
- Program-level designs are associated with integration testing.
- At code level, unit testing is done to validate individual units.

7.2.1 STRUCTURED APPROACH TO TESTING

Testing activities for software development life cycle phases must be planned in advance, and conducted as per plan. Test policy and test strategy/test approach for performing verification and validation activities, responsibilities of stakeholders for supporting or doing these activities, inputs and outputs from each phase of the process, and milestone deliverables must be documented beforehand to avoid any problem in final deliverable to customer. People involved in these activities must have required knowledge, expertise, experience, and training for doing verification and validation activities. Generally, testing activities are referred in project plan but they are detailed in quality plan and verification plan. Some organisations also refer to this as 'verification and validation plan' (V & V plan).

7.2.2 ACTIVITIES DURING EACH PHASE OF SOFTWARE DEVELOPMENT LIFECYCLE

Activities of verification and validation are planned during different phases of software development life cycle, from proposal level till product is finally accepted by the customer. The software development process plan must define the development and testing activities to be conducted in each phase of development, and also, the people responsible for conducting and supporting those activities as stakeholders. Plan of software development must decide about 5 Ws (What, Where, When, Why, and Who) and H (How) with respect to people, processes, tools, training, etc. The information must be readily available, to the team doing these activities and the stakeholders, about their roles and responsibilities for those activities. Activities must be referred to during each phase of software development and testing.

7.2.3 ANALYSE STRUCTURES PRODUCED DURING DEVELOPMENT PHASES FOR ADEQUACY AND TESTABILITY

Documents and work products produced during a life cycle phase must be analysed beforehand to understand their coverage, relationships with different entities, structure in overall development, and traceability. An organisation must have a definition of processes, guidelines and standards which can be used for making such documents and artifacts. Documents and artifacts compliance must be measured in numerical terms with respect to adequacy for the purpose and testability (pertaining to customer needs or organisational standards).

7.2.4 GENERATE TEST SETS BASED ON STRUCTURES

Functional test scenario and test cases are generally developed based upon the functional requirements of the software. Functional requirements refer to the operational requirements of an application. Some requirements may be implemented through designs. In reality, developers implement designs and not requirements.

Structural test scenario and test cases are developed from the structures defined in the design specifications. They must correspond to the structures of the work product produced during development. Requirements/designs are verified and validated by preparing use case diagrams, data flow diagrams, and prototypes with the question 'what happens when' to identify the completeness of design and requirements. For validation testing scenario, one must use techniques like boundary value analysis, equivalence partitioning, error guessing, and state transition for defining valid as well as invalid ways by which user may interact with the system.

7.2.5 ADDITIONAL ACTIVITIES DURING DESIGN AND CODING

Testing in low-level design and coding phases must confirm that first phase output of capturing the requirements and developing architecture matches with the inputs and outputs required by the low-level design phase. High-level design and low-level design must ensure that requirements are completely covered so that software developed covers all requirements. Verification and validation of design must ensure that the requirement verification and validation is proper and can be handled through the structures created for the purpose. Similarly, verification and validation of coding must ensure that all aspects of designs are covered by the code developed. Definition of next phase verification and validation activities can be initiated in previous phases with definition of output criteria.

7.2.6 DETERMINE THAT STRUCTURES ARE CONSISTENT WITH PREVIOUSLY GENERATED STRUCTURES

Software development is like a flow of events, starting from capturing the requirements till acceptance testing is completed successfully. The outputs of one phase must match with the input criteria of the next phase. There must be consistency between various life-cycle phases. If an architect defines one approach of implementation, then the designer must follow the same path while creating a low-level design. If a system designer decides some approach/reusability, then the developer must understand and implement the approach every time he writes a piece of code. Consistency between various phases can improve the maintainability of an application.

7.2.7 REFINE AND REDEFINE TEST SETS GENERATED EARLIER

Test artifacts such as test scenario, test cases, and test data may be generated in each phase of software development life cycle from requirements, design, and coding, as the case may be. The test artifacts so

generated must be reviewed and updated continuously as per review comments and changes in requirements, designs and related artifacts. One must challenge the validity of each artifact to ensure that it meets the desired results in terms of output criteria. If something is found to be missing, extra or wrongly interpreted in any of the artifacts, it must be corrected before testing execution begins.

7.3 TESTING DURING PROPOSAL STAGE

Requirement statement development starts from system proposal. A proposal is created when the customer asks for information, quotation, proposal, etc. At proposal stage, system description may not be very clear. It must talk about major problems to be solved, the possible solution for which the system is designed, and any major constraints in such definition. People use different approaches such as formal/informal proof of concept, modeling, prototyping, etc. The success of all these approaches during proposal stage lies in successfully defining the problem and proposed solution. It may also consider major constraints related to people, technology, etc.

Feasibility study may or may not be a part of the proposal. Sometimes, conducting feasibility study also needs approval as it may involve some cost and effort, and customer may not want people from other organisations to understand things completely. Feasibility study is done by the customer as well as the supplier in search of a possible solution/approach to solve the problem faced by the customer. It may include technical feasibility, economic feasibility, implementation feasibility, organisational fit, process fit, and people fit.

7.4 TESTING DURING REQUIREMENT STAGE

Requirement gathering stage must cover all the requirements for the system, defined in different groups such as technical, economic, legal, operational and system requirements. Verification of problem definition and requirements definition is the foundation on which the system requirement specification is developed further.

Characteristics of good requirements are mentioned below.

Adequate The requirements must explain the entire system covering end-to-end scenario from the user's side. All the assumptions and constraints/limitations in the system must be defined and documented with possible answers. Many times, scenario or use cases are applied to define end-to-end scenario to find if there have been any definition gaps while creating requirement statement. Requirements must not talk about any impossible thing happening.

Clear/Unambiguous Customer expectation must be reflected in requirements clearly. Requirements must be very clear to the architect, designer as well as developer. It must describe various functions, outputs in various forms, system performance requirements, and interfaces with other systems, in the system as applicable. There must not be any doubt about the outcome of system working to customer as well as developing organisation. Final-user scenario must be defined from requirements so generated. Definitions of actors and transactions must be very clear. Requirements must be prioritised and any trade-off done must be explained.

Verifiable/Testable The requirements must be verifiable and testable. The requirement statement is used for defining functional and structural test scenarios and test cases. It must be used for defining functional as well as structural test data and test events. Testers must be able to search the expected results from the requirement statement.

Measurable Tests defined from requirements must have the expected results, possibly in numerical terms or atleast measurable terms which can be verified during testing. When tester executes any test case,

there must not be any doubt about whether the customer expectations have been met or not. Requirement can be a definite number or a range with boundaries defined with terms like minimum, maximum, etc.

Feasible Requirements must be feasible. It must not refer to some thing which is impossible, or not implementable with given technology, approach, software or system configuration. Requirements must define any future state of organisation which can be achieved through proposed solution.

Not Conflicting with Each Other Two requirements must be in sync with each other. They must not specify anything which is contrary to each other. Requirements must be supplementing and supporting each other. In case of any conflicting requirements, one must ask the customer for a trade-off decision. Generally, requirements are prioritised to avoid any conflicting status. If there is any conflict, the requirement with greater priority would be implemented after getting consent from the customer.

Requirements must include constraints in terms of user profiles, system designs, hardware setups, performance criteria, and quality attributes of the proposed system. It must describe very clearly to the system architects about the number of users expected with average/maximum load, response time expected under different load conditions, minimum and maximum system resources available, types of users, authentications or security requirements.

7.5 TESTING DURING TEST-PLANNING PHASE

Test-planning phase includes defining a test strategy and test approach for the given application, writing test plan, test scenario, test cases, and test data. The senior management from testing/test manager is responsible for defining test strategy/approach, while the test lead/test manager develops a test plan to incorporate test strategies, define schedules, methods of testing, evaluation criteria to declare the outcome of testing, and define test-team approach with roles and responsibilities and structure for the system under testing. Testing artifacts must be reviewed for their consistency and accuracy. Verification of testing artifacts may include the following.

- **Generate Test Plan to Support Development Activities** Test plan must be consistent with the application development methodology, schedule, and deliverables. It must describe respective verification and validation activities to be conducted during each phase of software development life cycle. Test plan must refer to test approach or test strategy, define test objectives, scope of testing, assumptions and risks associated with testing. It must contain methods used for defining test data, test cases, test scenario, and execution of testing activities. It must include how many defects would be expected to be found in a work product during testing, the method to be followed when the defects found will be logged in defect management system, information and analysis required to be shared with stakeholders, and any recommendation at the end of testing.
- **Generate Test Cases Based on System Structure** Functional test cases must be based on functional requirements, and structural test cases must be defined on the basis of design and non-functional requirements of the system. It must be used to define test data using different techniques available such as boundary value analysis, equivalence partitioning, error guessing, and state transition. Test cases and test data must be derived from test scenario.
- **Analyse Requirement/Design Coverage** It is very difficult for any test team to create a test suite to cover 100% requirements and designs produced during software development life cycle. The test

manager decides the objective of requirement coverage and design coverage in test plan, and the customer approves it. Coverage less than 100% indicates a risk, and customer must be involved in making such decision. In case of any shortfall in coverage with respective objectives defined, the test team must analyse the situation and perform risk-benefit analysis of lesser coverage with the help of customer and take corrective measures if required. Theoretically, there is no possibility of less coverage as we cannot expect any requirement/design component given to the customer without testing. Practically, it may not be possible, or it may not be required to cover all requirements, but only P1 requirements may be covered to larger extent. One must conduct the analysis and decide the strategy for coverage.

7.6 TESTING DURING DESIGN PHASE

Design is the backbone of a software application. A successful design can convert the requirements into a good application. Design may be made by system architects or designers, and reviewed and approved by project manager and/or customer. Design must reflect the requirements correctly. Verification and validation of design may include the following.

- **Consistency with respect to Requirements** Designs must be consistent with requirements defined. Sometimes, customer expectations as defined in requirements may be contradicting with each other, and one must perform trade-off. If there is any trade-off between the requirements for implementation, it must be mentioned clearly in design, and customer approval should be taken. The requirement coverage of design must be analysed and measured. If there is no design for a given set of requirements, it will never get implemented. On the other hand, if there is a design component not referring to any requirements, it is considered as a defect. This is something extra given which is not required by the customer.
- **Analyse Design for Errors** Designs generated must be reviewed and tested for completeness and accuracy. A design is implemented by coding. Errors in the design will directly reflect the errors in coding and application so developed. Consistency between design elements must be maintained to avoid any mismatches. Reusability can create robust design and flexible code. Designs must be optimised. They must also be traceable to requirements.
- **Analyse Error Handling** A design must define the error-handling process during application use. It must consider all possible errors such as erroneous data input and invalid transactions, and how design must handle them. Error handling of all kinds and possibilities must be covered in designing aspects. An organisation must have standards for error handling, error messaging and user interactions so that designers are very clear about handling them during design. Error handling plays very crucial role in usability testing.
- **Developers Verify Information Flow and Logical Structure** The designs must be analysed for logical flow of information during transactions. Data flow diagrams or dummy execution of the system is done, and outputs derived are measured against expected outputs. Designs must be consistent with the requirements and user expectations. Logic behind different algorithms and handling of different conditions must be analysed to find out completeness as well as redundancy of designs.
- **Testers Inspect Design in Detail** Testers use design for defining structural test scenario, test cases, and structural test data. Test scenario must be end-to-end scenario considering data flow in the system, and must contain valid as well as invalid conditions, and error handling during user interactions with the system. The normal user must be protected by a good designer from wrong working of the system.

7.6.1 ASPECTS TO BE CHECKED

- **Missing Test Cases** Test cases not defined for a particular scenario (either requirement or design) must be caught in test-case review. Missing test cases must be added to close the review comments. Testers must try to write the test scenario using the requirement statement without referring to design. This helps in identifying the missing test cases. Similarly, integration and interface test cases must be written on the basis of design without checking coding.
- **Faulty Logic** If the logic or algorithm described in design is not correct as per requirement statement, then it must be found by testing. Defects so found must be corrected by designers in design, and then by developers in code. The designs must be traceable back to requirements. Analysis of design must be done.
- **Module Interface Mismatch** The data input/output from one module to another must be checked for consistency with design. Parameter passing is a major area of defects in software, where communication in two modules is affected. Analysis of interfaces between different systems must be done as defined by the requirement specifications, and also how it is handled in design. Interface test cases must test the scenario where one system is communicating with another system.
- **Data Structure Inconsistency** Review of mismatch between data structures and definitions between different modules and system must be done for verification of designs. Data formats from different areas must match during transactions to avoid any loss of data due to mismatch of formats.
- **Erroneous Input/Output** If the system needs to be protected from erroneous operations such as huge/invalid input/output, the design must describe how it will handle the situation. A user must be protected from any mishap through proper designs. Different controls such as protective or, detective or corrective controls can be applied to give sufficient protection to users.
- **User Interface Inadequacies** System may need users to input some information at some point when they are interacting with the system. Entry may be through different methods such as keyboard entry, electronic entry, mouse click, system-to-system data transfer and so on. Similarly, the system may ask users to perform different activities and output may be given to users in different ways such as screen displays, reports, etc. The user interfaces defined by the design must be adequate for the purpose of communication, so that users can work with the system comfortably.
- **Correctness of Decisions and Conditions along All Paths** A system may go through different paths and branches of algorithms based on situations faced by it. The design must be such that all the possible conditions must be defined completely. No single condition must have two different logics or outcomes possible. Logic must be checked for any redundant scenario.
- **Inconsistency with respect to Requirements and High-Level Design** Design and development must be consistent with requirements and high-level design. As the system architecture is defined in high-level design which must reflect system requirements, any deviation can lead to extra or missing functionalities, or inappropriately implemented system.

7.7 TESTING DURING CODING

Coding is the most crucial stage in software development where product is actually build. In most of the projects, the number of developers as well as code files is very large in comparison to qualified people

who can review them. Many organisations are completely dependent on self review and peer review for verification of code. Also, unit testing is done by the developers on their own program/code file to ensure that they achieve their objectives as defined by low-level designs. It is crucial but important for the organisation to establish/institutionalise verification/validation of a code so that the product matches with requirement specifications.

7.7.1 ASPECTS TO BE CHECKED

- **Coding Standards/Guidelines Implementation** Many organisations have coding standards/guidelines defined and used in coding phase of product development. When the code files are written, the developers must use these standards/guidelines. While reviewing code, the peer must make sure that standards and guidelines are implemented correctly. Coding standards and guidelines define the best way or suggested way of doing things. It helps in optimisation and better readability/maintainability of code in future.
- **Coding Optimisation** Coding standards must also talk about optimisation of code. It talks about how nesting must be done, how declaration of variables and functions must be done, how reusable components must be used and so on. Peer review must verify that the written code is properly adhering to optimisation guidelines.
- **Code Interpreting Design** Coding must interpret designs correctly. Coding files and what they are supposed to implement must be defined in low-level design. There must not be anything more or less than what has been defined in low-level design.
- **Unit Testing** Unit testing must be done by the developers to ensure that written code is working as expected. Sometimes, unit testing is done by peer of an author (of a code) to maintain independence of testing with respect to development. Unit test cases must cover valid as well as invalid conditions faced by users. Unit test case logs must be prepared and available for peer review, SQA review as well as customer audits. Defects in unit testing must be logged, and correction as well as corrective/preventive actions must be taken to close them.

7.8 VV MODEL

'V model' is also termed 'validation model' or 'test model' as it mainly considers only validation activities associated with software development which are popularly known as 'testing activities'. But, quality checking involves verification as well as validation activities. 'VV model' considers all the activities related to verification as well as validation. It is also termed 'Verification and Validation Model' or 'Quality model'.

'VV model' talks about verification and validation activities associated with software development during entire life cycle. Let us talk in brief about various activities associated with each phase of software development life cycle.

Requirements As requirements are obtained from customer using techniques described earlier (like joint application development, customer or market survey, and prototyping) there must be some arrangement for conducting requirement review. The intention would be to find if there is any gap existing between user requirements and requirement definition, and also to check whether all the requirements have been captured correctly and completely or not. Requirement review may concentrate on the structure of requirement specifications statement using a template and content of requirement specifications as per process definition.

Requirement Verification Generally requirement verification is done through inspection of requirement specification document using checklist, standards or guidelines. Experts in domain, customer representative, and other stakeholders may be involved in conducting such an inspection. Outcome of the inspection must be recorded and defects must be corrected before going further to next stages of development life cycle. Trade-offs, if any, must be documented. Many organisations have a checklist approach to fix the problems in requirement statement before it goes for inspection.

Requirement Validation Requirement validation takes place at two or more stages during software development. The first stage of validation involves writing complete use cases by referring to requirement statement. Any assumption made while writing use case may be a possible gap unless all stakeholders agree that they are implied requirements. Use cases must cover all the possible scenarios. The second stage of validation is through system testing. System test scenario and test cases are defined using requirement specifications. Requirement traceability through test scenario and test cases establish coverage of requirements.

Another requirement validation, though by review, happens when a customer reviews requirement specification and signs off as accepted. Anything signed by the customer as acceptance is considered 'validation'. Requirement validation also happens when customer conducts acceptance testing on the work product. Any defect (popularly called as anomaly) is recorded as a possible defect. One may have to refer to requirement statement to find whether reported defect is actually a defect or not. Sometimes, there may be a deliberate decision taken while creating requirement statement which is found during acceptance testing. This may not be considered as a defect.

Design Design may include high-level design or architectural design, and low-level design or detail design. Designs are created by architects (high-level designs)/designers (low-level designs) as the case may be.

Design Verification Verification of design may be a walkthrough of design document by design experts, team members and stakeholders of the project. Project team along with architect/designer may walkthrough the design to find the completeness and give comments, if any. Traceability of design with requirement must be established in requirement traceability matrix. Many organisations follow some specific tools or methodologies (like UML) to create designs. Use of any case tool must be validated so that defects introduced by tools are known beforehand.

Design Validation Validation of design can happen at two or more stages during software development life cycle. The first stage of validation happens when data flow diagrams can be created by referring to the design document. If the flow of data is complete, design is considered to be complete. Any interruption in flow of data indicates lacunae in design. This is also termed 'flow anomaly'. Sometimes, there is a circular reference in data flow which will lead to a loop which will never stop. Another option may be to create models or prototype to validate design. The second stage of validation happens at integration testing and interface testing. Integration testing is an activity to bring the units together and test them as a module. All units are created as per low-level design and tested in unit testing. Once it is confirmed that they work individually, they are integrated to form a module or system as the case may be. Interface testing is an activity of testing connectivity and communication of application with the outside world. Once the system is integrated as defined in earlier phase, one may have to test it for outside connections like database, browser, operating system, communication software, and user interface. Interface is defined by architectural designs.

Another design validation, though by review, happens when a customer reviews the design specification and signs off as accepted.

Coding Coding is an activity of writing individual units as defined in low level design. It is done by developers where low-level designs are implemented. At some places, database (including tables and stored procedures) is also handled by different teams.

Code Verification As coding is done, it undergoes a code review (generally peer review). Peer review helps in identification of errors with respect to coding standards, indenting standards, commenting standards, and variable declaration issues. Checklist approach is used in code review.

Code Validation Validation of coding happens through unit testing where individual units are tested separately. The developer may have to write special programs (such as stubs and drivers) so that individual units can be tested. The executable is created by combining stubs, drivers and the units under testing. This executable is tested with the test cases mainly derived from low-level design. Test results and defects are then logged.

Another code validation may happen, when customer reviews/permits unit testing of the code file and signs off as accepted.

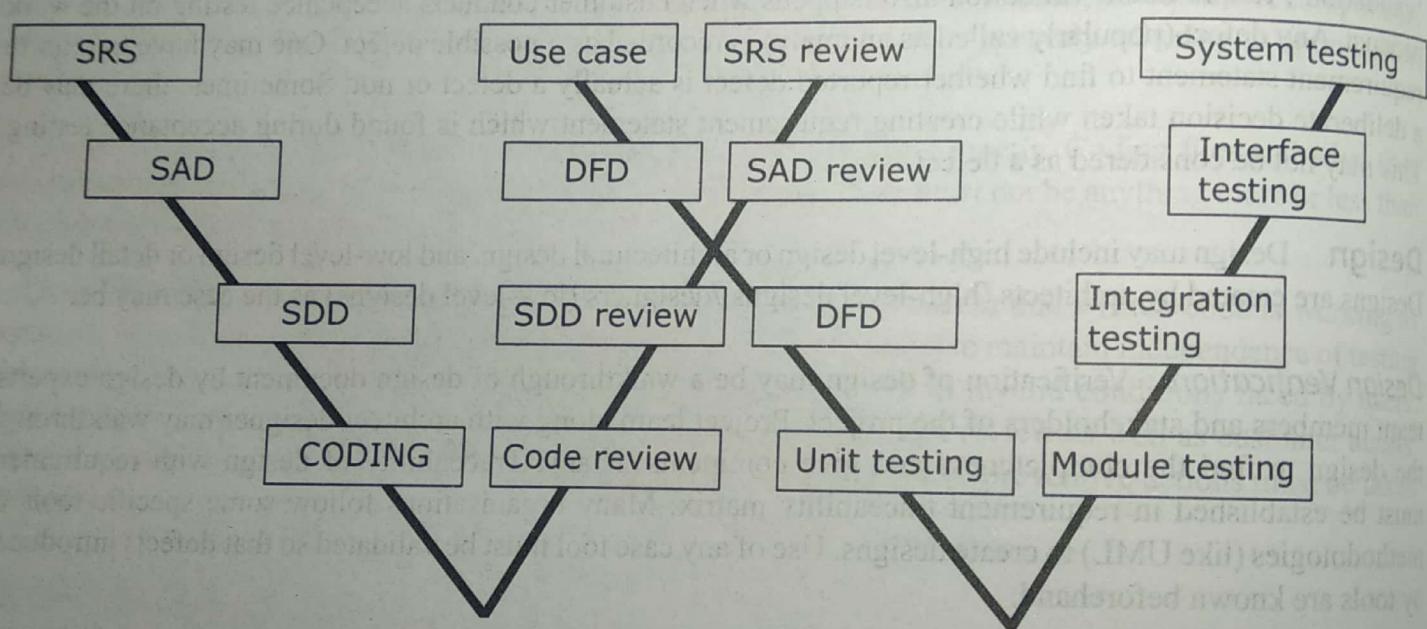


Fig. 7.2

VV model (Verification and Validation model)

7.9 CRITICAL ROLES AND RESPONSIBILITIES

In software verification and validation, three critical roles can be identified. It does not refer to different persons, but activities done when different roles are taken by those entities. Depending upon organisation size, maturity, and type of the project, these roles may be performed by different entities at different stages. There is a possibility of overlap between these roles, or gap between these roles. Gap indicates lacunae in organisation structure which must be fixed immediately. Let us try to define these roles and their responsibilities.

Development Development team may be comprised of various roles under them. They may be performing various activities as per their roles and responsibilities at different stages. These subroles may include development manager, project leads, module leads, team leads, developers, and unit testers as per organisation structure and project requirements. Some of the activities related to development group may be as follows

- Project planning activities include requirement elicitation, estimation, project planning, scheduling, and definition of quality attributes required by the customer. Project planning is the foundation of the project's success. Generally, senior people in development (such as project manager) may have the responsibility to create a project plan.
- Resourcing may include identification and organisation of adequate number of people, machines, hardware, software, and tools as required by the project. It may also involve an assessment of skills required by project, skills already available with them and any training needs of team. This may result into different plans such as training plan, procurement plan, etc.
- Interacting with customer and other stakeholders as per project requirements. The interactions may be to gather requirements, get sign offs for each phase, get queries resolved, or send deliverables to the customer.
- Defining policies and procedures for creating development work, verification and validation activities to ensure that quality is built properly, and delivering it to test team/customer as the case may be.
- Supporting testing team by acknowledging defects, giving inputs about requirements, giving executable on time, solving the queries raised by testing team or sending it to customer, as the case may be.
- Doing development-related activities such as capturing requirements, creating designs, coding, and implementation. It may also include quality control related activities such as reviews, walkthrough, etc.

Testing Testing team may include test manager, test leads, and testers as per scope of testing, size of project, and type of customer. Generally, it is expected that a test team would have independence of working and they do not have to report to development team. Roles and responsibilities of test team may include the following.

- Test planning including test strategy definition, and test case writing. Test planning may include estimation of efforts and resources required for testing.
- Resourcing may include identification and organisation of adequate number of people, machines, hardware, software, and tools as required by the project. It may also involve an assessment of skills required by project, skills already available with them and any training needs.
- Interacting with customer and other stakeholders as per project requirements. It may include asking queries, responding to customer/development team requests and so on.
- Defining policies and procedures for creating and executing tests as per test strategy and test plan. Testers may have to take part in verification and validation activities related to test artifacts.
- Supporting development team by providing adequate information about the defects. If required, testers may have to reproduce defects in front of customer/development team.
- Doing acceptance testing related activities such as training and mentoring to users from customer side before/during acceptance testing activities.

Customer Customer may be the final user group, or people who are actually sponsoring the project. Customers can be internal to an organisation or external to the organisation. Roles and responsibilities of a customer may include the following

- Specifying requirements and signing off requirement statement and designs as per contract. It may also include solving any queries or issues raised by development/test team.
- Participating in acceptance testing as per roles and responsibilities defined in acceptance test plan. A customer may be responsible for alpha, beta and gamma testing, as the case may be.

- Review and approve various artifacts as defined by contract or statement of work. A customer may have to participate in various reviews, walkthroughs, and inspection activities as defined.



Tips for verification and validation

- Understand verification and validation activities during software development and testing. Testing includes both verification as well as validation.
- Understand the roles and responsibilities of each activity related to verification and validation. They are always interrelated to each other.
- Understand 'V model' of validation as well as 'VV model' of verification and validation.

Summary



This chapter offers an elaborate discussion of 'V model' of software validation and 'VV model' of software verification and validation. We have also seen activities related to verification and validation, and roles and responsibilities of three critical entities in entire development. Understanding of these roles and responsibilities is crucial to derive a good product.

- 1) What are the characteristics of good requirements?
- 2) Describe V & V activities during proposal.
- 3) Describe V & V activities during requirement generation.
- 4) Describe V & V activities for test artifacts.
- 5) Describe V & V activities during designs.
- 6) Describe V & V activities during coding.
- 7) What are the roles and responsibilities of development group?



CHAPTER 9

LEVELS OF TESTING



OBJECTIVES

This chapter details verification and validation activities associated with various stages of software development life cycle starting from proposal. It also covers various ways of integration testing.



9.1 INTRODUCTION

As seen in 'V testing' approach earlier, testing is a life-cycle activity which starts at the time of proposal and ends when acceptance testing is completed and product is finally delivered to customer. Testing begins with a proposal for software/system application development/maintenance, and ends when

the system is formally accepted by user/customer. Different agencies/stakeholders are involved in conducting specialised testing required for the specific application. Definition of these stakeholders/agencies depends on the type of testing involved and level of testing to be done in various stages of software development life cycle. Table 9.1 shows in brief the level of testing performed by different agencies. It is an indicative list which may differ from customer to customer, product to product and organisation to organisation.

9.2 PROPOSAL TESTING

A proposal is made to the customer on the basis of Request for Proposal (RFP) or Request for Information (RFI) or Request for Quotation (RFQ) by the customer. Before making any proposal, the supplier must understand the purpose of such request, and devise the proposed solution accordingly. One must understand customer problem and the possible solution. Any proposal prepared in response to such request is reviewed by an organisation before sending it to the customer. It is reviewed by different panels or groups in the organisation such as technical group and commercial group. The intention is to ensure that the organisation must be able to stand by its words, if the proposal gets converted into a contract. The organisation must assess the impact of proposal on the organisation as well as on the prospect/customer.

Technical Review Technical review mainly involves technical feasibility of the kind of system or application proposed, the availability and requirement of skill sets, the hardware/software and other requirements of system. The proposal is also reviewed on the basis of time required for developing such system, and efforts required to make the proposal successful. The technical proposal is a description of overall approach, and not

Table 9.1

Different agencies involved at different levels of testing

Testing	Agencies involved
Proposal review	Customer, Business Analyst, System Analyst, Project Manager
Proposal testing	Customer, Business Analyst, System Analyst, Project Manager
Requirement review	Customer, Business Analyst, System Analyst, Project Manager, Project Leader, Test Leader
Requirements testing	Customer, Business Analyst, System Analyst, Project Manager, Project Leader, Test Leader
Design review	Customer, Business Analyst, System Analyst, Architect, Project Manager, Project Leader, Test Leader, Developer
Design testing	Customer, Business Analyst, System Analyst, Architect, Project Manager, Project Leader, Test Leader, Developer
Code review	Project Leader, Project Team, Customer
Test artifact review	Project Leader, Project Team, Customer, Tester, Test Leader
Unit testing	Project Leader, Project Team, Customer
Module testing	Project Leader, Project Team, Tester, Customer
Integration testing	Project Leader, Project Team, Tester, Customer
System testing	Project Manager, Test Leader, Tester, Customer
Acceptance testing	Project Manager, Tester, User/Customer

the final accepted approach solution/method and may undergo many iterations of changes as per discussions with customer. Requirement and estimations at the stage of proposal are not very specific but rather, they are indicative. It works on rough-cut methods, and estimations are ball-park figures that prospect may expect.

Commercial Review A proposal undergoes financial feasibility and other types of feasibilities involved with respect to the business. Commercial review may stress on the gross margins of the project and fund flow in terms of money going out and coming in the development organisation. Generally, total payment is split into installations depending upon completion of some phases of development activity. As different phases are completed, money is realised at those instances.

Several iterations of proposals and scope changes may happen before all the parties involved in Request for Quotation (RFQ) and proposal agree on some terms and conditions for doing a project.

Validation of Proposal A proposal sometimes involves development of prototypes or proof of concept to explain the proposed approach to the problem of the customer. One must define the approach of handling customer problem in the model or prototype.

In case of product organisation, the product development group along with marketing and sales functions decide about the new release of a product.

9.3 REQUIREMENT TESTING

Requirement creation involves gathering customer requirements and arranging them in a form to verify and validate them. The requirements may be categorised into different types such as technical, economical,

legal, operational and system requirements. Similarly, requirements may be specific, generic, present, future, expressed, implied, etc. The customer may be unaware of many of these requirements. The business analyst and system analyst must study the customer's line of business, problem and solution that the customer is looking for before arriving at these requirements. Assumed, implied or intended requirement is a gray area and may be a reason for many defects in the final product. Requirement testing makes sure that requirements defined in requirement specifications meet the following criteria.

- **Clarity** All requirements must be very clear in their meaning and what is expected from them. Requirements must state very clearly what the expected result of each transaction is, and who are the actors taking part in the transactions. Anything which hampers the clarity of requirements must be removed or clarified.

Illustration 9.1

Many requirement statements have words like 'Application must be user friendly' or 'Application performance must be fair'. Such requirements cannot be implemented as nobody knows the exact requirements.

One may have to raise queries to understand the meaning of user friendliness or fair performance. If possible such statements must have some numerical figures to illustrate them. Tester will be able to write exact test case on the basis of these numbers.

- **Complete** Requirement statement must be complete and must talk about all business aspects of the application under development. It must consider all permutations and combinations possible when application is used in production. Any assumption must be documented and approved by the customer.

Illustration 9.2

Sometimes requirement statement does not specify the transaction. If requirement statement says 'Enter valid Login and Password', then it is not a complete requirement as (it does not specify how one should enter valid Login and how to go to Password). It could be by 'Tab' or by clicking the mouse or it will automatically go to Password.

If any part of transaction requires an assumption, it indicates incomplete requirements. One has to go back to customer or business analyst/system analyst to get this clarified.

- **Measurable** Requirements must be measurable in numeric terms as far as possible. Such definition helps to understand whether the requirement has been met or not while testing the application. Qualitative terms like 'good', 'sufficient', and 'high' must be avoided as different people may attach different meanings to these words.

Illustration 9.3

Sometimes, requirement statement may states that—enter valid credential and click 'Login' to go to next page. It is not very clear how much time it will take to reach next page.

These can not be verified under usability as performance testing.

- **Testable** The requirement must help in creating use cases/scenario which can be used for testing the application. Any requirement which cannot be tested must be drilled down further, to understand what is to be achieved by implementing these requirements.
- **Not Conflicting** The requirements must not conflict with each other. There is a possibility of trade-off between quality factors which needs to be agreed by the customer. Conflicting requirements indicate possible problem in requirement collection process.
- **Identifiable** The requirements must be distinctively identifiable. They must have numbers or some other way which can help in creating requirement traceability matrix. Indexing requirement is very important.

Validation of Requirements Requirement testing involves writing use cases using requirement statement. No assumption is to be made while writing use cases from requirement statement. If complete use cases can be developed using the requirement statement, requirements can be considered as clear, complete, measurable, testable and not conflicting with each other. Each assumption indicates lacunae in requirement statement. One has to ask the customer whether the assumptions are correct or not. As per feedback given by the customer, requirement statements must be updated.

9.4 DESIGN TESTING

Once the requirement statement satisfies all characteristics defined above, the system architect starts with system high-level architectural designing. The designs made by system architects must be traceable to requirements. The system designer starts building the low-level or detail design with the help of architectural design. Low-level design must be traceable to architectural design which in turn is traceable to requirements. The design must also possess the characteristics given below.

- **Clarity** A design must define all functions, components, tables, stored procedures, and reusable components very clearly. It must define any interdependence between different components and inputs/outputs from each module. It must cover the information to and from the application to other systems.
- **Complete** A design must be complete in all respect. It must define the parameters to be passed/received, formats of data handled, etc. Once the design is finalised, programmers must do their work of implementing designs mechanically and system must be working properly.
- **Traceable** A design must be traceable to requirements. The second column of requirement traceability matrix is a design column. The project manager must check if there is any requirement which does not have corresponding design or vice versa. It indicates a defect if traceability cannot be established completely.
- **Implementable** A design must be made in such a way that it can be implemented easily with selected technology and system. It must guide the developers in coding and compiling the code. The design must include interface requirements where different components are communicating with each other and with other systems.
- **Testable** Testers make structural test cases on the basis of design. Thus, a good design must help testers in creating structural test cases.

Validation of Design Design testing includes creation of data flow diagrams, activity diagrams, information flow diagram, and state transition diagram to show that information can flow through the system completely. Where the flow gets interrupted, it indicates that design is not complete. Flow of data and information in the system must complete the loop. It must consider the data definitions, attributes and input/output formats. Another way of testing design is creating prototypes from design.

9.5 CODE REVIEW

Code reviews include reviewing code files, database schema, classes, object definitions, procedures, and methods. Code review is applied to ensure that the design is implemented correctly by the developers, and guidelines and standards available for the purpose of coding are followed correctly. Code must have following characteristics.

- **Clarity** Code must be written correctly as per coding standards and syntax, requirements for the given platform. It must follow the standards and guidelines defined by the organisation/project/customer, as the case may be. It must declare variables, functions, and loops very clearly with proper comments. Code commenting must include which design part has been implemented, author, date, revision number, etc so that it can be traced to requirements and design.
- **Complete** Code, class, procedure, and method must be complete in all respect. It must suffice the purpose for which it is created. One class doing multiple things, or multiple objects created for same purpose indicates a problem in design. Code must be readable and compilable. Proper indenting, and variable initialisation must be followed as defined by coding standards.
- **Traceable** Code must be traceable with design components. It must declare clearly about the requirements, design, author, date, etc. Code files having no traceability to design can be considered as redundant code which will never get executed even if design is correct.
- **Maintainable** Code must be maintainable. Any developer with basic knowledge and training about coding must be able to handle the code in future while maintaining or bug fixing it.

9.6 UNIT TESTING

Unit is the smallest part of a software system which is testable. It may include code files, classes and methods which can be tested individually for correctness. Unit testing is a validation technique using black box methodology. Black box testing mainly concentrates on requirements of the system. In unit testing, units are tested for the design in addition to requirements as low-level design defines the unit.

- Individual components and units are tested to ensure that they work correctly as an individual as defined in design
- Unit testing requires throwaway drivers and stubs as individual files may not be testable or executable without them
- Unit testing may be performed in debugger mode to find how the variable values are changed during the execution. But, it may not be termed 'black box testing' in such case as code is seen in debugging.
- Gray box testing is also considered as 'unit testing technique' sometimes as it examines the code in detail along with its functioning. Gray box testing may need some tools which can check the code and functionality at the same time.
- Unit test cases must be derived from use cases/design component used at lowest levels of designs.

9.6.1 DIFFERENCE BETWEEN DEBUGGING AND TESTING

Many developers consider unit testing and debugging as the same thing. In reality there is no connection between the two. Difference between them can be illustrated as shown in Table 9.2

Table 9.2

Difference between debugging and unit testing

Debugging	Unit testing
It involves code checking to locate the causes of defect	It checks the defect and not the causes of the defect
Code may be updated during debugging	It does not involve any correction of code
The test cases are not defined for debugging	Test cases are defined based on requirements and design
Generally covers positive cases to see whether unit works correctly or not	Covers positive as well as negative cases

9.7 MODULE TESTING

Many units come together and form a module. The module may work of its own or may need stubs/drivers for its execution, if the module cannot be compiled into an executable. If the module can work independently, it is tested by tester. If it needs stubs and drivers, developers must create the same and perform testing. Module testing mainly concentrates on the structure of the system.

- Module testing is done on related unit-tested components to find whether individually tested units can work together as a module or not.
- Module test cases must be traceable to requirements/design. Generally, module test cases are derived from low-level design.

9.8 INTEGRATION TESTING

Integration testing involves integration of units to make a module/integration of modules to make a system/integration of system with environmental variables if required to create a real-life application.

Integration testing may start at module level, where different units and components come together to form a module, and go upto system level. If module is self executable, it may be taken for testing by testers. If it needs stubs and drivers, it is tested by developers.

Though integration testing also tests the functionality of software under review, the main stress of integration testing is on the interfaces between different modules/systems. Integration testing mainly focuses on input/output protocols, and parameters passing between different units, modules and/or system. Focus of integration is mainly on low-level design, architecture, and construction of software. Integration testing is considered as 'structural testing'. Figure 9.1 shows a system schematically

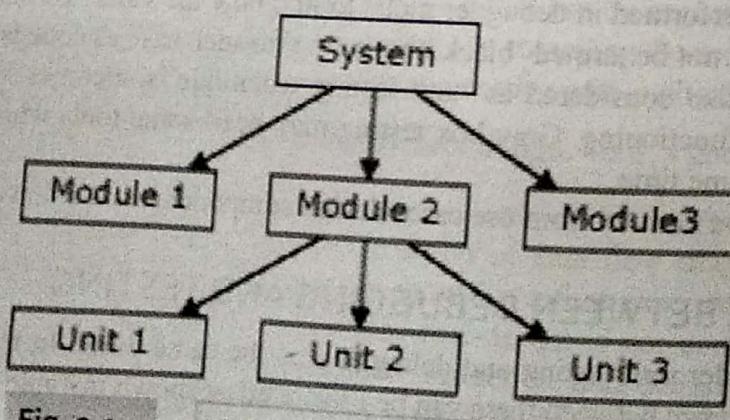


Fig. 9.1

Integration testing view

There are various approaches of integration testing depending upon how the system is integrated. Different approaches have different benefits and limitations.

9.8.1 BOTTOM-UP TESTING

Bottom-up testing approach focuses on testing the bottom part/individual units and modules, and then goes upward by integrating tested and working units and modules for system testing and intersystem testing. The process goes as mentioned below. Figure 9.2 shows a bottom-up integration and testing.

- Units at the lowest level are tested using stubs/drivers. Stubs and drivers are designed for a special purpose. They must be tested before using them for unit testing.
- Once the units are tested and found to be working, they are combined to form modules. Modules may need only drivers, as the low-level units which are already tested and found to be working may act as stubs.
- If required, drivers and stubs are designed for testing as one goes upward. Developers may write the stubs and drivers as the input/output parameters must be known while designing.
- Bottom-up approach is also termed 'classical approach' as it may indicate a normal way of doing things. Theoretically, it is an excellent approach but practically, it is very difficult to implement.
- Each component which is lowest in the hierarchy is tested first and then next level is taken. This gives very robust and defect-free software. But, it is a time consuming approach.

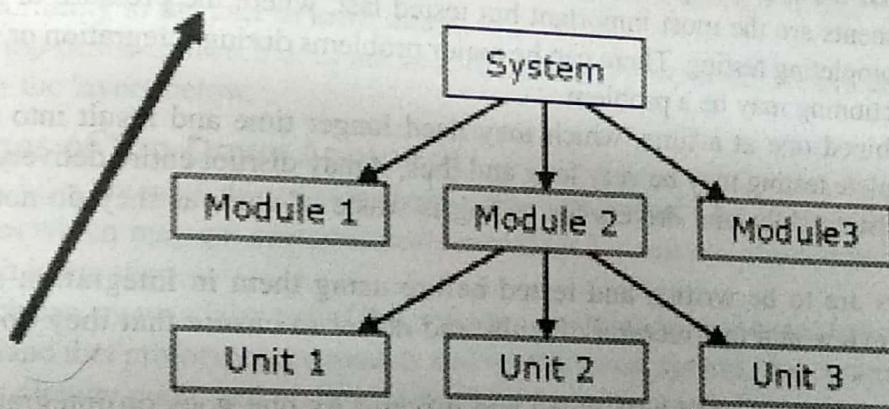


Fig. 9.2

Bottom-up integration approach

Bottom-up approach is suitable for the following.

- Object-oriented design where objects are designed and tested individually before using them in a system. When the system calls the same object, we know that they are working correctly (as they are already tested and found to be working in unit testing).
- Low-level components are general-purpose utility routines which are used across the application; bottom-up testing is the recommended approach. This approach is used extensively in defining libraries.

Stubs/Drivers

- Stubs/drivers are special-purpose arrangements, generally code, required to test the units individually which can act as an input to the unit/module and can take output from the unit/module

Stub Stub is a piece of code emulating a called function. In absence of a called function, stub may take care of that part for testing purpose.

Driver Driver is a piece of code simulating a calling function. In absence of actual function calling the program, driver tries to work as a calling function.

- Stubs are mainly created for integration testing like top-down approach. Drivers are mainly created for integration testing like bottom-up approach.
- Stubs/drivers must be very simple to develop and use. They must not introduce any defect in the application. Most of the times, they are software programs, but it is not a rule. Stubs/drivers must be taken away before delivering code to customer/user.
- Reusability of stubs/drivers can improve productivity, while designing and coding multipurpose stubs/drivers can be a big challenge.
- Estimation must consider a time required for creating stubs/drivers.

Advantages of Bottom-Up Approach

- Each component and unit is tested first for its correctness. If it found to be working correctly, then only it goes for further integration.
- It makes a system more robust since individual units are tested and confirmed as working.
- Incremental integration testing is useful where individual components can be tested in integration.

Disadvantages of Bottom-Up Approach

- Top-level components are the most important but tested last, where the pressure of delivery may cause problem of not completing testing. There can be major problems during integration or interface testing, or system-level functioning may be a problem.
- Objects are combined one at a time, which may need longer time and result into slow testing. Time required for complete testing may be very long and thus, it may disrupt entire delivery schedule.
- Designing and writing stubs and drivers for testing is waste of work as they do not form part of final system.
- Stubs and drivers are to be written and tested before using them in integration testing. One needs to maintain the review and test records of stubs and driver to ensure that they do not introduce any defect.
- For initial phases, one may need both stubs and drivers. As one goes on integrating units, original stubs may be used while large number of new drivers may be required (which are to be thrown at the end).

9.8.2 TOP-DOWN TESTING

In top-down testing approach, the top level of the application is tested first and then it goes downward till it reaches the final component of the system. All top-level components called by tested components are combined one by one and tested in the process. Here, integration goes downward. Top-down approach needs design and implementation of stubs. Drivers may not be required as we go downward as earlier phase will act as driver for latter phase while one may have to design stubs to take care of lower-level components which are not available at that time.

Top-level components are the user interfaces which are created first to elicit user requirements or creation of prototype. Agile approaches like prototyping, formal proof of concept, and test-driven development use this approach for testing. Figure 9.3 shows a top down integration and testing.

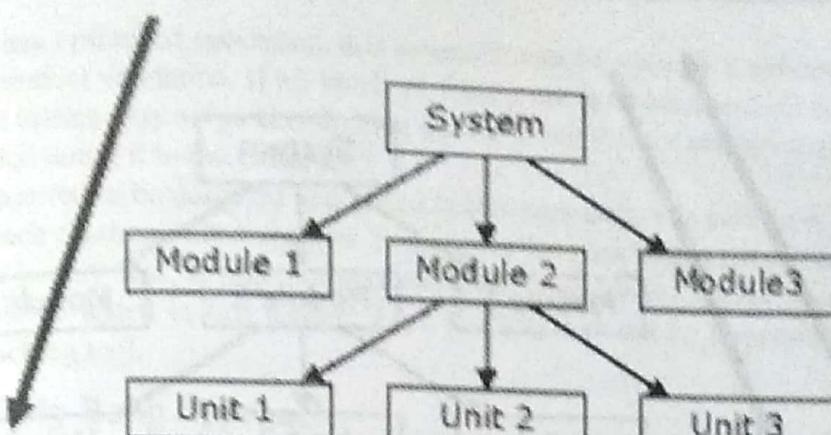


Fig. 9-3

Top-down integration approach

Advantages of Top-Down Approach

- Feasibility of an entire program can be determined easily at a very early stage as the topmost layer, generally user interface, is made first. This approach is good if the application has user interface as a major part.
- Top-down approach can detect major flaws in system designing by taking inputs from the user. Prototyping is used extensively in agile application development where user requirements can be clarified by preparing a model. If software development is considered as an activity associated with user learning, then prototyping gives an opportunity to the user to learn things.
- Many times top-down approach does not need drivers as the top layers are available first which can work as drivers for the layers below.

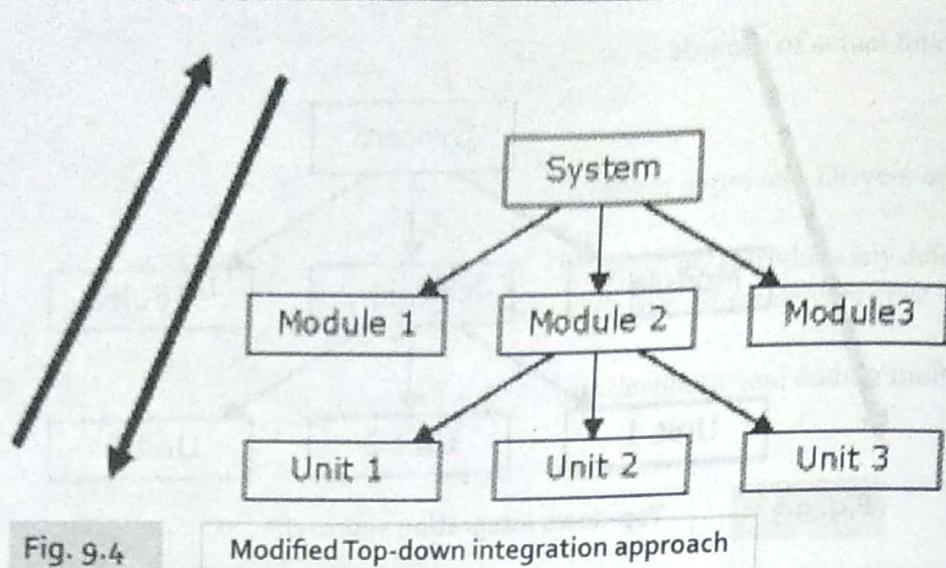
Disadvantages of Top-Down Approach

- Units and modules are rarely tested alone before their integration. There may be few problems in individual units/modules which may get compensated/camouflaged in testing. The compensating defects cannot be found in such integration.
- This approach can create a false belief that software can be coded and tested before design is finished. One must understand that prototypes are models and not the actual system. The customer may get a feeling that the system is already ready and no time is required for delivering it for usage.
- Stubs are to be written and tested before they can be used in integration testing. Stubs must be as simple as possible, and must not introduce any defect in the software. Large number of stubs may be required which are to be thrown at the end.

9.8.3 MODIFIED TOP-DOWN APPROACH

Modified top-down approach tries to combine the better parts of both approaches, viz. top-down approach and bottom-up approach. It gives advantages of top-down approach and bottom-up approach to some extent at a time, and tries to remove disadvantages of both approaches. Development must follow a similar approach to incorporate modified top-down approach. The main challenge is to decide on individual module/unit testing for bottom-up testing as normal testing may start from top.

Tracking and effecting changes is most important as development and testing start at two extreme ends at a time. Any change found at one place may affect another end. Care must be taken that the two approaches must meet somewhere in between. Configuration management is very important for such an approach. Figure 9.4 shows a modified top-down integration and testing.



Advantages of Modified Top-Down Approach

- Important units are tested individually, then combined to form the modules, and finally, the modules are tested before system is made. This is done during unit testing followed by integration testing.
- The systems tested by modified approach are better in terms of residual defects as bottom-up approach is used for critical components, and also better for customer-requirement elicitation as top-down approach is used in general.
- It also saves time as all components are not tested individually. It is expected that all components are not critical for a system.

Disadvantages of Modified Top-Down Approach

- Stubs and drivers are required for testing individual units before they are integrated (atleast for critical units). Stubs are required for all parts as integration happens from top to bottom.
- Definition of critical units is very important. Criticality of the unit must be defined in design. Critical unit must be tested individually before any integration is done.

This approach actually means testing the system twice or atleast more than once. The first part of testing starts from top to bottom as we are integrating units downward, and the second part from bottom to top for selected components which are declared as 'critical units'. This may need more resources in terms of people, and more time for test cycles than top-down approach but less time for test cycles than bottom-up approach. This approach is more practical from usage point of view as all components may not be equally important.

9.9 BIG-BANG TESTING

Big-bang approach is the most commonly seen approach at many places, where the system is tested completely after development is over. There is no testing of individual units/modules and integration sequence. System testing tries to compensate for any other kind of testing, reviews, etc. Sometimes, it includes huge amount of random testing which may not be repeatable.

Advantages of Big-Bang Approach

- It gives a feeling that cost can be saved by limiting testing to last phase of development. Testing is done as a last-phase of the development lifecycle in form of system testing. Time for writing test cases and defining test data at unit level, integration level, etc may be saved.

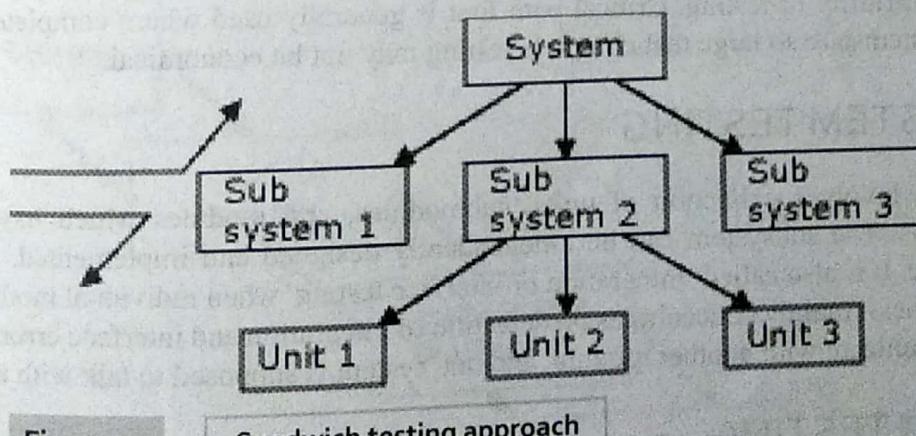
- If an organisation has optimised processes, this approach can be used as a validation of development process and not a product validation. If all levels of testing are completed and all defects are found and closed, then system testing may act as certification testing to see if there are any major issues still left in the system before delivering it to the customer.
- No stub/driver is required to be designed and coded in this approach. The cost involved is very less as it does not involve much creation of test artifacts. Sometimes test plan is also not created.
- Big-bang approach is very fast. It may not give adequate confidence to users as all permutations and combinations cannot be tested in this testing. Even test log may not be maintained. Only defects are logged in defect-tracking tool.

Disadvantages of Big-Bang Approach

- Problems found in this approach are hard to debug. Many times defects found in random testing cannot be reproduced as one may not remember the steps followed in testing at that particular instance.
- It is difficult to say that system interfaces are working correctly and will work in all cases. Big-bang approach executes the test cases without any understanding of how the system is build.
- Location of defects may not be found easily. In big bang testing, even if we can reproduce the defects, it can be very difficult to locate the problematic areas for correcting them.
- Interface faults may not be distinguishable from other defects.
- Testers conduct testing based on few test cases by heuristic approach and certify whether the system works/does not work.

9.10 SANDWICH TESTING

Sandwich testing defines testing into two parts, and follows both parts starting from both ends i.e., top-down approach and bottom-up approach either simultaneously or one after another. It combines the advantages of bottom-up testing and top-down testing at a time. Figure 9.5 shows a sandwich testing approach.



9.10.1 PROCESS OF SANDWICH TESTING

- Bottom-up testing starts from middle layer and goes upward to the top layer. Generally for very big systems, bottom-up approach starts at subsystem level and goes upwards.
- Top-down testing starts from middle layer and goes downward. Generally for very big systems, top-down approach starts at subsystem level and goes downwards.

- Big-bang approach is followed for the middle layer. From this layer, bottom-up approach goes upwards and top-down approach goes downwards.

9.10.2 ADVANTAGES OF SANDWICH TESTING

- Sandwich approach is useful for very large projects having several subprojects. When development follows a spiral model and the module itself is as large as a system, then one can use sandwich testing.
- Both top-down and bottom-up approaches start at a time as per development schedule. Units are tested and brought together to make a system. Integration is done downwards.
- It needs more resources and big teams for performing both methods of testing at a time or one after the other.

9.10.3 DISADVANTAGES OF SANDWICH TESTING

- It represents very high cost of testing as lot of testing is done. One part has top-down approach while another part has bottom-up approach.
- It cannot be used for smaller systems with huge interdependence between different modules. It makes sense when the individual subsystem is as good as complete system.
- Different skill sets are required for testers at different levels as modules are separate systems handling separate domains like ERP products with modules representing different functional areas.

9.11 CRITICAL PATH FIRST

In critical path first, one must define a critical path which represents the main function of a system, generally represented by P1 requirements or P1 functionalities of an application. Testing defines the critical part of the system which must be covered first. Development team concentrates on design, implementation and testing of critical path of a system first. This is also termed 'skeleton development and testing'. It is applied where critical path of a system is important for system performance and for business from customer's perspective. One must understand the criticality of system functions from user's perspective or from business perspective, and decide on the priority of testing. Critical path first is generally used where complete system testing is impossible and systems are so large that complete testing may not be economical.

9.12 SUBSYSTEM TESTING

Subsystem testing involves collection of units, submodules, and modules which have been integrated to form subsystems. The subsystem can be independently designed and implemented, and also can be a separate executable. It is also called 'integration or interface testing' when individual modules are as good as independent systems. It mainly concentrates on detection of integration and interface errors where one unit is supposed to communicate with another module, and one system is supposed to talk with another system.

9.13 SYSTEM TESTING

System testing represents the final testing done on a system before it is delivered to the customer. It is done on integrated subsystems that make up the entire system, or the final system getting delivered to the customer. System testing validates that the entire system meets its functional/nonfunctional requirements as defined by the customer in software requirement specification. The criteria for system testing may involve an entire domain or selected parts depending upon the scope of testing. Generally system testing goes through the following stages.

- *Functional Testing* Functional testing intends to find whether all the functions as per requirement definition are working or not. Generally, any software is intended for doing some functions which must be

defined in requirement statement. Once functions are tested, all defects related to functions must be fixed to make the system correct.

User Interface Testing Once the functionalities are set correct, the next step is to set the user interface correct (if the system has a user interface). User interface testing may involve colors, navigations, spellings, and fonts. Sometimes, there can be a thin line between functional testing and user interface testing, and one may take a decision depending upon the situation. There are few systems where there is no or very minimal user interface. In such cases, user interface testing may not be applicable.

Once the system is tested for functionalities and user interface, it is ready to go for other types of testing such as security, load, and compatibility. It depends upon the scope of testing, type of system under testing and which types of testing are involved as a part of system testing. If the scope does not include functionality testing, it is not required and one may directly go to user interface testing.

9.14 TESTING STAGES

Generally an organisation performs unit testing as first part of testing. The inputs from unit testing are used to identify and eliminate defects at unit level so that they will not go further down. Once unit testing is over and units are ready for integration, then integration or module testing is done. Module testing may be done by development/test teams depending upon the requirements of stubs and drivers. After module testing, the system goes for subsystem testing, which is also considered as 'integration testing'. Once the system completes all levels of integration, either it goes for system testing, or interface testing and then system testing as the case may be. Interface testing addresses the issues associated with communication of a system with another system that already exists or is expected to exist in future.

Once the system is through with system testing, it is taken for acceptance testing to check whether it fulfils the acceptance criteria or not. The stages of testing are graphically shown in Fig. 9.6.

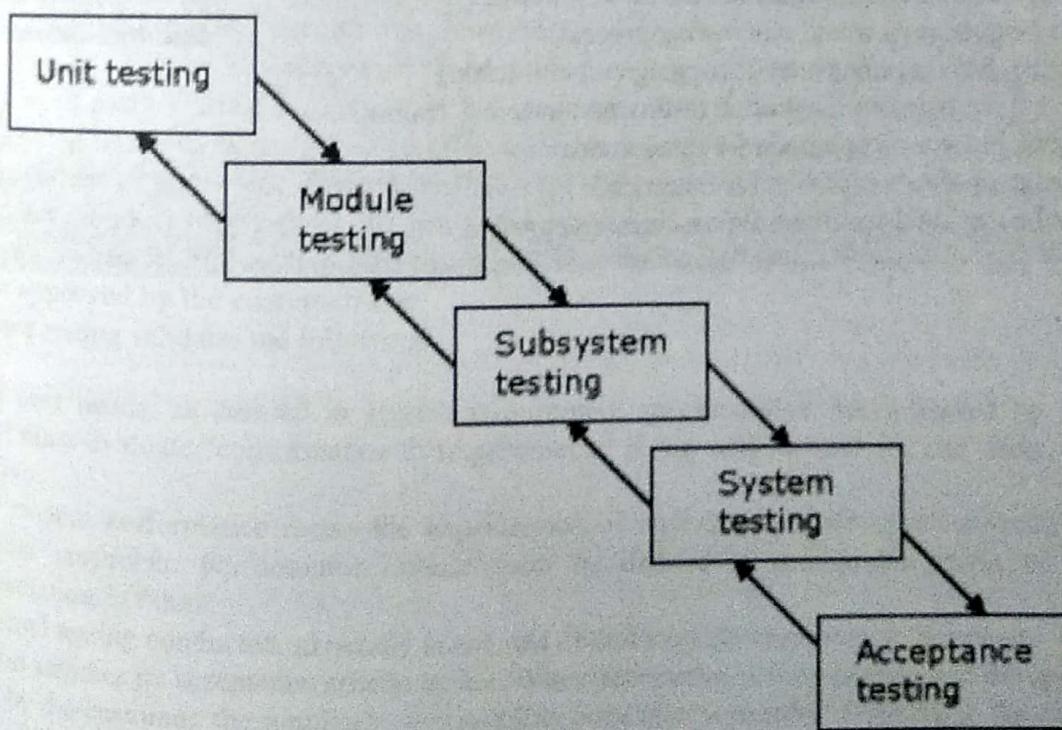


Fig. 9.6

Software testing stages

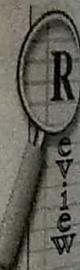
 **Tips for life-cycle testing**

- Identify various stages of software development life cycle, and verification and validation activities related to these stages. The ultimate aim of all activities is to reduce the defects going to the customer.
- Identify the integration approach designed for the application under testing.

Summary

This chapter details verification and validation activities related to various stages of software development phases. It gives advantages and disadvantages of the various approaches of integration testing.

- 1) Describe proposal review process.
- 2) Describe requirement verification and validation process.
- 3) Describe design verification and validation process.
- 4) Describe code review and unit testing process.
- 5) Describe difference between debugging and unit testing.
- 6) Differentiate between integration testing and interface testing.
- 7) Describe bottom-up approach for integration.
- 8) Describe top-down approach for integration.
- 9) Describe modified top-down approach for integration.



CHAPTER 11

SPECIAL TESTS (PART I)



OBJECTIVES

This chapter covers special techniques which are other than system testing. They may be required as per specific requirements of the customer or a specific application.



11.1 INTRODUCTION

Testing is wrongly considered as only functionality testing by many people. Though functionalities are very important for a software application, yet testing is much more than testing functionalities. Testing may include some specialised testing methods and techniques as required by user/customer,

or as appropriate for the application under testing. The special testing must be included in a scope statement for testing and in the requirement specifications defined by the customer or business analysts, and system analysts, as the case may be. Such testing may be termed 'special testing' as it may or may not be done, in general, for any application. These types of testing may be done on and above normal approaches of unit testing, integration testing, system testing through functional testing depending upon the specific requirement of a customer. Most of these types of testing are done at system level, though it is not a rule. Special testing may need some special testing skills, tools and techniques.

11.1.1 SPECIALISED SYSTEMS AND APPLICATIONS

Testing of some specialised systems or special user requirements is included in this type of testing. These systems may be made and used for a special purpose, and need to have some special characteristics with respect to the definition in requirement statement. Examples of such system may be eBusiness systems, security systems, administration system, antivirus applications, operating systems and databases. These systems are designed for specialised customers and users, and may or may not be intended for general category of people.

Let us understand few testing methods falling under the category of specialised testing.

11.2 COMPLEXITY TESTING

Complexity testing is a verification technique where complexity of system design and coding is verified through reviews, walkthroughs or inspections as per planned arrangement. It is used in the following.

- Customer has some specific requirement for complexity measurements. Complex programs are difficult to maintain in future. More complex code is difficult to maintain as well as it may introduce few defects when complex decisions are executed. Black box testing also becomes complicated when code is very complicated.
- Applications have major algorithms and decision loops, and programmers may become victims of complexity while handling these decisions. Optimisation, simplicity and complexity must be balanced by designer and implemented by developers.
- Complexity leads to complex testing, and permutations and combinations of system increase infinitely. It may be possible that complex decisions may give wrong outputs.

Complexity may be measured by different ways and methods. Cyclomatic complexity measures the amount of decision logic in a single software unit. It is used for two related purposes in the structured testing methodology.

- It gives the number of recommended tests for software. More decisions taken by the systems, and more nesting of the decisions may mean more number of tests to be performed to ensure adequate coverage.
- It is used during all phases of the software life cycle development (beginning with design) to keep software reliable, testable and manageable. Complex written code and complex designs are more susceptible to failure in complex conditions.

Cyclomatic complexity is based entirely on the structure of the softwares control flow graph.

11.2.1 CONTROL FLOW GRAPHS

Control flow graphs describe the logic structure of software unit, where decisions can go while executing instructions. Each flow graph consists of nodes and edges, and each decision may lead to several control flows in the application. The nodes represent statements or expressions, and the edges represent transfer of control between different nodes depending upon the condition it faces. When decisions are encountered, edges increase to more extent than the nodes, and represent complexity of the code.

Each possible execution path of a software module has a corresponding path from entry to exit node of the unit's control flow graph. This correspondence is the foundation for the structured testing methodology. Nodes do not create much problem in testing as the flow remains unidirectional, while condition increases complexity by adding more edges than the nodes.

11.2.2 DEFINITION OF CYCLOMATIC COMPLEXITY

Cyclomatic complexity is defined for each unit to be $[e - n + 2]$, where 'e' represents number of edges and 'n' represents number of nodes in the control flow graph, respectively. More decisions add more edges than the nodes, and increase complexity of software. Sometimes, there is no option but to add the decisions if application requirements demand the complexity. Designers must try to reduce complexity as minimum as possible.

11.2.3 LIMITING CYCLOMATIC COMPLEXITY

There are many good reasons to limit cyclomatic complexity. Overly complex modules are more prone to errors in designing, coding and implementation as well as testing. Complex coding is hard to understand, implement and maintain. It is also hard to test the software as there are multiple paths possible. It is hard to modify design and coding when defects are found, as there are complex dependencies due to various decisions.

11.2.4 MONOLITHIC COMPLEXITY

Cyclomatic complexity mainly concentrates on number of controls which affect complexity of application. Even if somebody tries to control cyclomatic complexity, another challenge would be to avoid monolithic code. One should try to avoid coding in monolithic way. Object-oriented programming helps in creating objects separately than the code, so that they can be used as and when required, in the application.

11.2.5 ADVANTAGES OF SMALL CODE

- One may create the objects or functions which may be called again and again as required. If the object functions correctly at one place, there is higher probability that it will work correctly at other instances. There is always a problem of inheritance and polymorphism.
- It avoids huge coding, and coding/development efficiency improves. One may have to refer to objects and libraries already created.
- Testing becomes easier, as objects may not need testing again and again. If the defect is found in object at one place, it would be found at all other places.
- Reusability reduces code size and inventing the wheel again and again, can be reduced.

11.2.6 DISADVANTAGES OF SMALL CODE

- Over reliance on objects can create problems in software, as people may not have good knowledge about the objects. In maintenance, if there is no good documentation available, then people will not have any clue of what a particular object does.
- Change in object may affect the entire application. At few places, it may create problems as these changes will be directly applied at all places where the objects are used.
- Object maintenance is a big challenge as changes in object may affect the entire application.

11.3 GRAPHICAL USER INTERFACE TESTING

Graphical user interface is the most important part of the application along with functionality, as it may have effect on usability. Generally, application system testing starts with functionality testing. It is followed by graphical user interface testing. Graphical user testing is also known as 'GUI' testing or 'UI' testing. Other than system type of software, most of the applications have user interface from where a user interacts with the system.

Graphical user interface includes the following.

All colors used for background, control colors and font color have a major impact on users. The user must be able to identify entities on the screen correctly and efficiently. Wrong color combinations and bright colors may increase fatigue of users.

All words, fonts, and alignments used on the screen which would be read every time when a user is interacting with an application. Very small or very large font, captions not aligned properly, or spelling mistakes can create a negative image about the application, in the mind of a user. Scrolling pages up and down as well as navigations for different hyperlinks and pages must be avoided as much as possible. More scrolling reduces usability, and users may get frustrated. If there are multiple pages, then locations of similar controls must be at logical locations, and consistency must be maintained in page layout.

Error messages and information given to users must be usable to the user. Messages must guide users to perform the correct actions. Very long or very small messages should be avoided as much as possible. Messages must be meaningful.

Reports and outputs produced, either on screen or printed by user should consider the readability issue, font, size of screen, and paper size on printer.

Screen layout in terms of number of instructions to users, number of controls and number of pages are defined in low-level design. More controls on a single page and more number of pages reduce usability of an application.

- Types of controls on a single page are very important from usability point of view. Provision of drop down controls in place of free text improves operability of application. Logical placement of controls and maintenance of tab sequence improves usability.
- Number of images on a page or moving parts on screen can hamper performance of an application. Pages must be as simple as possible for easy loading/unloading of pages in a web application.

Graphical user interface defects are generally considered as high-priority defects. It has direct relationships with usability testing, look and feel of an application. It affects the emotions of users and can improve acceptability of an application.

11.3.1 ADVANTAGES OF GUI TESTING

- Good GUI improves look and feel of the application. It helps in psychological acceptance of the application by the user.
- GUI represents a presentation layer of an application. Prototyping is used extensively for clarifying GUI requirements. Good GUI helps an application due to better experience of the users.
- Availability of 'help' and usefulness of 'help' routines can be ensured so that user can access 'help' in case of problems. Preventive controls help operator in doing things in right way without much trials.
- Consistency of screen layouts and designs improves usability of an application. Tab sequence provides a logical way of doing the sequence.

11.3.2 DISADVANTAGES OF GUI TESTING

- When the number of pages is large and number of controls in a single page is huge, it creates problem in testing. 'Looking but not seeing' phenomenon may be witnessed in GUI testing, where spelling mistakes may go unnoticed due to fatigue on part of the tester.
- Special applications testing like those made for blind people or kids below age of five may need special training for testers, as they have to behave like target users. Entire interface concepts may change due to change in intended users. GUI in terms of images shown on the screen like medical and diagnostic application may be very difficult to test, if testers do not have sufficient domain expertise.
- Testers may give more importance to functionalities than GUI testing. Defects in GUI are considered as cosmetic defects and neglected many times.

- Often, GUI testing is considered as low level of testing and given to junior testers. Importance of GUI may not be recognised by test team and development team.

11.4 COMPATIBILITY TESTING

Any system is made of many components. In case of software system, there may be components such as different types of machines, printers, hardware, different supporting softwares such as operating systems, databases and communication systems. As the world is growing, there are many possibilities of these components being present or absent in the system, and many more new inventions in technologies are possible. When an organisation develops software that works' on few components only, it automatically restricts the size of the market where that product can be sold. Also, if the product cannot work with some other variables in the user environment, then it restricts the market. Any product manufacturer will aim to create a situation where the product being developed must be working on all possible scenarios of these components. Compatibility testing refers to testing the software on multiple configurations to check the behaviors of different system components and their combinations.

The variables can be,

- Operating systems
- Databases
- Browsers
- Languages

The hardware can be,

- Machines and servers
- Routers
- Printers

Integration with other communication systems can be,

- Mailing softwares
- Messaging softwares

Different languages used across the globe can be,

- Chinese, Korean, or Japanese
- Hindi, Urdu, or Hebrew
- English, German, or French

This is not an exhaustive but an indicative list, and one may add many things to it. Compatibility testing is performed to ensure that the application functions properly on multiple system configurations which are possible at user end. More compatibility helps in increasing the market size and number of customers for product organisations. But it also affects the product in a negative manner, as the system becomes more fragile. Every component may have some specific requirements or input/output criteria, and manufacturer may have to devise a plan to meet them to ensure compatibility. Matching more and more criteria can be a challenging task as designs and implementation becomes very complex. Let us consider compatibility testing

in the form of multiplatform testing as an example. Similar logic can be extended to other compatibility issues like hardware compatibility, browser compatibility, etc.

11.4.1 MULTIPLATFORM TESTING

Multiplatform testing is one set of testing in compatibility testing where the system is expected to work with various platforms. Multiplatform testing involves testing of software on different platforms to ensure that their performance is not affected in a negative manner, as the platform is changed. There are various platforms available in the world which get updated frequently as technology changes. Platforms can be of generic nature or they can be proprietary ones, created for specific purposes. They may be freeware, open source or licensed.

From business perspective of a product development organisation, if the application works on fewer platforms, then it automatically restricts the number of customers or users using these platforms. If the customer has a platform, and various systems are already running using that platform, then the customer will prefer to use applications running on the existing platform. Customer will not change the platform for getting a new application. Product organisations will always prefer to have maximum possible platform coverage so that they can have customers from all corners of the world.

- Multiplatform testing involves verification and validation activities to ensure that software does its intended function in the same/similar way when the platform is changed. Validation across different platforms can be done in terms of parallel testing, and verification can be done by comparing the results of processing on different platforms.
- Generally, there is one platform which may be referred to as 'base platform' or 'mother platform' on which development is done or targeted, while it is expected to behave normally on some range of platforms other than this base platform.
- The range of platforms on which the application must work will be defined very clearly in requirement statement. The platform must be in existence so that testing on these platforms can be performed. Future expected platforms cannot be covered in multiplatform testing.
- Multiplatform testing ensures that software will perform in the same manner regardless of the platform on which it is working. The range of performance variations allowable, as we change from one platform to another, must be mentioned in requirement statement.

11.4.2 MAJOR CONCERN IN MULTIPLATFORM TESTING

Multiplatform testing has some limitations. These can be declared as the concerns of compatibility testing on multiple platforms. The major concerns in multiplatform testing are as follows.

- Platforms used for testing may not be platforms in operations when the application is put into real use. As the platform vendors keep on updating the platform by releasing various patches, service packs and hot fixes, the compatibility tests done under some configuration may not hold true when the platform itself is changed by these updates.
- One needs to define the platforms to be included in multiplatform testing as exhaustive compatibility with all platforms in the world may not be possible or feasible. The list of platforms cannot be infinite. It can be possible in reality that software may be deployed on the platform, which is not included in testing.
- Sometimes, the application manufacturer does a cost-benefit analysis to analyse the targeted market and the type of configuration used by this market, and defines few platforms where the application is tested for compatibility. It may not be feasible, nor is it required to support software on all the platforms existing in the world, as target market may not need it. Thus, the support may not be comprehensive.

- List of needed platforms and configurations may not be complete. Platforms may have different variations or flavors or configurations such as language settings, which may have an effect on application performance on different configurations. One has to make some assumptions as exhaustive testing may not be feasible or required.
- The application performance may be governed by platform configuration. As the platform configuration changes, the performance of an application running on it may get affected to some extent.

11.4.3 PROCESS OF SOFTWARE TESTING ON MULTIPLE PLATFORMS

Multiplatform testing involves a set of the following activities, in general. It may change to some extent depending upon the scope of testing, platforms and application under testing.

- Define the base platform which will be used as a basis for definition of success or failure of the application on targeted platform. Normal testing is done on the base platform as per definition of system testing. During compatibility testing, behavior of an application on the base platform is considered as the expected result. Actual results on targeted platform are compared with base platform results, and any deviation is considered as a defect.
- Prepare the list of platforms on which the application is targeted to be used. This must be defined and/or approved by the customer because it involves effort and time to conduct compatibility testing. Since customer may perform cost-benefit analysis, all the targeted platforms may not be included, or the complete set of testing may not be done on each of these platforms.

If there are three platforms (say PA, PB and PC) and a base/mother platform (say PM), then the customer may consider the probability of these platform distributions in the target market. The target market has a distribution as shown in Table 11.1.

Table 11.1

Multiplatform testing coverage

Platform targeted	Distribution	Testing coverage
PM	—	Base platform, system testing
PA	80%	100%
PB	15%	25%
PC	5%	Smoke testing

It is governed by risk analysis done by the customer. Generally, primary platform is fully covered while tertiary may only be smoke tested.

- Assess test laboratory configuration with respect to base platform and targeted platform description as defined in requirement statement. It must include the service packs or hot fixes to be included/excluded in testing and configurations of the system which can affect the application.
- Generally, an application interacts with the platform for performing various service tasks such as display and printing. The customer or designer must define the interfaces between the application and platform. List of interface items in the application that are being affected by platform change may define the extent of testing to be done. This must be done with the help of architectural design to understand interfaces between platform and application.

- List the interface platform effects that can help in determining which tests are must, to ensure that the connections are established correctly. If these interfaces are not matched correctly, it may affect the functionalities of the application adversely. Changes in parameter when the platform gets updated or changed must be noted to allow the specific test cases to be executed.
- Execute tests, as defined by the test plan on various platforms. The test cases may be selected depending upon the coverage offered or expected by the customer, risk analysis, and cost-benefit analysis done previously.
- Compare the results of an application performance on the targeted platform with the performance of an application on mother platform. Any deviation between two platforms acts as a probable defect.

11.4.4 TYPES OF COMPATIBILITY

For multiplatform testing, there are three main types of compatibilities possible. There can be various compatibilities between these three extreme ends.

- *Friend Compatibility* Friend compatibility happens when the application behavior on new platform is as if it is working on its base platform. There is no effect of change in the platform on the behavior of an application. The application utilises all the facilities and services available in the given platform efficiently and functioning is optimised. It may be a highly desirable state but the application becomes heavyweight, if we provide this compatibility with all possible configurations and components. One has to do cost-benefit analysis to determine how much friend compatibility is required, and how much is the minimum acceptable to the user.
- *Neutral Compatibility* The application behavior on new platform is similar to its working on parent platform. Only difference is that the application does not use the facilities provided by new platform at all, and behavior may be little bit slower. Sometimes, the application has its own utilities and services, and uses them as if nothing has been provided by any platform. This is termed 'neutral compatibility'. This type of compatibility may overcome the issue associated with changing platforms. But, it makes the system very heavy as it needs to have all the facilities required by the application and provided by the platform.
- *Enemy Compatibility* Here, the application does not perform as expected on new platform similar to its base platform, or it does not perform at all when put on targeted platform. Some functionality may be affected adversely, if application is not compatible with the targeted platform. This may be termed 'enemy compatibility'.

11.4.5 INTERNATIONALISATION

Software systems made in one part of the world may be used in many other parts. The usage of a system depends upon its ability to suite the environment available in other parts of the world.

Illustration 11.1

If a software has a user interface in English, it may be used by people who understand English. For others, such software is of no use as they cannot read English and hence, they are not able to interact with the system. If the manufacturer of the system wishes to have an international market, then internationalisation of the system becomes mandatory.

Internationalisation does not stop at language level but there are many other aspects to be controlled when a system is internationalised. In computing, internationalisation and localisation are means of adapting computer software to different languages and regional differences. Internationalisation is the process of designing a software application so that it can be adapted to various languages and regions, without making any engineering changes in the system. Localisation is the process of adapting software for a specific region or language by adding locale-specific components and translating text.

Due to their length, the terms 'Internationalisation' and 'Localisation' are frequently abbreviated to 'i18n' (where 18 stands for the number of letters between the 'i' and the 'n' in internationalisation, a usage coined at DEC in the 1970s or 80s) and 'L10n' (the capital 'L' on 'L10n' helps to distinguish it from the lowercase 'i' in 'i18n') respectively.

Some companies use the term 'globalisation' for the combination of internationalisation and localisation. Globalisation can also be abbreviated to just 'g11n'.

Scope of Internationalisation When a system is expected to work in a different language/region setting, one may have to decide the coverage. Focal points of internationalisation and localisation efforts may include the following.

- Language selected for original implementation and number of languages where application will be used are listed
- Computer-encoded text are included
- Alphabets/scripts—most recent systems use the Unicode standard to solve many of the character encoding problems during development
- Different systems of numerals such as Roman numerals and non-Roman numerals are identified
- Writing direction, e.g., left to right in English and right to left in Arabic is understood
- Spelling variants for different countries where the same language is spoken but spellings are different, e.g., localization (en-US) vs. localisation (en-GB) and colour (en-GB) vs. color (en-US) are considered
- Different words used for same entity, e.g., mailman (en-US) vs. postman (en-GB) and anticlockwise (en-GB) vs. counterclockwise (en-US) are considered
- Text processing differences, such as the concept of capitalisation which exists in some scripts and not in others, different text sorting rules, etc. are applied to application
- Graphical representations of text (printed materials and online images containing text) with regional differences are noted
- Spoken (audio) part of application must be handle
- Subtitling of film and video when the direct translation is required
- Cultural issues are to be understood
- Images and colors—Issues of comprehensibility and cultural appropriateness are considered
- Names and titles as used in different cultures and regions are considered
- Government assigned numbers (such as the Social Security number in the US, National Insurance number in the UK, and Isikukood in Estonia) and passports are considered
- Telephone numbers, addresses and international postal codes are considered
- Currency (symbols and positions of currency markers) in different regions are considered
- Weights and measures in different parts of world are noted
- Paper sizes are considered
- Writing conventions such as grammar differences for different regions is considered
- Date/time format, including use of different calendars at different places
- Time zones and their changes such as EST to EDT etc.

- Formatting of numbers (decimal points, positioning of separators, and character used as separator) in different regions are considered

Any Other Aspect of the Product or Service That Is Subject to Regulatory Compliance The distinction between internationalisation and localisation is subtle but important from the usage point of view. Internationalisation is the adaptation of products for potential use virtually everywhere, while localisation is the addition of special features for use in a specific locale. Internationalisation is done once per product, while localisation is done once for each combination of product and locale. The processes are complementary, and must be combined to lead to the objective of a system that works globally. Subjects unique to localisation may include the following.

- Language translation
- National varieties of languages where locale culture comes into picture
- Special support for certain languages such as East Asian languages (Chinese, Japanese, and Korean need double byte characters)
- Local customs and beliefs
- Local contents
- Symbols used locally
- Order of sorting followed in different regions
- Aesthetics expectations
- Cultural values and social context for different regions

Development of Internationalisation Systems The current prevailing practice is for applications to place text in resource strings which are loaded during program execution as needed, as per the language setting for particular region. These strings, stored in resource files, are relatively easy to translate. Programs are often built to reference resource libraries depending on the selected locale data.

Thus, to get an application to support multiple languages, one would design the application to select the relevant language resource file at runtime. Resource files are translated to the required languages. This method tends to be application-specific and at best, vendor-specific. The code required to manage data entry verification and many other locale-sensitive data types also must support differing locale requirements. Modern development systems and operating systems include sophisticated libraries for international support of these types.

Difficulties in Developing Internationalisation Systems While translating existing text to other languages may seem easy, it is more difficult to maintain the parallel versions of texts throughout the life of the product. For instance, if a message displayed to the user is modified, all of the translated versions must be changed accordingly to reflect the changes. This, in turn, results in somewhat longer development cycle.

Many localisation issues (e.g. writing direction and text sorting) require more profound changes in the software than text translation. To some degree, the development team needs someone who understands foreign languages and cultures, and has a technical background about the system and usage.

11.4.6 TESTING OF INTERNATIONALISATION

Testing process of internationalisation system is more parallel to multiplatform testing. Only difference is that in place of different platform, system is tested in different language setting/regional setting. System

development vendor must have a list of languages/cultures where testing must be done. Salient features of testing of internationalisation are as follows.

- Functional testing is a primary issue for any system, and internationalisation may also follow the same route as that of multiplatform testing. System may be set in mother language/culture setting and targeted language/culture setting. Whatever is the behavior on the mother setting is considered as expected results, and actual results are compared with them. Any deviation may be considered as a defect.
- User interface testing is very important for internationalisation testing. Whatever is shown by the system on the screen, printed on a printer, etc. must follow the norms of language/culture.
- Width required for different captions must match with the standards defined. Generally, targeted languages/cultures are listed and the longest string in each for a particular word is calculated. Controls must be capable of accepting the specific length. Another way can be where the control sizes are changed at run time as per size of the string.
- Screen/print layout must match with region/country/language specific requirement such as right to left, left to right, comma usage and decimal point usage. As the languages/conventions change, there must be appropriate change in layouts.
- Colors, pictures, and maps must be appropriate as per religion, language, culture, etc. It should not annoy or offend the users by showing something which is not acceptable as per faith of these users.

11.5 SECURITY TESTING

Security testing is a special type of testing intended to check the level of security and protection offered by an application to the users against unfortunate incidences. The incidences could be loss of privacy, loss of data, etc. The application is checked for the possible perpetrators which can affect the system adversely, by peeping inside the system, and the points of penetration where system can be broken by these perpetrators. There are always some weak points in a system, which are vulnerable to outside attacks/unauthorised entry in the system.

Security testing cannot be done by any verification method though there can be indirect proof of security by reviews, inspection, etc. One must follow validation activities to prove that system is protected enough against any external attacks and unwelcome guests. No system can be fully protected from all types of attacks. It is also not required. Some definitions associated with security are given below.

Vulnerability No system in the world is perfect. There are some weak parts and some strong parts of any system. The weaker parts of the system represent the vulnerabilities in the systems. These parts of the system are less protected and represent weaker parts or possible points of penetration. There is no system in existence which does not have any vulnerability. One must take precaution not to expose these weak points to outsiders.

Threats Threat represents the possible attacks on the system from outsiders with malicious intentions. Threats do exist in a system where the exposure to the world is more. Threat is defined as an exploitation of vulnerabilities of the system.

Perpetrators Perpetrators are the entities who are unwelcome guests in the system. They can create a problem in a system by doing something undesirable like loss of data and making changes in system. Perpetrators can be people, other systems, viruses, etc. They represent the possible threat to the system.

points of Penetration The points where the system can be penetrated or where the system is least guarded represents the point of penetration. These points represent the vulnerabilities in the system.

11.5.1 THE PROCESS OF SECURITY TESTING

The process of security testing may differ from product to product, organisation to organisation and customer to customer. It is driven by the concept of risk and security expectations by the users. It is also driven by cost-benefit analysis. The process of security testing is given below.

- Make a list of all possible perpetrators of the system. This may include the people using the system, internet cloud if any, and possible attacks like hacking, viruses, etc. Perpetrators will represent the threat to the system. Internal authorised users also represent a threat when they try to use the system for malicious purpose.
- Make a list of all penetration points for the system where the attack is likely to happen. Attack can be on different layers such as presentation layer, logic layer, and database layer. It can be at different locations such as server, terminal, pipeline, etc. It can be physical locations such as development area, maintenance area, enhancements area, user area, etc. These points represent the weak areas or vulnerabilities in the system.
- Make a penetration matrix with one dimension as perpetrators, and another dimension as a point of penetration. Each quadrant will give the possible failure point of a system.
- Define the probability of security breakage, and impact of such breakage for each failure point represented by each quadrant. Each quadrant has probability and impact associated with it. Product of probability and impact represents the risk associated for the application. Higher score or greater RPN/RIN represents more problematic situation. Such cases must be taken for protection first during development. They also indicate the probable areas where testing must be concentrated.
- Execute tests on the basis of high-risk prioritisation which is a product of probability and impact or RPN/RIN.

Table 11.2

Security matrix/Penetration matrix

		Perpetrators			
		PxI	PxI	PxI	PxI
Points of Penetration	PxI	PxI	PxI	PxI	PxI
	PxI	PxI	PxI	PxI	PxI
	PxI	PxI	PxI	PxI	PxI
	PxI	PxI	PxI	PxI	PxI

Table 11.2 shows a Security matrix/Penetration matrix

It can be possible that security concepts of two different systems may differ significantly depending upon the definition of risk by users and concept of residual risk or acceptable level of risk for the users. It is mainly driven by customer perception about security, risk type and type of the application as viewed by customer. Similar applications can have different risk rankings depending on the type of users, type of usage, etc.

11.5.2 SOME COMMON AREAS OF SECURITY TESTING

- Systems containing highly classified data such as employee master, customer master, and project master may be more prone to threats as perpetrators may like to obtain the important data of organisation. Data may be very sensitive for the organisation or it can attract some legal actions if lost.
- eBusiness, and eCommerce systems may need very high protection of users privacy and information involving financial transactions. Loss of money over the system due to security issue can lead to legal consequences.
- Communication system may have a loss of information during transit or privacy protection problems. This may cause huge loss to the user if perpetrators can take/alter the information while in transit.

Common security checking may involve devising test cases that subvert the programs security checks and try to break the system defenses. Some of these can be as follows:

- Users are not supposed to write or store their passwords anywhere. The passwords must not be shared with anyone. Obtaining a password using unofficial method to break a system is one method of security testing.
- People may be able to guess about a password, if it follows some pattern. Names of near and dear ones, date of birth, religious faith, etc. can be used to decode the password.
- Login and password copying and pasting must not be allowed by the system. Small utilities can be used to break the password, if copy and paste functionalities are allowed.
- Idle terminals should get locked, after being left idle for some time. There may be some timeframe defined for it as appropriate from organisations perspective. If the terminal does not get locked automatically, then an unauthorised person may start accessing the system as valid user has already opened the system.
- Check permissions of different user groups/users and their privileges for system usage. Admin permission may be limited to few people only.
- People may not be able to access database directly and make backhand changes in the database. If somebody can open the database directly, this can affect the application as the records can be directly added in database without going through validation and verification routines defined in system. One may need to check database security.
- Sometimes, there are limits defined for number of users for the system or for individual groups. Typically, number of users with admin privilege should be restricted. Try to create more users than allowed in user group.
- Deleting user groups like admin/supervisor should not be allowed by the system. When the application is installed, 'super user' login may be used to create the first user with admin privileges. One person with admin privilege should not be able to delete another user with admin privilege.
- Renaming supervisor/admin group with some other names, and creating new groups with these names should not be allowed by the system. Deleting groups containing admin users or any active users should not be allowed.

11.6 PERFORMANCE TESTING, VOLUME TESTING AND STRESS TESTING

Performance, volume and stress testing are three different kinds of testing, but often, there is confusion between them, or the terms are used interchangeably. The reason for this confusion can be attributed to the fact that each one of them has some effect on the other two, or all three are related to each other directly.

11.6.1 PERFORMANCE TESTING

Performance testing is intended to find whether the system meets its performance requirements under normal load or normal level of activities. Normal load must be defined by the requirement statement. Generally, system performance requirements are identified in requirement statement defined by the customer and system design implements them. Performance criteria must be expressed in numerical terms. Design verification can help in determining whether required measures have been taken to meet performance requirements or not. This is one area where verification does not work to that much extent, and one needs to test it by actually performing the operations on the system.

Performance criteria must be measurable in quantitative terms such as time in 'milliseconds'. Some examples of performance testing can be as given below.

- Adding a new record in database must take maximum five milliseconds. It means that when the record is added in database, it may take time lesser or equal to five milliseconds.
- Searching of a record in a database containing one million records must not take more than one second. One must add one million records in the system, and then use the search criteria to test this.
- Sending information of say one MB size across the system with a network of 512 KBPS must not take more than one minute. User may have to try various combinations in this testing.

Generally, automated tools are used where the performance requirements are very stringent and human senses may not be capable of capturing them exactly.

Illustration 11.2

If the normal usage of system is say 1000 users and the maximum expected time for logging in for a valid user is say 5 milliseconds, then one needs to put 999 users in the system first, doing some activities and new valid user logs in the system. Finding a system performance under such condition is performance testing.

11.6.2 VOLUME (LOAD) TESTING

Volume testing talks about the maximum volume or load a system can accept before it collapses due to load or volume. It can be described in terms of concurrent users, number of connections, maximum size of an attachment to be transferred over a network, etc. When the volume is above normal, it will definitely have poor performance. One must not check the performance under volume testing. Only factor to be assessed is whether the system is living or dead due to increased volume or load on the system.

Usage of Volume Testing in Design There are different options available in designing with respect to volume testing. It is governed by a 'factor of safety' expected by the user and the consequences of a load level that is more than the expected load.

- One way is not to allow more number of users than the stipulated number, so that system does not undergo heavy load at all. This can prevent more number of people from entering into the system, and thus, the system is protected from any excessive load. When a user tries to enter the system which is already loaded completely, the system can be made unavailable to the new user.
- Another way can be to allow users to join the system and give them warnings that they are extra joiners and may experience slow responses from the system due to load. This can have adverse effect on the

system performance but users will be able to work with the system. Sometimes, users may accept slow running system rather than a system which is not available.

- This approach can be used to create session timeouts if a user is not using a system for some stipulated time, so that new users can join the system without overloading it. This can be used along with security requirements to protect normal user.

Volume testing needs some kind of automation when the requirements are very stringent. One may have to increment the load by smallest possible factor and conduct many iterations till one finds a point where system actually collapses.

Example of Volume Testing Process Simple transactions are created and the number of users is incremented till the system fails. Transaction selected must represent very high probability of happening.

Illustration 11.3

The system may be designed to handle 1000 concurrent users. As soon as one more user tries to enter the system, it may give a message that system is already full and nobody can log in at that moment. Another way of handling is to allow the user to enter the system but system performance may get deteriorated. System may give a message that there can be a performance issue as system is overloaded.

11.6.3 STRESS TESTING

Stress testing talks about the system resources required by the transactions that have been planned to be undertaken. If the system has limited resources available, the response of the system may deteriorate due to non-availability or loading of resources. Stress testing is also governed by the 'factor of safety' expected by the user to protect system failures due to resource constraints. Stress testing is used to define the resource level required by the system for its efficient and optimum performance.

Example of Stress Testing Process Simple transactions are created, and the system resources such as processor size, RAM size, and bandwidth are reduced to find the minimum level of resources where system is barely living. Transactions selected must represent very high probability of happening.

11.7 RECOVERY TESTING

Recovery testing is intended to find how the system as a whole, or an individual machine/server as a component of the entire system recovers from a disaster. There are various methods of disaster recovery and system requirements/designs must specify the methods expected to be used during recovery. Some level of verification as a proof of recovery is possible but actual testing scenario is most important. This is also termed 'disaster recovery'. Disaster recovery is a subset of business continuity planning.

11.7.1 SYSTEM RECOVERY

System is made of several components such as machines, terminals, routers, and servers. System failure may mean failure of any of these components, or failure in terms of communication or physical disturbances in

the system components or failure of all or many components together. One must test the samples using high probability scenarios.

Different strategies can be designed and implemented depending upon the type of users and their expectation of uptime from the system. There are three famous ways of disaster recovery of a system.

- System Returns to the Point of Integrity After Meeting Disaster** When a system meets with any disaster, the transactions in the process are lost completely. When the system recovers from the disaster, it comes to the point of integrity last known to the system and user may have to start from that point onwards. The definition of point of integrity differs from system to system as per user expectations defined in requirement specifications and interpreted through system design. For some systems, if connectivity of the system is disturbed momentarily, 'refresh' button can bring it to the previous page where the system was last before it met with a disaster. While for a secured system, it may come to the initial point from where the user starts interacting with the system. This is very low level of disaster recovery as all the information in between is lost, since the system has met with a disaster.

- Storing Data in Temporary Location** When the system meets with a disaster, the transactions made till that point are stored temporarily by the system. As soon as system returns to its original state, user is shown the last transactions and asked to confirm whether transactions must be committed or not. If user accepts the transaction, it may be recorded in the system and user can progress from that point onwards. If user rejects it, transaction is lost permanently and user may continue to use the system. For storing the data in temporary location, the system may need a space for temporary storage. When number of users is very large, storing temporary data may need big volumes of storage space available as well as facility to get back to user about temporary data when users logs in after disaster. This is a very advanced way of handling disaster as data is stored in temporary files till the user accesses it again and confirms the fate of it.

- Completing the Transaction** When the system meets with a disaster or stops working, the transactions upto that point are automatically committed. The system will try to complete the committed transaction till the point upto which it is possible to proceed and stops at that point. It may need internal design to complete the transaction to the point possible as all components do not fail at a time. This method of disaster handling is midway between the two extreme ends defined above.

11.7.2 MACHINE RECOVERY

Machines such as application server and database server may contain very vital information and data. It is very important that data should not be lost when the system meets with disaster. When there is a problem with such machines, the data available may be lost or may not be available for some time and user may suffer. It can be a huge loss for the user if the data remains unusable for long time or is lost permanently. Machine recovery supplements the system recovery to some extent. It is driven by the value of data to the user and kind of system one is working with. Cost-benefit analysis is essential for deciding on the type of backup mechanism one wishes to use. Machine recovery is of four types, as given below.

- Cold Recovery**, Cold recovery talks about backing up of a data at a defined frequency such as once in a week. The data is backed up on an external device like tape/CD. Backup media may be kept at a location as defined by backup plan. When there is a problem with the machine, data restored on these external devices

is recovered on some other machine, and that new machine is introduced in a system to replace the original machine. It may take a considerable time to make second machine available, putting all the prerequisites and data on it. Sometimes data may be lost as external device backup may not be successful or may not be recoverable. In any case, data updated after last backup is lost permanently.

- **Warm Recovery** Warm recovery happens when a backup is taken from one machine to another machine directly. Frequency of backup may be more frequent and automatic backup is possible as it is machine-to-machine backup. Data is already on the hard drive of both machines, if the backup has been successful. Backup machine used may not be of same configuration as that of the original machine. When some problem happens with the original machine, backup machine may be introduced in the system temporarily. Backup machine sometimes need upgradation of configuration to replace the original machine, or performance of backup machine may not match with the original machine. Sometimes, user may face service problem, if the backup machine is inferior compared to actual machine. Here, cost involved in maintaining two machines is much more than 'cold backup'. Data may be lost in case the data is updated after last backup.

- **Hot Recovery** Hot recovery represents a scenario where two machines, original as well as backup machine, are present in the system. One machine is a primary machine while the second machine is treated as a backup machine or standby machine. Both machines are of same or similar configurations and capabilities. Backup frequency is defined as per backup plan. When a problem occurs with one machine, controls are shifted to second machine containing backup data so that it can be used. Cost involved in maintaining two machines may be very high. Data may be lost in case the data is updated after last backup.

- **Mirroring** As we know, hot recovery represents a system with equal configuration machine. Mirroring also involves two machines with same configuration and backup frequency is mirroring data, i.e., every millisecond data is copied from one machine to another. The data is backed up online. If something happens to primary machine, then backup machine takes over the primary machine immediately without any manual intervention. This arrangement may need a huge cost but data availability is almost continuous.

11.8 INSTALLATION TESTING

Most of the applications need installation before they can be used. Installation testing is intended to find how the application can be installed by using the installation guide or documentation given for installation along with installation media like CD. It attempts to identify ways in which installation procedures lead to correct results. Installation may be done by using any external device such as CDs or pen drive, or it may be from network or remote installation, as the case may be.

11.8.1 WHAT DOES INSTALLATION TESTING DETERMINE?

- Installation procedures are documented correctly or not in the installation manuals supplied with product. Users must be able to use the instructions given in the installation manual for installation and installation must be successful.
- If installation needs any training, personnel involved in installation must be trained for the same. Users after requisite training must be able to install application independently.
- Installation may be auto-run or semi-automatic or manual as the case may be (as defined by requirements). In case of auto-run, entire installation must be done automatically without any manual intervention. In case of semi-automatic installation, it should halt as and when user inputs are required. In case of manual installation, people should be provided adequate help while installing software.

- If installation involves migrating from one system to another, or upgrading from old system to new system, the process of upgradation must be documented accordingly. Installation must be able to detect existing system and must help user for upgrading it accordingly.

11.8.2 INSTALLATION PROCESS

Installation may be done through devices like CD, pen drive, and floppies etc. It can be done by using network, or from one machine to several machines at a time. Sometimes, remote installation is also required. Installation may have some prerequisites which are essential for installation or working of software. During installation, it must identify the presence or absence of these prerequisites, and if they are not available, it must inform the user about it. Installation may be possible on any partition if requirements are defined accordingly. Installation must not replace any of the existing files available on the disk.

Installation process can be fully automatic or it may need user actions as the case may be. If it needs user actions, the process must guide user about what actions are available, along with which action is most recommended by the system.

11.8.3 UN-INSTALLATION TESTING

Un-installation testing is used where user requirements define the same in requirement document. If it is expected to be available, un-installation must clean all the components and files installed during installation. It must not leave any thread that the installation was done on that system or machine.

Un-installation testing can be a requirement of product where the product removed from the system must clean the system completely. The process of un-installation testing may be as follows.

One may have to take an image of a hard disk before installing software. This can be used to note all the files existing at the time of installation. Then, the tester must install the application and again capture the image of hard disk. One must compare two images to find if any pre-existing file has been overwritten during installation. Then, one may uninstall the application and take an image of hard disk again. This image must be compared with the first image before installation of software. These two images must match exactly if clean un-installation is proved.

11.8.4 UPGRADATION TESTING

Sometimes, an application may need an upgradation, in order to upgrade it from older version to newer version. Upgradation may be done by using patches released by the product manufacturer from time to time. It may be done using CD, floppies or any other media used for upgradation. Process of upgradation may be as follows.

During upgradation, the installer must be able to identify that there is an older version of same application available on the disk. No upgradation is possible when there is no application existing on the disk. Also, if existing application is more updated than the upgradation available, no upgradation is possible.

Upgradation also follows similar process as that of installation. It may be automatic, semi-automatic or manual as the case may be. User must be helped adequately for upgradation.

11.9 REQUIREMENT TESTING (SPECIFICATION TESTING)

Requirement testing is also termed 'specification testing'. Every system must be requirement tested to ensure that the system being made will suffice the users needs. Process of requirement testing begins from requirement phase and continues till operations and maintenance phase, where at every stage of development,

requirements are tracked and traced to meet user expectations. Objective of requirement testing includes the following.

- User requirements are implemented correctly or not
- Correctness is maintained as required by customer
- Processing complies with organisations and customers policies and procedures

Methods of requirement testing include the following.

- Creation of requirement traceability matrix to determine whether all requirements are implemented or not during software development. Traceability includes requirements, designs, coding, test cases and finally, test results. Any test case failure can indicate which requirements have not been met.
- Use of checklist to verify whether system meets organisational policies and regulations and legal requirements. Checklist approach is used to verify the processes used for gathering requirements including formats, and templates prescribed.
- Create the use cases/prototypes or models from requirement statement to understand the characteristics of requirement.

11.9.1 REQUIREMENT TESTING PROCESS

One must have a requirement statement for testing requirements. The tester is expected to write the business case using requirement statement. He/she must identify the actors in the business case, and transactions done by different actors during execution of these use cases. At any time while writing a business case, if the tester has to assume something, it gives lacunae in requirements in terms of completeness. At any place where decisions are involved, all possible outcomes of the decisions must be covered in requirements. Any outcome not covered gives incompleteness to requirements.

Test cases are written by referring to the requirement statement. These test cases are executed in system testing. If any defect is found, it is noted and fixed during defect fixing phases. System may need retesting/regression testing to verify that all defects are fixed and no new defect has been introduced when old defects are fixed.

11.10 REGRESSION TESTING

Regression testing is intended to determine whether the changed components have introduced any error in unchanged components of the system. Regression testing may not be considered as special testing in development projects. But often, maintenance projects may consider it as special testing as regression testing of entire application may be very costly. Regression testing can be done at,

- Unit level to identify that changes in the units have not affected its intended purpose, and other parts of the unit are working properly even after the changes are made in some parts.
- Module level to identify that the module behaves in a correct way after the individual units are changed. Whatever was a correct behavior before making a change must be retained, but where it was failing and change was necessary must also behave correctly now (second part is called retesting).
- System level to identify that the system is performing all the correct actions that it was doing previously as well as actions intended by requirements after change is made in some parts of the system.

Regression testing is performed where there is high risk that changes in one part of software may affect unchanged components or system adversely. It is done by rerunning previously conducted successful tests

to ensure that unchanged components function correctly after the change is incorporated. It also involves reviewing previously prepared documents to ensure that they remain correct after changes have been made in the system.

11.10.1 IMPORTANT DEVELOPMENT METHODOLOGIES WHERE REGRESSION TESTING IS VERY IMPORTANT

- Any kind of maintenance activity conducted in a system may need a regression testing cycle. Maintenance may include defect fixing, enhancements, reengineering or porting, as the case may be.
- Iterative development methodology needs huge regression testing as there are several iterations of requirement changes followed by design changes and changes in code. Iterative development makes a system fragile.
- Agile development needs huge cycles of regression testing.

There are many tools available for automating regression testing. As efforts and time required for doing regression testing is huge, automation may be a faster and cheaper method.

11.11 ERROR HANDLING TESTING

When normal users are working with the system, it may be possible that they may enter wrong data or select wrong options. Application is expected to help user through error messages, if anything unexpected happens with a system. Error handling has a direct relationship with usability of an application, and a distant relationship with security of system. System may give various error messages when something wrong is being tried by the user, or system does something wrong while processing data entered by the user. Error messages indicate to users that something is wrong or give a method to resolve the issue or prevent the error from happening. Error handing testing is done to determine the,

- Ability of system to properly process erroneous transactions and protect the users from making any mistake during data entry. It may be possible that user may be prevented from entering erroneous transactions or an error message may be given, when such entry or processing is detected by the system.
- All reasonably expected errors by application system are recognised as and when they occur, and the appropriate error messages are given to the users when an error happens. It must help user in identification and correction of errors as defined by requirement statement.
- Procedures must provide that high-probability errors will be detected and corrected properly before system processes such data. There may be several types of error messages such as the following.
- Preventive Messages** When user tries to enter some wrong data, system identifies a wrong entry and prevents such entry in system.

Illustration 11.4

When user tries to enter a date '13/05/2009' in the system, which accepts date in format 'MM/DD/YYYY', the system automatically flashes a message that user is expected to enter date in 'MM/DD/YYYY' format and 13 is not a valid month.

- **Auto-Corrective Message** An example of auto corrective message is when system tells user about what is wrong and corrects it automatically. It is mainly used to prevent the user from reentering the transaction, where there is a single way of entering the transaction and second time, it may not be accessible to user for correction. This avoids user frustration as the data entered by user is corrected automatically. But this can be problematic, if user entered a data, where correct version of data is not known to system.
- **Suggestive Message** In case of suggestive messaging, system tells the user about what is wrong and provides suggestions about correcting it. User can overrule the suggestion and may reenter transaction again or may accept the suggestion, as the case may be. This avoids automatic entry but user must be in a position to know correct entry of data either by reentering or accepting suggestion made by system.
- **Detective Message** Detective messaging happens when system tells the user about what is wrong but there is no guidance available about what can be the correct transaction. These messages neither provide any guidance nor make any suggestion to user. The control is given to user for correcting entry without much information about what is wrong and how it should be corrected.
- Reasonable control over errors during correction either by auto correcting or suggesting, or protection of a system from erroneous transactions and malicious users with just detecting or sometimes without showing detection to users. Error correction must not introduce any error in the system.
- Error handling must happen throughout development life cycle as well as during acceptance testing and use. Good error handling helps user while working with system and improves usability.

Types of errors and the way they must be handled must be defined in requirement statement. Cost of error handling and time estimation for building proper controls must be considered while developing such system. One must understand security of malicious use as error handling may help intruder to use the system maliciously. Good error handling reduces security of system to some extent.

11.12 MANUAL SUPPORT TESTING

Most of the systems need manual intervention or an input by user for its working at some time or another. Manual support testing is intended to test the interfaces between people as users of an application and application system. Manual testing is used to determine whether,

- Manual support procedures are sufficiently documented, complete and available to user. It can be in the form of online help or user manual or trouble shooting manual.
- Manual support people are adequately trained to handle various conditions of working including entering data processing, taking outputs as well as handling errors. People must be able to work with system independently.
- Manual support and automated segments are properly interfaced within the application. Help available must provide guidance to common user when some problem is encountered by them.
- User must be able to work with system without assistance of system personnel. He/she must be able to evoke help and complete self servicing.
- Provide inputs to support group, and enable manual support group to enter into the system at proper time or when users need their help.
- Prepare output reports, and ask people to take necessary actions based on these reports.

11.13 INTERSYSTEM TESTING

No system in real world works alone. There are possibilities that the system developed may have to interact with many other supporting systems or systems existing before the new system is installed. Some systems

may be automated while some may be manual systems. Testing of interfaces between two or more systems is essential to make sure that they work correctly and information is transferred between different systems. System testing is designed to determine whether,

- Parameters and data are correctly passed between application and other systems to avoid any communication failure. Acceptance of data and output of data must match with the requirements.
- Documentation for the involved system must be accurate, complete, and matching expected inputs and outputs. It must define parameter passing and bridge of communication between various systems.
- System testing must be conducted whenever there is a change in the parameters between applications communicating with each other. There may be changes in application or there may be changes on other external systems.
- Representative set of test transactions is prepared in one system and passed on to another system for processing and the results are verified for correctness.
- Manual verification of documentation is done to understand the relationship between different systems.

11.14 CONTROL TESTING

Control testing is done to check data validity, file integrity, audit trail, backup and recovery, and documentation for the system under development. Control testing is done to determine the following.

- Data processed is accurate, complete and can be used by the normal users. Users must complete all mandatory fields before saving the record. If mandatory fields are not completed, it must stop user by messaging and shifting control to those fields which are mandatory.
- Transactions entered in the system are authorised by identifying the user permissions. Unauthorised persons may not be able to access the records, or may not be able to save it or modify it or delete it.
- Audit trail is maintained to know what transactions are done by the user while working with the system. Transactions must be traceable from start to end in entire data processing life cycle.
- Process must meet user needs. It must be supported by requirement statement.
- Ensure integrity of processing of transactions and data from entry till exit. Data must not be lost, modified or added during the transaction processing.
- Identify risks associated with different users using the system, and their ability to interact with the system. Access right definitions must be followed.
- Create risk conditions in the test laboratory to assess the level of control mechanism. Subject system to these conditions for testing to validate the control provisions in system designs.
- Evaluate effectiveness of controls defined by the requirements. Controls must be effective, efficient and must justify their presence.
- Run program to accumulate details and then compare with total.

11.15 SMOKE TESTING

Smoke testing involves testing basic functionality of software application developed to ensure that application is living and one can work with it. Generally, these tests are performed without any user input. Steps involved in smoke testing can be installation, navigating through the application, invoking or accessing some major functionalities. Smoke testing is not for approving or certifying the application. It only tells the tester whether the application is alive or not. Smoke testing is not applied for testing the application but to ensure that normal

user will be able to work with it. If smoke testing fails, user will not be able to work with the application. Such failure may result into rejection of an application without going further.

Success of smoke testing is one of the entry criteria for system testing. If smoke testing fails, system cannot be taken for further testing. Generally, smoke testing is done by test manager or senior tester, as the case may be. Smoke testing is also termed 'Smell test' as test manager may have to make a judgment about the system in very less time.

11.16 SANITY TESTING

Sanity testing is performed to test the major behavior or functionality of the application. Depth of sanity testing is more than smoke testing. It normally includes a set of core tests of basic GUI and functionality to demonstrate connectivity to database, application servers, printers, etc. Some people consider smoke testing and sanity testing as the same while some differentiate between them. Sanity testing is also known as 'Build Verification Testing (BVT)' where the testers test if consistent results can be obtained when some test cases are executed. Sanity testing helps in identifying whether the test case results are consistent or not, and also application behaviors are reproducible or not. If sanity testing fails, application may be rejected without doing any further testing. Success of sanity testing is another entry criterion before system testing. Generally, sanity testing is done by test manager or senior tester, as the case may be.

11.17 ADHOC TESTING (MONKEY TESTING, EXPLORATORY TESTING, RANDOM TESTING)

Adhoc testing is performed without any formal test plan, test scenario, test cases, or test data. It is also called 'exploratory testing' or 'monkey testing' or 'random testing'. Testers try to test the system with different combinations of functionalities on the basis of error guessing and experience about similar application in past. This helps in identifying some hidden defects that might have been missed in all previous test efforts.

Disadvantages of adhoc testing

- This testing may not be reproducible as tester may not remember all the steps done till the point where the defect has been seen. Sometimes, defect is not reproducible only because tester does not remember the steps.
- The scenario tested may not be definable, and may not represent real-life business scenario. Some adhoc scenario with an intention to break the system may not have much value in real life. Sometimes, probability of happening of such events may be next to '0'.
- Adhoc testing needs testers with very good domain expertise and good command over testing process.

Advantages of adhoc testing

- The scenarios tested are adhoc, and there may not be a particular sequence. System may be under stress while executing such scenarios.
- It may try some scenarios which may not be considered at all while writing requirement statement.
- It may need less time as there is no test plan, test scenario, and test cases to be written. It is also termed 'playing with an application'.

11.18 PARALLEL TESTING

Parallel testing is done by comparing the existing system with newly designed system to validate it against existing system. It is used to compare results from two different systems in parallel to find out the similarities

and differences between them. Parallel testing is done extensively in acceptance phases, typically in beta testing or business pilot where existing system, legacy system or manual operations are compared with the new system being developed.

Parallel testing is done to determine whether,

- New version of application or new system performs correctly with reference to existing system that is supposed to be working correct. It is extensively used in beta testing where new system behavior is compared with existing system or manual operations, as the case may be.
- There is consistency/inconsistency between two systems. Consistency may be mainly in terms of user interactions, user capabilities, etc. Consistency with respect to processing of transactions and controls designed for validation is also evaluated.
- Parallel testing is used extensively in business piloting or beta testing while accepting a new system. This is also called 'comparison testing' where old system behavior is considered as correct. Parallel testing is used extensively in compatibility testing.
- Same input data must be used in both systems and outputs from two systems may be compared with each other. Input data entry process may be modified as per requirements of new system. It may be possible that existing manual system may have a manual data entry transaction while new system may need electronic data entry. Same thing is applicable for data output processes where output formats may change.
- New system is used in parallel with the existing system for certain time period, to find the differences between two systems. Thorough cross-checking of the outputs, and comparison with outputs from existing system is required.
- Security, productivity, and effectiveness of new system must be comparable with the old system. If there is any lacuna in new system with respect to old system, new system may get rejected.

11.19 EXECUTION TESTING

Execution testing is performed to ensure that system achieves desired level of proficiency in production environment when normal users are using it in normal circumstances. Execution testing may be considered as alpha testing if it is done in development environment, or beta testing if it is done in user environment. Data used in execution testing must be shared by customer. Execution testing involves actual working on the system in production environment to determine whether,

- System meets its design objectives as defined in requirements and expected by users. Execution testing is used to evaluate users experience with new system.
- System must be used at that point of time when results can be used to modify system structure, if required. Alpha and Beta testing anomalies may be used to modify system, if required.
- Execution testing may be conducted by using hardware/software monitors, by simulating the functioning of the system, and by creating programs to evaluate performance of completed system.

11.20 OPERATIONS TESTING

Operations testing is performed to check that operating procedures are correct as documented in user manuals, and staff can properly execute the application by using the documentation available with it. Operations testing is done to determine whether,

- The system documentation is complete of operator documentation, user manual, etc. People must be able to work with the system by referring to these documentations.

- The user training, if required as per requirement, is complete and user can use the system on the basis of training provided. Effectiveness of training may be evaluated in this testing.
- Operations testing must occur prior to placing the application into production environment. Actual users must be capable of working with system, and system must be able to work with normal users.
- Operations testing must be conducted without any assistance provided to operators, as if it was part of normal computer operations.

11.21 COMPLIANCE TESTING

There are many standards available and used by different user groups depending upon user application domain in which application will be working. These may be mandatory/recommended by different customers, governing bodies, etc. for different software systems as per the domain in which these systems will be used. There may be few regulatory or statutory requirements enforced by different agencies in the environment where application is expected to work. Examples of standards applicable for software may include medical standards such as 'FDA (Food and Drug Administration) regulation' for software in medical domain in United States, standards for war equipments for software working in military operations and equipments, and special standards for software working in aviation industry. Compliance testing is intended to check the application/development processes with the standards applicable to such application.

Compliance testing is done to check whether system is developed in accordance with the prescribed standards, procedures and guidelines applicable to them as per domain, technology, customer, etc. It helps to determine whether,

- Development and maintenance methodologies are followed correctly or not while developing/maintaining system belonging to different domains. Domain related protocols must be followed.
- Completeness of system documentation with respect to applicable standards is ensured. Documentation must be clear and complete and must be complying with standards applicable.
- Compliance depends upon managements desire to have standards enforced as well as geographical and political environment where application will be working. Customer requirements must include definition of regulatory and statutory requirements.

Generally, compliance testing is done by verification method where artifacts and screens are reviewed with respect to standards applicable. Compliance testing is done using the following.

- Checklist prepared for evaluation or assessment of product
- Peer reviews to verify that the standards are met
- SQA reviews by quality professionals
- Internal audits

11.22 USABILITY TESTING

Usability testing is done to check 'ease of use' of an application to a common user who will use the application in production environment. It involves using user guides and help manuals (including online help) available with application by normal user to find its usefulness. It is applied to determine whether,

- It is simple to understand application usage through look, feel and support available like online help. It includes testing whether help is available and user can use it effectively.

- It is easy to execute an application process from user interface provided. If training is required for using the application, it must be provided to users who will be using it.
- Usability testing is done by,

- Direct observation of people using the system, noting their interactions with system and ease with which these users can work with the system under testing. In case of any problem during working, evoking help or referring to user manuals may be checked if these are sufficient to help users in solving their problems.
- Conducting usability surveys by checking the deployment or implementation of system in production environment. Normal users must be able to use system on their own without any assistance.
- Beta testing or business pilot of application is user environment.

Usability testing checks for human factor problems such as,

- Whether outputs from the system such as printouts and reports are meaningful or not. They must be evaluated from users perspective of 'fit for use'.
- Is error diagnostic straightforward, or are common people not able to understand the error messaging. Error messaging must help common users using the system.
- Does user interface have conformity to the syntax, format, and style observations as demanded by users or generally agreed standards. These may be required as per customer standards or standards imposed by statutory/regulatory bodies.
- Is the application easy to use to the common users
- Is there an exit option available in all choices so that user can exit the system at any moment.
- System must not annoy intended user in function availability or speed or user interfaces or any other feature. Use of system icons and pictures must be as per generally agreed methodology and practices of user community.
- System taking control from user without indicating when it will be returned can be a problem as user may not be aware of how much time it would take. It must indicate which operation is going on and how much time it will take.
- System must provide online help or user manual to get self service. In case of any problem, people must be able to evoke help.
- Consistent in its function and overall design.

11.23 DECISION TABLE TESTING (AXIOM TESTING)

Decision table is a good way to capture system requirements that contain logical conditions, and to document internal system design handling various conditions faced by the system. They may be used to record complex business rules that a system is expected to implement. Specifications are analysed, and conditions and actions of the system are identified. The input conditions and actions are most often stated in such a way that they can either be true or false (Boolean).

Decision table contains the triggering conditions, often combinations of true and false for all input conditions, and the resulting actions for each combination of conditions. Each column of the table corresponds to certain business rule that defines a unique combination of conditions, which result in the execution of the actions associated with that rule. The coverage standard commonly used with decision table testing is to have

at least one test per column, which typically involves covering all combinations of triggering conditions. One may use equivalence partitioning and boundary value analysis for testing such system.

Table 11.3

Decision table

		Purchased volume and discounts		
No. of Books	1-50	51-500	501-5000	5001 and more
Discount offered	0%	2%	3%	5%

Table 11.3 shows a sample decision table

The strength of decision table testing is that it creates combinations of conditions that might not otherwise have been exercised during testing. It may be applied to all situations when the actions of the software depending on several logical decisions are occurring at same time. Table 11.3 indicates single dimension of variable. In practical situations, there can be multiple dimensions possible.

Axiom testing is a part of decision table where system works on some relationships between variables. When somebody is testing a combination of two variables X and Y which are related to each other with a relationship expressed as $Y = f(X)$, it may be termed 'axiom testing.'

11.24 DOCUMENTATION TESTING

Software development life cycle generates many artifacts which are not part of final system or executable, but are very important to understand the system in future. These may be termed 'system documentation' in general.

System documentation may contain requirement specifications, requirement changes, impact analysis, design artifacts, design changes, impact analysis, code documentation, project plans, and test plans. All these documents are required to build the right system.

When the system is delivered to the customer, it is not only the executable or sources which are delivered but it should contain all these support documentation required for future maintenance of a system.

Documentation testing involves review of all the documentation accompanying sources/executable to the customer so that system can be maintained in future. All the artifacts must be in sync with each other and must represent a system which is being delivered. Sometimes, documentation also includes release notes, known issues and limitations if any, installation guide, user guide, and trouble shooting guide.

The purpose of documentation testing is to thoroughly go through all written material that will be presented to the user as a part of implementation. The testing needs to be done in a few different ways.

- Ask users to follow all documented procedures. These should be written to provide step-by-step guidance on how to accomplish a given task. If the users cannot successfully complete the procedures, the documentation needs to be improved.
- Have people with strong language skills to review all the documentation for professionalism and readability.
- Try out all alternative ways that are documented to accomplish a task. In many cases, the primary way works, but the alternatives do not work as expected.
- If one is describing policies or standards, then make sure that the appropriate authorities in the organisation review and approve them. It would be disastrous to misquote or misapply an important company policy.

- Evaluate any manual forms, checklists, and templates to ensure that they are accurate and the appropriate information is being collected.

11.25 TRAINING TESTING

Sometimes, the vendor is expected to give training to users who will be using the system in future. A better approach is to test training as a part of system testing. This implies that the training must be ready at this point in the life cycle and not created at the very end of the project. One must perform testing in a controlled test environment to ensure that it is effective, accurate and bug-free.

In the case of distance learning, one must make sure that the technology used is correct for the purpose. One must make sure that there are adequate equipments for imparting such training. Training material may be a deliverable of the project. It needs to be tested for accuracy and defects, before it is rolled out for the first time.

11.26 RAPID TESTING

Rapid testing is a powerful technique that can be used to complement conventional structured testing. It is based on exploratory testing techniques, and is used when there is too little time available to obtain full test coverage using conventional methodologies. Rapid testing finds the biggest bugs in the shortest time, and provides the highest value for money.

In an ideal world, rapid testing would not be necessary, but in most development projects, there are a number of critical times when it is necessary to make an instantaneous assessment of the products quality at that particular moment.

Some areas where rapid testing is applied extensively are given below.

- 'Proof of concept' test early in the development cycle.
- Prior to, or following migration from development to the production environment.
- Sign-off of development milestones to trigger funding or investment.
- Prior to public release or delivery to the customer.

Although most projects undergo continuous testing, it does not usually produce the information required to deal with the situations listed above. In most cases, testing is not scheduled to be complete until just prior to launch, and conventional testing techniques often cannot be applied to software that is incomplete or subject to constant change.

Structured testing is a vital part of any development project but it cannot meet all the quality assurance objectives. Its primary objective is usually affirmative testing, i.e., to verify that the software does all the things it is supposed to. However, software is now so complex that there are often a seemingly infinite number of permutations of the data variables, and variations in the time domain can add another layer of complexity.

In addition to affirmative testing, it is necessary to identify undesirable behavior, so that it can be corrected. But structured testing is far less useful as a technique for doing this. The number of permutations means the time required for analysis, scripting and execution. The time required may not be available and the time requirement could only be reduced if the tester knew in advance what the faults were likely to be.

How Does Rapid Testing Work? Rapid testing is based on exploratory testing techniques, which means that the tester has a general test plan in mind but is not constrained by it. The plan can be adapted

on-the-fly in response to the results obtained for previous tests. The downside is that it is not possible to guarantee total test coverage, but the benefit is that a skilled tester can quickly find faults that would have eluded a scripted test. We often refer to exploratory testing as 'expert testing', as it requires a high level of technical knowledge and experience to be effective.

Rapid testing extends the exploratory concept by making judgment about what faults to report and the level of details to be recorded. Once a fault has been identified, the time taken to investigate and document it reduces the time available to find other faults, so the tester may fail to find the serious faults if they spend too much time reporting less important issues. We refer to this as the 'quality threshold' and it is fundamental to the effectiveness of rapid testing. Many testers find it an alien concept but it is a necessary one.

The tester will consult with the customer to determine the initial quality threshold, but it is often necessary to vary it during the course of testing depending on the product quality. If the overall product quality is good, it may be possible to reduce the threshold and investigate the less serious faults.

11.27 CONTROL FLOW GRAPH

Control flow graph is a flow of a control when a program is getting executed. If a program does not have any kind of decision, flow happens in a single direction. Whenever there is any decision to be taken by a program, there is a possibility of different flow graphs possible. Each decision induces multiple paths in a program while it is getting executed.

Dominators When we begin from the start of a program, and there exists a set of code which will always be executed when a path is selected, then it is termed 'dominator'. These are to common paths independent of any decision or branch.

Post-Dominator From any point in an application if we go to the end of the application, then if we have to go through particular set of code, it is defined as 'post-dominator'. There may be several possible paths depending upon the number of decisions.

11.27.1 PROGRAM DEPENDENCE GRAPH

When the program is getting executed, execution may be dependent on two factors, viz. data dependence and control dependence.

- Data dependence flow depends upon the data input to the system. Consider loops like 'if' and 'while' where the loops get executed on the basis of data. Data input decides how system will work.
- Control dependence flow is defined as the flow of instructions due to control points in a program. If the program has redundant code, control will never go to it. Control may be defined by user requirements.

11.27.2 CATEGORY PARTITION METHOD

Category partition method is used to generate the test cases from requirements. The following steps are involved in generating test cases by category partitioning method.

Analyse the Requirements The requirements are put into different sets like functions, user interface, and performance requirements. These may be tested independently and they are put in different test suites.

Identify Categories The kind of input which will have specified expected results of categories of output with some specified inputs are analysed. Decision tables can be used effectively to identify those categories.

Partition the Categories Inputs with similar output, or output with similar input are put into different partitions or classes.

Identification of Constraint There can be some categories of inputs and outputs which cannot exist together or which are impossible. They must be identified. These may be defined as exclusion for testing.

Creating Test Cases Accordingly Test cases are created noting the possible constraints defined in user requirements.

Processing the Test Cases The test cases defined above are executed and results are captured.

Evaluate the Output The output is verified with respect to expected output.

Generate the Test Sets If the output obtained and the expected output are matching, then it may be added in test set.

11.27.3 TEST GENERATION FROM PREDICATE

A condition which a code can achieve is called 'predicate'. This statement indicates a condition and action part when that part of code gets executed. Condition part is represented by predicate while action part represents expected result from such action. Testing used to find that there are no problems in predicate is called 'predicate testing'. Predicate testing covers any opportunity of detecting a fault created by developer while coding a program.

11.27.4 FAULT MODEL FOR PREDICATE TESTING

Predicate testing may target for three different classes of faults as defects, viz. incorrect boolean operator used, incorrect relational operator used, and incorrect arithmetic operator used.

11.27.5 DIFFERENCE BETWEEN CONTROL FLOW AND DATA FLOW

Very often, there is a possibility of confusion between control flow and data flow. Many times, data selected may decide the flow of event and it may be taken as control flow while sometimes, control flow is seen to be affecting data flow. Table 11.4 specifies the difference between the two.

Table 11.4

Difference between control flow and data flow

Control flow	Data flow
Control flow is process oriented. It defines the direction of control flow as per decision of system.	Data flow is information oriented.
It does not manage data or pass data from one component to another.	Data flow passes data from one component to another.
It functions as a task coordinator. Control flow requires task completion.	All transformation of data are at a work either simultaneously or one after another.
It is synchronous in nature. Even if the tasks are not connected with each other, they can be synchronous.	It may or may not be synchronous in nature.
Tasks can be executed in parallel or one after another.	Generally, tasks are executed one after another, if there is a serial dependency. Otherwise, they are independent of each other.

11.28 GENERATING TESTS ON THE BASIS OF COMBINATORIAL DESIGNS

An application is expected to work under various environmental configurations such as Hardware, browsers and operating systems etc. This is called 'test configuration'. In reality, there may be huge number of configurations possible, and one may not be able to test all of them in a test lab due to various constraints like time, money etc. Environment under which the application is expected to work may give one or more factors. Each factor may possibly be tested by using one test case, which makes total number of test cases very large tending to infinity.

In combinatorial designs, we try to reach some finite level of test cases which can give adequate confidence that program will work in all possible combinations. The set of input and output are partitioned so that the value selected may represent each partition.

11.28.1 COMBINATORIAL TEST DESIGN PROCESS

Combinatorial test design process may be as follows.

Modeling the Input Space and Test Environment The model consists of set of factors and corresponding levels. Factors are decided on the basis of interaction between application and environment in which it is expected to work.

Generate Combinatorial Object This is an array of factors and levels selected for testing. Such an array will have each row covering atleast one test configuration from the list.

Generate Tests and Test Configurations From the combinatorial object, a tester may have to generate the test cases and test configurations. Figure 11.1 shows schematically how the test cases can be generated.

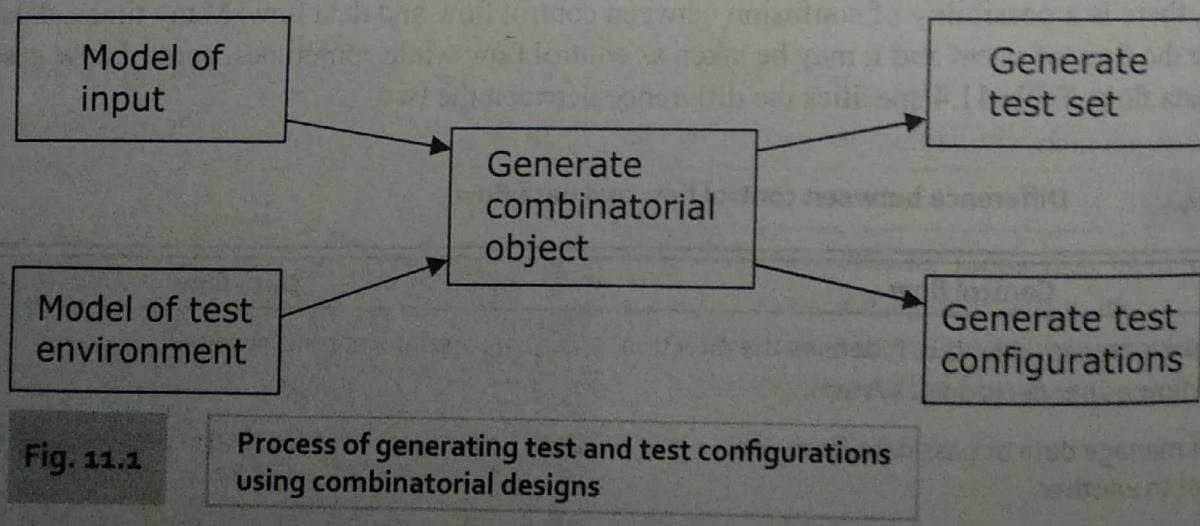


Fig. 11.1

Process of generating test and test configurations using combinatorial designs

11.28.2 GENERATING FAULT MODEL FROM COMBINATORIAL DESIGNS

The aim of combinatorial designs is to generate test cases where there is maximum probability of faults or defects. Simple fault is a fault triggered by the input variable irrespective of the other input variables. Pairwise interaction faults are triggered when there are two variable combinations which can yield a fault. Three-way interaction fault is when three variables in combination create a fault but individually there is no fault.

11.29 STATE GRAPH

State graph and state table are useful models for describing software behavior under various input conditions. State testing approach is based upon the finite state machine model for the structures and specifications of an application under testing. If we draw a state graph, states are represented by nodes. Whenever an input is provided, the system changes its state. This is also called 'state transition'.

When there are multiple state graphs, it becomes difficult to draw them. It is further difficult to understand them. To remove this complexity, state tables are used. State tables may be defined as follows.

- Each row indicates a state of an application.
- Each column indicates the input going to an application.
- Intersection of rows and columns indicate the next state of application.

11.29.1 GOOD AND BAD STATE GRAPHS

Generally, it may be considered that all the graphs usable in software may be good state graphs. There are few instances when we come across bad state graphs also. Some characteristics of good state graphs are given below.

- Total number of states possible is equal to product of all possibilities of factors that make a state.
- For every state and output, there is exactly one transition specified to exactly one state.
- For every transition, there is one output, however small it may be.
- For every state, there is a sequence of inputs that will drive the system back to the same state.

11.29.2 NUMBER OF STATES

Number of states is an important parameter to determine the extent of testing. Some factors affecting number of states are given below.

- All component factors of state.
- All allowable values of the factor.

Thus, number of states is a product of number of allowable values of all the factors.

Impossible States States which appear to be impossible in reality are impossible states. They may not be possible due to some limitations outside the application. These states will never be possible in real life scenario.

Equivalent States When transitions of two or multiple states result into the same final state, they are called 'equivalent states'. Equivalent state can be identified as,

- The rows corresponding to two states are identical with respect to inputs/outputs/next state of an application.
- There are two sets of rows which have identical state graphs.

Unreachable (Redundant) State When the requirements are captured or designs are made, some states are created which fall in a category where control will never reach. They are different from impossible states in the sense that the flow will never reach to them. Impossible states are not practical but unreachable states are redundant.

Dead States Dead state is a state, where there is no reversal possible. These states cannot be left even when the user wishes to come out of them. There is no possibility to come back to the original state.

11.29.3 MATRIX OF GRAPHS

Matrix of graph is a square array with equal number of rows and columns. Each row and column represents one node.

- The size of the matrix corresponds to number of nodes.
- There is a place to put direct relation between every node to any other node.
- Connection from one to another may or may not have connection from another node to original node.

A graph contains a set of nodes and relations between them. If 'A' and 'B' are two nodes and 'R' is the relation between them, it is represented as A-R-B. Different types of relations which may exist, are as follows.

Transitive Relationship Consider a relation between three nodes 'A', 'B' and 'C' such that A-R-B and B-R-C indicate A-R-C, then it is called 'transitive relationship'.

Reflexive Relationship Reflexive relationships are self loops.

Symmetric Relationship Consider a relationship between nodes 'A' and 'B' where A-R-B indicates that there is B-R-A. This relationship is considered as symmetric relationship. This is also called 'undirected relationship'.

Equivalence Relationship If a relationship between two nodes is reflexive, transitive and symmetric, then it is called 'equivalence relationship'.



Tips for special testing (different types of tests)

- Understand the requirement statement clearly. Know and comprehend the scope of testing, type of application, and customer's line of business before devising any test.
- Understand the support documentation, and tools required for doing testing. Performance testing, load testing, and stress testing cannot be done without tools unless the requirements are not very stringent.
- Understand the sequence of activities to be done while conducting different tests. Many tests can be combined, for example, usability testing, error handling, and operations testing may be done along with functional testing.

Summary

In this chapter, we have seen different tests which fall under the category of special tests. They must be defined by requirements and test plan. List of testing can be as given below.

- Complexity testing
- User interface testing
- Compatibility testing
- Security testing
- Installation, uninstallation and upgradation testing
- Requirement (specification) testing
- Regression testing
- Error handling testing
- Manual support system testing
- Intersystem testing
- Control testing
- Smoke testing and sanity testing
- Adhoc testing (exploratory, random or monkey testing)
- Execution testing
- Operations testing
- Compliance testing
- Decision table (axiom) testing
- Documentation testing
- Training testing

- 1) Describe complexity testing.
- 2) Describe user interface testing.
- 3) Describe different types of compatibilities.
- 4) What are the concerns in compatibility testing?
- 5) Describe the challenges faced by testers during internationalisation testing.
- 6) Describe security testing.
- 7) Describe machine backup types.
- 8) Describe system disaster recovery types.
- 9) Describe installation, uninstallation and upgradation testing.
- 10) Explain requirement (specification) testing.
- 11) Explain regression testing.
- 12) Explain error handling testing.
- 13) Explain manual support system testing.
- 14) Explain intersystem testing.
- 15) Explain control testing.
- 16) Differentiate between smoke testing and sanity testing.



CHAPTER 12

SPECIAL TESTS (PART II)



OBJECTIVES

This chapter covers special techniques required for system testing due to specific nature of an application. They may be required as per specific requirements of the customer, specific application, specific development methodology, specific technology, etc.

12.1 INTRODUCTION

There are various types of software development methodologies and software systems in existence as per user requirements or domain in which they are working. These system designs are dependent on system requirements and technological requirements of the users. Let us consider some special

development methodologies and technologies which affect testing approach. They have been termed 'new development and testing methodologies' by some organisations. As the technology is changing every day, the testing approach also needs refinement and redefinition to suite the type of system being developed. Technology and methodology of development may be considered as different by some people, while others may consider that there is a dependency between the two. The approach followed herein considers both technology and methodology of development as same because new methods of development pose similar problems as faced while using new technology.

An organisation must have a definition of new technology. ERM (Enterprise Risk Management) for the organisation may define the risks associated with new technologies. There are various definitions of new technology. New technology may mean any of the following.

- A technology may be new to an organisation but not new to the world. When the first project is done, using any new approach or technology, by an organisation, it may not have required experience in such areas. The baseline figures of performance may not be available. Also, people with required experience may not be available to do the projects. Some organisations have special task force to handle such situation, which are termed 'Research and Development Cell' by few organisations. CMMi describes these groups as 'Technology Change Management' and 'Process Change Management' groups. Sometimes, these groups are sporadic in nature, i.e., they are assembled to solve a particular problem and disassembled once the problem is over. In such cases, an organisation may get some inputs from outside world, e.g., hiring experts in new technology from outside.

- New technology may mean a technology which is completely new to the world. Here, the organisation is on its own and there is no support available from any entity (whether insider or outsider) other than the vendor of that technology. An organisation must have good processes to handle such situations along with possible risk evaluation for using a technology that is new to the world. There may be 'Research and Development Cell' or 'Technology Change Management' and 'Process Change Management' groups, as the case may be.

Let us consider the effect of new technologies on the organisation in terms of addition of risk to developers and users, and changes required in testing process.

12.2 RISK ASSOCIATED WITH NEW TECHNOLOGIES

New technologies are introduced in an organisation due to some benefits which they can bring to the organisation. These technologies also introduce some risks to the development team as well as final users. Risks may be arising from newness of technology to a development organisation, user organisation or to the entire world. Few of them are as defined below.

12.2.1 UNPROVEN TECHNOLOGY

A new technology may be used by an organisation to get the benefits arising out of it. At the same time, it may have some drawbacks or lacunae, which come as part and parcel of the new technology. As the organisation is new to it, it may not be aware of these negative points. Also, it may not be able to use all advantages offered by the new technology. Sometimes, the new technology is not assessed properly for positive and negative factors before it is used. Technology needs some time to mature and as users use it, they keep on giving feedback to owners of technology and thus help in the maturing process.

12.2.2 TECHNOLOGY ITSELF IS DEFECTIVE OR NOT MATURED ENOUGH TO USE

The new technology used by an organisation may be defective as there may have been lot of limitations and unknown aspects overlooked by the group releasing it. Some aspects may have been compromised and people using the new technology may not be aware of the areas where one should be careful. It may introduce some defects in the product, and users may find it difficult to use such products, or may face several problems while using such applications. New technology needs sufficient validation along with verification and fixing of the defects found during these processes.

12.2.3 TECHNOLOGY IS INEFFICIENT AS IT IS NOT MATURED

The new technology may be inefficient, and may not have all the expected services. It may hamper performance and usability of an application adversely. While using such an application, the customer may find it problematic due to ineffective or inefficient way of handling things during development of technology. Generally in final deliveries, one mentions the limiting factors for usage with respect to any technology and methodology. Some of these may be due to use of new technology which may be known as technical limitations. Also, there may be few limitations not known to development team which may surprise the user at some instance.

12.2.4 TECHNOLOGY IS INCOMPATIBLE WITH OTHER TECHNOLOGIES ALREADY IN USE

There are many existing systems communicating with each other at the user's place even before introduction of the new application working with new technologies. Introduction of new systems which cannot communicate

with existing systems may pose a major problem as users may not like to discard existing systems so easily while adapting to new systems. Data transfer and information transfer between different systems may be a basic requirement from user's perspective, and it may become a major problem. Eventually, customer may reject new systems if there is no connectivity/communication offered with existing applications and technologies.

12.2.5 NEW TECHNOLOGY MAKES EXISTING TECHNOLOGY OBSOLETE

Customer may have several other systems already running when a new technology is proposed. If a new system requires certain special aspects which may not be acceptable to existing systems, it may affect the usage of existing applications and hence, may not be welcome. Common users may not like to scrap their existing systems to get a new system in such a scenario.

12.2.6 VARIATION BETWEEN TECHNOLOGY DELIVERED AND DOCUMENTATION PROVIDED

Every technology may be accompanied by huge documentation such as user manuals and troubleshooting manuals etc. Developers and users may refer the documentation provided by owners of new technologies to understand and use them in production. If a gap exists between documentation and technology, then people may not be able to use the technology by referring to documentation available. Documentation may not be in sync with technology, and this may hamper usage of new technology (as people may not know how to use the new technology).

12.2.7 LACK OF DEVELOPER'S/USER'S COMPETENCIES TO USE NEW TECHNOLOGY

A new technology may require people to undergo a training to understand it and develop capabilities for using it. Some skills may be acquired by classroom trainings, some may be acquired by hands-on experience with mentor such as on-the-job training while some may need a special training. New technologies should be user friendly so that people with basic knowledge can handle them with guidance or using trial-and-error approach. If new technologies are difficult at all fronts, users may not like them and may be reluctant to use them.

12.2.8 LACK OF KNOWLEDGE AND SKILL FOR OPTIMAL USAGE OF TECHNOLOGY

Technology may not be usable by the organisation if people with required skills and experiences are not available for using the new technology. An organisation may not have needed information and knowledge base to use these technologies. There may be several options with some limitations available for implementing these technologies. But due to lack of knowledge or skill people may not be aware of using these options.

12.2.9 TECHNOLOGY IS NOT INCORPORATED INTO ORGANISATION'S PROCESS DEFINITION

For a good product, we need a combination of good people, good technologies and good processes together. If organisational database does not have the definition of processes which can support new technologies, then success depends upon the heroics of people and chances. This may lead to major problems with these technologies, and people may try to implement technology with their own methods which may not be the best approach.

12.2.10 TECHNOLOGY LEADS TO OBSOLESCENCE OF EXISTING DEVELOPMENT/TESTING TOOLS

Development tools/test tools available with an organisation may not be able to support new technologies, if there are compatibility issues. In such cases, we may have to rely on manual efforts of development and testing which may be incapable or insufficient for the purpose. New technology may need new tools which means further investment and training for developing organisation/users.

12.2.11 INADEQUATE SUPPORT FOR TECHNOLOGY FROM VENDOR

When an organisation uses new technologies, it may have to depend on the vendor providing service support for such technologies, atleast for initial phases of usage. Service support may include trouble shooting and trainings which may not be available as and when required by development organisation/users. Many aspects depend upon people, both internal as well as external, to make the project successful. Technology becomes a problem as vendor support is not adequate or not available at all.

12.3 PROCESS MATURITY LEVEL OF TECHNOLOGY

Organisations may have different process maturities, and usage of new technologies may be affected by these maturities. The maturity levels may be defined as given below.

12.3.1 LEVEL 1: ADHOC USAGE OF NEW TECHNOLOGY

This is an initial or base maturity level for given technology. In this maturity level, the technologies are used by an organisation depending upon the people skills and experiences available. Some organisations specialise in particular technology, and each and every project is done in that technology irrespective of whether it is the proper technology or not. People will be selecting technology as per their choice and not as per project requirement.

12.3.2 LEVEL 2: MANAGED USAGE OF NEW TECHNOLOGY

Technologies are selected on the basis of evaluation of technologies done by the users. Evaluation is done by referring the support material available or provided by the vendor, or it may be based upon somebody's experience. An organisation works with new technology on the basis of such descriptions without having any hands-on self knowledge with technology. Thus technological approach is designed depending upon the documentation provided without any detailing about usefulness and problems with the technology. This is also called 'static approach of technology selection' or 'evaluation based upon documentation'.

12.3.3 LEVEL 3: DEFINED USAGE OF NEW TECHNOLOGY

An organisation selects a technology on the basis of documentation and conducts experimentation to complete the project using that technology. Returns may not be sufficient but once selected, technology is not changed. Lot of 'research' is done to make selected technology successful. The organisation may learn lessons from its success (or failures) and the same can be used for other projects in similar technology. Making mistakes and correcting them is acceptable in this approach.

12.3.4 LEVEL 4: QUANTITATIVELY MANAGED USAGE OF NEW TECHNOLOGY

At high level of technology process maturity of an organisation, there is a measurement of benefits and limitations of selected technologies for a given project or task. Customer/user is involved in making decisions

about the selection of technology on the basis of benefits and shortcomings analysis. Benefits of new technology are compared with the drawbacks or limitations, and decisions are taken depending upon the analysis.

12.3.5 LEVEL 5: OPTIMISED USE OF TECHNOLOGY

At highest level of maturity, an organisation has a plan to overcome the shortcomings faced by particular technology and enhance the benefits available with technology further for giving better output to the customer. The customer does not suffer due to limitations of technology, or negative points are diluted by optimising efforts, and enhanced benefits from the positives of technologies are obtained.

12.4 TESTING ADEQUACY OF CONTROL IN NEW TECHNOLOGY USAGE

One must be careful about testing new technologies as there may not be sufficient history available. Following few controls are essential while using new technologies.

• Testing Actual Performance Achieved As Against Stated Performance of the Technology By Manufacturer of Technology

Vendor may claim something as attributes or performance parameters while releasing new technology. These claims may include different advantages/services available while using new technology. As a tester, one must check the actual performance standards achieved by the selected technology as against the claimed standards of performance. Any gap found can be termed as a defect or shortcoming in new technology. In 'Level 3' maturity, such limitations will limit the benefits, and claim may be lodged with vendors of technology, if feasible. At 'Level 5' maturity, these limitations are overcome by some alternative approach while claim may be lodged with vendor. Testers must ensure the following.

- Documentation given by technology vendor along with new technology represents actual technology execution. Documentation related defect can create usability issue with technology selected for implementation.
- Training courses about new technology must be effective and complete, so that transfer of needed knowledge to use the technology is effective. Training and level of knowledge transfer must be validated by users of new technology.
- New technology is compatible with existing technology, so that customer may retain old applications along with new one. Compatibility issues, if any, must be addressed.
- Stated performance criteria of the technology must be matching with actual performance criteria reached by the selected technologies.
- Promised vendor support must match with actual vendor support offered while servicing customer/user of new technology. Service levels must be defined and achieved in contract.
- Expected test processes and tools are effective in testing new technologies. If existing tools can be used as it is or with some modifications, then it may add to customer delight.

• Test the Adequacy of Current Process Definition Available to Control Technology

An organisation needs process definition to support new technology implementation and usage. If support is not available, then testers may have to raise issues and get the process definition. It can be a risky situation and customer must be involved in making decisions or customer must be informed about the possible lacunae of process definition. If the processes are available, testers may have to check the usage of these processes to get the outcome. Testers must ensure that,

- Standards for development and usage are available for the selected technology.
- Procedures for development, maintenance, troubleshooting and usage are available.

- Quality control can be ensured in new technology. There must be adequate process coverage for verification/validation.

If one finds that there are no adequate controls, then the actions must be initiated by testing group. Some actions may be as follows

- Identify risks that technologies are not supported by process framework and report them to stakeholders. It may be shared with user groups.
- Identify potential weak areas and try to create mitigation actions, if the risk becomes a reality. There must be adequate contingency plans in case risk materialises.
- Conduct tests when specific practice discovers problems. Problems must be corrected and retested.

- **Assess Adequacy of Staff Skills to Effectively Use Technology** People must be adequately trained and skilled to use the technologies effectively and efficiently. If proper skills are not available, then proper actions must be taken by the organisation to improve the skills or procure people with desired skills. Testers must ensure that,

- Technological process maturity level of an organisation is assessed and organisation leadership knows it. Actions must be initiated to achieve optimising level.
- Training is available for developers and testers for using new technology efficiently and effectively. Effectiveness of training must be evaluated.
- Performance evaluation of technologies on the basis of limitations and benefits must be conducted. Users must be informed about possible shortcomings of the given technology.

12.5 OBJECT-ORIENTED APPLICATION TESTING

Object-oriented development has made a dramatic change in development methodologies. One object may be used at several instances giving a benefit of optimisation, reusability and flexibility. It improves productivity with good maintainability of an application. There are many effective approaches to testing object-oriented software. A test process that complements object-oriented design and programming can significantly increase reuse, quality and productivity of development and testing process.

- As with conventional languages used in development, it results in simple programming mistakes, unanticipated interaction and incorrect or missing behavior of application or individual object at different instances. Reuse in no way guarantees that a sufficient number of paths and states of object have been exercised to reveal all faults in the application built by using objects. Reuse is limited to the extent that supplier classes are trustworthy and can be used at several places during application development.
- While the iterative and incremental nature of object-oriented development is inconsistent with a simple, sequential test process (testing of each unit, then try to integrate the units and test all of them in integration testing, then do system test, etc), it does not mean that testing is completely irrelevant in object oriented development. The boundary that defines the scope of unit testing and integration testing is different for object-oriented development from the orthodox development and testing approaches. Tests can be designed and exercised at many points in the process of development and usage of objects. Thus "design a little, code a little" becomes "design a little, code a little and test a little". The scope of testing corresponds to the shift with the scope of integration from very object oriented-specific for a small scope of the development project and increasingly less object oriented-specific for a larger scope of the development project.

- Adequate testing requires a sophisticated understanding of the system under testing. One must be able to develop an abstract views of the dynamics of control flow, data flow and state space in a formal model used for development. One must have complete understanding of system requirements, at least as good as the designer understands or users understand these requirements. One must be able to define the expected results for any input and state selected as a test case.
- There are many interactions among components that cannot be easily foreseen in unit testing until all components of a system are integrated and exercised through testing. Even if we eliminate all individual sources of error, integration errors are likely to be present in object-oriented development. Compared to conventional systems, object-oriented systems have more components which must be integrated earlier in development processes and tested for correctness as individuals as well as a group when they come together. Since there are elements of a system that are not present until the code has been loaded and exercised, there is no way that all faults could be removed by class or class-cluster testing alone, even if every method was subjected to a formal proof of correctness and unit testing. Static methods cannot reveal interaction errors within different objects with the target or transient performance problems in real-time systems.
- Testing activities can begin and proceed in parallel with concept definition, object oriented architecture, object oriented designs and programming through integration and system testing. When testing is correctly interleaved with development, it adds considerable value to the entire development process as defects are found immediately when they occur.
- The cost of finding and correcting errors is always higher as the time between fault injection and detection increases. The lowest cost is possible when one prevents errors from entering the system. If a fault goes unnoticed, it can easily take hours or days of debugging to diagnose, locate and correct it after the component is in widespread use. Failures in operational systems can cause severe secondary problems including customer complaints, rejection of an application etc. Proper testing is cheaper in comparison to 'find and fix' the defect even when done manually.
- Effective testing is guided by information about likely sources of defects. The combination of polymorphism, inheritance and encapsulation are unique to object-oriented development, creating opportunities for error getting introduced that do not exist in conventional languages. Testing strategy must help testers to look for these new kinds of errors and offer criteria to decide when enough looking is completed.
- Each sub-class is a new and different context for an inherited super-class feature. Different test cases are needed for each context. Inherited features need to be exercised in the unique context of the sub-classes. We need to retest inherited methods, even if they were not changed in individual instance. Although methods may work perfectly well in super-class, the action and interaction may be affected in sub-classes and those must be tested.
- Even if many server objects of a given class function correctly at top level, there is nothing to prevent a new client class from using it incorrectly. Thus, all uses of a server objects need to be exercised at client classes also. An interesting corollary is that we cannot automatically trust a server because it performs correctly for one client and testing may get extended accordingly.

12.6 TESTING OF INTERNAL CONTROLS

- Introduction of software system is a risky proposal from users perspective. There are various failures possible as seen earlier. Testers must be aware of the possible risks when the application will be used by users in production environment. Risk assessment includes understanding of the following.
- Inherent and residual risk acceptable to customer/user using the software. Inherent risks are the born risks of particular approach while residual risks are the risks left after adequate controls are provided.

- Estimating likelihood impact in terms of probability of happening of particular event and possibility of event in terms of percentage occurrence. Also, if the user can see when the risk is approaching, and reaction time is available for him when the risk materialises.
- Qualitative and quantitative measurements of probability, impact and detection ability of different risks associated with application usage can give a risk rating expressed as RPN or RIN. Risks must be taken for prevention/correction, as the case may be.
- Correlation of events where one risk may increase/decrease the probability or impact of some other risk must be known. If one risk increases/decreases probability/impact of other risks, both together may create more problems than individually, or may eliminate the problem.

12.6.1 TESTING OF TRANSACTION PROCESSING CONTROL

An application may process the data received from different inputs, and outputs may be given back to user in different forms. Application controls must be designed to maintain data accuracy in all the phases of data receipts, data processing and outputs. The control placement may include the following.

- **Transaction Origination** The points where the data for processing originates must be controlled. Sometimes, data may originate in other systems and it is transferred to the given system for processing through different methods. Data may originate in manual operations also. Generally these controls may not be applicable to the given system, if data preparation happens outside the system either manually or automatically but data entry may be controlled in such cases.

- **Transaction Entry in System** The point where the data enters into the system for processing must be controlled by the application. There must be adequate user support provided through verification and validation of data entries and availability of help and proper error messaging for common users. Data verification and validation must be done at entry point where data enters the system. This is the most vulnerable point for the system.

- **Transaction Communications Within/Outside the System** When the data is communicated from one place to another either in same system or between different systems, adequate precautions must be taken so that neither the data (completely or partly) is lost during transactions nor something is added or modified. It must measure amount of data transferred from one system to another including data going and coming back from database.

- **Transaction Processing by the System** When the system processes data, it must have adequate control to check completeness of processing. All the data entered into system must be processed, and outcome must contain the results from all data processed. If any part of data is not processed completely or is processed partially, it must notify user about the same.

- **Storage and Retrieval of Data from the Database** Data may be stored on different media and databases and also, physically/logically at different locations. Data entering in the system, and outputs generated from the system must match each other. Data in storage should not be altered, deleted or changed in terms of format changes.

- **Transaction Output** Outputs of processing may be transferred from one system to another or may be given to users in different ways. It may include printing, displaying or sending data from one location to another, or from one system to another. Transactions output must show the processing results correctly and must match data input and data output in data transfer process.

12.6.2 TESTING SECURITY CONTROL

Security controls are more than data controls and processing controls. They are designed for the system, which are exposed to outside/external/internal attacks from possible perpetrators. Security testing is a thought process where one needs to understand the thinking of an intruder and try to device control mechanisms accordingly. One must have an analysis of the following.

- **Points Where Security Is Most Often Penetrated (Points of Penetration)** These are the areas where the system is exposed to outside world and there are possibilities of outside attacks. The points where probability of unwelcome guest entering the system is more are the penetration points for the system. These are termed 'threat points' also. Few threat points may be,

- Data preparation facilities where the data is prepared before it enters into a system. It may be a manual operation point or data preparation in some other systems.
- Computer operations where the system processes or transfers the data from one place to another. These may include server rooms, processors, etc.
- Non-IT areas where people working with system may not understand the system security concept. They may not be aware of requirements of security practices and may cause a system failure.
- Software development, maintenance, and enhancement places where the data is used for building, maintaining or testing of an application
- Online data preparation facilities where there is no adequate validation of data prepared before it enters in the system.
- Digital media storage facilities which may be subject to electro-magnetic fields, temperatures, humidity, etc. Storage facilities may not be secured enough and some storage media may be damaged due to wrong handling.
- Online operations where the data is directly and manually entered in the system without any validation or verification, or data is transferred from one system to another without any control.

Points Where System Is Least Protected (Vulnerable Points) It is not possible and not required to have a system which is protected at all places. There will be few points where the system is least protected. These are called 'vulnerable points' or 'weaker parts' of the system.

Build Risk Matrix (Penetration Point Matrix) One may build a risk matrix for an application depending upon its analysis of vulnerability of, and threats to the system. A typical risk matrix is as shown in Table 12.1

Table 12.1

Risk matrix

Weakness/least protected points	Vulnerable points in the system				
	P x I	P x I	P x I	P x I	P x I
P x I	P x I	P x I	P x I	P x I	P x I
P x I	P x I	P x I	P x I	P x I	P x I
P x I	P x I	P x I	P x I	P x I	P x I

Attributes of Effective Security Control A tester needs to find the effectiveness of security control to protect the users from any external attacks as well as other system failures. Following may be considered as the general attributes of effective security control.

- *Simplicity of Control and Usage* Different controls applied in software system must be very simple for people to understand and use them. Complicated controls affect the system usability adversely. People try to bypass the controls when they are complicated or difficult to follow.
- *Failure Safe Controls* Controls like 'Poka Yoke' are highly desired, where there is a single exit possible. Controls must not fail or probability of their failure must be as less as possible. Failure of control introduces another risk in system and one must evaluate these risks.
- *Open Design for Controls* Due to changes in technologies, one may need to modify the controls time and again. Process change also needs a modification of controls used. Control design must be open to accept the technology changes and process changes accordingly.
- *Separation of Privileges of Users* One person may not be able to do multiple tasks at a time. Typically, where the transaction entry and transaction approval is done by the same person, it can lead to problems as transaction entries can be manipulated. System must incorporate separation of privileges to avoid any wrong entries passing the controls.
- *Psychological Acceptability of Controls by Users* An organisation must plan for adequate training to the users of a system, about security and controls provided so that they can accept the controls. Purpose of controls must be explained to people. If people accept it psychologically, there are lesser chances of problems like bypassing of controls.
- *Layered Defense in System* An entire system must not fail together. If one part of a system fails, there must be some other defense available to detect the failure of control and prevent system failure. If wrong transaction entry is authorised, processing must be able to detect it and must help in initiating proper actions.
- *Compromised Recording* Audit trail is a famous example of compromised recording put in a system. Transactions done by people in the system must be recorded and subjected to checks and audits, if requirements specify the same. This can help in detection of problems, if any, in the transactions and processing. This may hamper the privacy of individuals. But this may be incorporated in the interest of security.

12.7 'COTS' TESTING

'COTS' stands for 'Commercially Of The Shelf' software. These softwares are readily available in the market and user can buy and use them directly. These may be integrated in a new development, or used for development or testing activities as a tool, as the case may be.

12.7.1 WHY SOFTWARE ORGANISATIONS USE COTS?

This is a very natural question one may ask because why would a software development organisation buy and use software from outside? Answer to this question may not be that simple. There are many limitations for a development organisation that prevent it from making its own required software, and one has to make a decision to buy 'COTS'. Some of the reasons are given below.

Line of Business An organisation may not be in a line of business of making such software which it requires. Let us consider an example of a development organisation developing software for banks, financial institutes, etc. It may not be in a line of business to develop automation testing tool for its test requirements.

In such case, the organisation may buy software from outside and use it without investing much time and resources for making such software in-house.

Cost-Benefit Analysis Sometimes, it is very costly to develop software in-house due to various limitations like knowledge, skills, resources, etc. It would be easy to go to the market and buy such software because buying may be more cost effective. Generally, 'COTS' products are much cheaper than the projects, as cost is distributed over number of users.

Expertise/Domain Knowledge An organisation may have knowledge about how to use the software that it needs. But it may not have development knowledge to build such software in-house. Such software can be bought from the market and used without going into the details of how it is built.

Delivery Schedules 'COTS' are available across the table by paying a price while developing such software in-house may take a very long time and huge efforts. Efforts and schedule may not be justified with respect to its use and benefits. Cost of buying such software may be very less compared to the cost of building it in-house.

12.7.2 FEATURES OF COTS TESTING

While testing a 'COTS' product, one must remember the basics of testing in mind. Once 'COTS' is purchased, it cannot be rejected. One may scrap it but money is gone permanently. Testers must remember the following features of COTS.

- 'COTS' are developed with general requirements collected from the market. It may not exactly match with organisation's needs and expectations. One must find the percentage fit of 'COTS' to the organisation business and then decide whether it is successful or not.
- Some 'COTS' may need changing business processes to suite the 'COTS' implementation in organisation. This is another way of Business Process Reengineering (BPR) for an organisation where internationally proven practices can be implemented by using 'COTS'. 'COTS' may have some of their best processes accepted at national/international level, and organisation may get benefited by using such processes along with the product.
- Sometimes, 'COTS' may need configuration of software or system to suite business needs. Generally, when 'COTS' are implemented, many business rules must be defined to customise the software to the organisation.

12.7.3 CHALLENGES IN TESTING 'COTS'

Testing of 'COTS' is a major challenge as the basics methodology of testing may not hold well in such testing. Some of the aspects which testers must remember while testing 'COTS' are as follows.

- Requirement statement and designs may not be available to testers as product manufacturer never shares them with any customer. While buying an operating system, we cannot expect that the requirement statement will be given by the organisation making such products. Generally in normal testing process, test scenario and test cases refer to requirements and design, but in case of 'COTS', they will be referring to business requirements which may differ from product requirements.
- Verification and validation records prepared during SDLC are very important for system testing and acceptance testing. SDLC acceptance reduces the risk of buying a wrong product. But in case of 'COTS', these records may not be available to testers before doing acceptance testing. One must understand software from organisation's perspective while testing it to find whether it can be accepted or rejected. Code reviews, requirement reviews, design reviews, unit testing, and integration testing records are not available to testers.

There are two methods of conducting acceptance testing before buying 'COTS'.

Evaluation of Software Product

- Evaluation of 'COTS' is done by referring to the information available about it from various sources such as white papers and user manuals. There may be various means of obtaining such information which may range from peer experiences, expert's judgment to search on internet. One may refer to white papers, demos available or to the user manuals to understand the 'COTS' usage.
- It is a static measurement of a software to understand its features and suitability for business as there is no hands-on or self-experience of using such product. One may have to make decisions depending upon these documents whether to buy 'COTS' or not.

Assessment of Software Product

- Assessment may mean actually executing some test cases on the product before deciding whether to buy it or not. Many product manufacturers give evaluation versions with limited time period validity so that users can get hands-on and decide about buying/not buying it. Expectation is that the buyer must conduct testing and then decide about the option.
- It is a dynamic measurement to gauge the suitability of the product. There must be test plan, test scenario, and test cases for testing a product. One may have to understand basic requirements or attributes of product before testing it.
- Assessment may need basic understanding of particular type of software and skill of assessment using evaluation versions. Testers must know how to operate new software for assessing it.

12.7.4 'COTS' TEST PROCESS

'COTS' testing involves an excellent knowledge about the process of buying and contracting in addition to technical capabilities of testing an application and the business process where COTS will be used. The following points may be involved in 'COTS' test plan.

- **Assure Completeness of Need Specification** Testers must know the organisational requirements clearly before going for testing of 'COTS'. Requirements may be in various areas as expressed in 'TELOS' (Technical, Economic, Legal, Operational and System).
- Output products and reports which are essential for normal working of the organisation must be defined beforehand. Some 'COTS' can be configured to get desired outputs in desired formats while some may not give such reports and outputs. These may need some external customisation.
- Information needed for management's decision making must be defined beforehand. One may be able to decide percentage fit on the basis of such definitions which may be used while making decision about buying software.
- There may be some statutory/regulatory requirements applicable for the organisation, and COTS must help in achieving them. If COTS does not achieve these requirements, the organisation may have to devise a method to enhance it with external programming or else, reject it.
- **Define Critical Success Factor of Buying** Various products have some specific requirements or expectations. Fulfilling these expectations are essential to decide whether procurement is successful or not.
- Testers must understand why an organisation is buying 'COTS'. Critical success factors are those which can define whether the 'COTS' has fulfilled its intended use or not. Some of the critical success factors may be,

- Ease of use to people in an organisation. If people do not understand English, then software in English is useless.
- Scalability/expandability of product considering future business plans of organisation buying it.
- Cost effectiveness of COTS when cost is compared as against the benefits achieved by implementing it.
- Portability of COTS from one setup to another, either hardware setup or location or software environment.
- Reliability of outcome of data processing.
- Security offered by the COTS.

Determine Compatibility with Environmental Variables An organisation must have definition of working environment where 'COTS' will be implemented. Environment may include hardware, software, and people around it. 'COTS' must fit to the existing environment as there may be several systems already existing in the specified environment. The organisation may not change its environment for implementing any 'COTS' unless there is no way out for it. Compatibility of 'COTS' may include the following aspects.

- **Hardware Compatibility** Environment may be comprised of machines, servers, printers, and communication devices that are existing and used by the organisation before introduction of COTS. It is expected that the new 'COTS' brought in this environment must be able to work without any problem.
- **Operating System Compatibility** Operating systems, browsers and databases used before implementation of 'COTS' should be usable after its implementation. As maximum as possible, data transfer from one system to another may be avoided if there are data compatibility issues. Usage of same database can improve data processing.
- **Software Compatibility** There may be many other softwares existing in system where 'COTS' will be implemented. It should not affect their existence or working. Other softwares may give input/take output from the 'COTS' software. 'COTS' must be able to integrate with them and communicate effectively.
- **Data Compatibility** Data transfer may happen from one system to another system when 'COTS' is implemented by the organisation, and 'COTS' should not hamper the data or formatting. Data formats, styles, and frequency mismatch can lead to severe system problems.
- **Communication Compatibility** Protocols used in communication may be same to avoid any communication loss when two different products are talking with each other. If there is no compatibility of communication protocol, one may need some adopters to convert protocols to facilitate communication.

• Assure That 'COTS' Can Be Integrated with Business

- Manual system working in an organisation may be partly/fully replaced by 'COTS' software. There may be increase/decrease in number of people required and skills required by them due to such implementation.
- Existing processes, methods, forms, formats, and templates used by a manual system may be replaced while implementing 'COTS'. These changes and their effect on normal users, auditors, etc. must be considered while deciding to implement 'COTS' in an organisation.
- There may be addition/deletion of steps/sub-processes due to introduction of 'COTS' in an organisation and users must be comfortable with such changes. Addition of steps may be resisted by people, if they find it unnecessary.

Demonstrating 'COTS' in Operation Similar to projects, there exists alpha and beta acceptance testing of product. 'COTS' may be demonstrated at two places, viz. vendor site demonstration representing alpha testing, and actual usage site demonstration representing beta testing.

- **Demonstration at Vendor Site** These demonstrations may be done using sample data provided by vendor or purchaser. The objective of such demonstration is to give basic awareness to user/purchaser about the software. It is used as training to users about new product.
- **Demonstration at Customer Site** These demonstrations may be done by using customer supplied data or real-time production data. Users are involved in demonstrations, and experts may guide them about how to use a new product. Users may get hands-on experience under the supervision of experts. This also exposes users to software and gives basic training of how to use it. Users know advantages as well as limitations of software, if any, during such demonstrations.
- **Evaluate People Fit** Testers need to analyse whether people would be able to work with the system effectively or not. One must understand whether software can be used as it is, or needs some configurations, modification or external changes to make it usable. Sometimes, additional training and support may be required by common users for working with software in real life.

12.8 CLIENT-SERVER TESTING

Client-server is an initial improvement from stand-alone applications where there are several clients communicating with the server. There are many advantages of client-server over stand-alone application. In its simplest form, client-server architecture gives an opportunity for number of users to work with software at a time. In simpler terms, client-server system may be viewed as—the requests are coming from number of clients for doing some actions and server is serving these requests.

12.8.1 FEATURES OF CLIENT-SERVER APPLICATION

- There are two (three) parts of system, viz. clients connected to the server (the third part is a network).
- Clients may be thick clients or thin clients as per the level of activities/actions done by them.
- Multiple users can use the system at a time and they can communicate with the server. Sometimes, clients may be able to communicate with each other via the server.
- Configuration of client is known to the server beforehand with certainty.
- Client and server are connected by real connection.

12.8.2 TESTING APPROACH OF CLIENT-SERVER SYSTEM

Client-server involves component testing and integration testing followed by various specialised testing, as per scope of testing involved.

Component Testing One needs to define the approach and test plan for testing client and server individually. One may have to devise simulators to replace corresponding components while testing the component targeted by test. When server is tested, we may need a client simulator, while testing of client may need a server simulator. We may have to test network by using client and server simulators at a time.

Integration Testing After successful testing of servers, clients and network, they are brought together to form the system, and system test cases are executed. Communication between client and server is tested in integration testing.

There are regular testing methods like functionality testing and user interface testing to ensure that system meets the requirement specifications and design specifications correctly. In addition to this, there are several special testing involved in client-server application. Some of these are given below.

Performance Testing System performance is tested when number of clients are communicating with server at a time. Similarly, volume testing and stress testing may be used for testing client-server applications. Since number of clients is already known to system, we can test the system under maximum load as well as normal load expected. Various user interactions may be used for stress testing.

Concurrency Testing Concurrency testing is a very important testing for client-server architecture. It may be possible that multiple users may be accessing same record at a time, and concurrency testing is required to understand the behavior of a system under such circumstances.

Disaster Recovery/Business Continuity Testing When the client and server are communicating with each other, there exists a possibility of breaking of the communication due to various reasons or failure of either client or server or link connecting them. Test for disaster recovery and business continuity may be involved to understand how system behaves in such cases of disaster. It may involve testing the scenario of such failures at different points in the system, and actions taken by the system in each case. The requirement specifications must describe the possible expectations in case of any failure.

Testing for Extended Periods In case of client-server applications, generally, server is never shut down unless there is some agreed (Service Level Agreement) SLA where server may be shut down for maintenance. It may be expected that server is running 24×7 for extended period. One needs to conduct testing over an extended period to understand if service level of network and server deteriorates over a time due to some reasons like memory leakage.

Compatibility Testing Client and server may be put in different environments when the users are using them in production. Servers may be in different hardware, software, or operating system environment than the recommended one. Clients may differ significantly from the expected environmental variables. Testing must ensure that performance is maintained on the range of hardware and software configurations, and users must be adequately protected in case of configuration mismatch. Similarly, any limiting factors must be informed to prospective user.

Other testing such as security testing and compliance testing may be involved if needed, as per scope of testing and type of system.

12.9 WEB APPLICATION TESTING

Web application is further improvement in client-server applications where the clients can communicate with servers through virtual connectivity. It has many advantages over a client-server application as multiple server networks can be accessed at a time from the same client. It improves communication between people at different places significantly.

12.9.1 FEATURES OF WEB APPLICATION

- Number of clients connecting to a server, is very large. Sometimes, number may tend to infinity. Client configurations cannot be controlled by definition.
- Different clients may have different configurations, and server may not be able to talk with them directly. There is a special arrangement required to overcome this. There is a universal client called ‘browser’ required in such circumstances.
- Communication protocols may differ from system to system. Sometimes, adopters are required to convert these communication protocols so that communication can be effected.

- Client and server are connected through world wide web cloud and there is no direct physical connectivity. As they are connected virtually, communication needs address where that communication is expected to reach, from client to server as well as from server to client.
- One client may be able to connect several servers at a time. This helps in faster communication between different domains.
- Generally, there is no client piece installation other than presence of browser as a universal client. Almost all the working is done by the server. Clients are extra thin.

12.9.2 TESTING APPROACH OF WEB APPLICATION

Web application involves component testing, integration testing, functionality testing, and GUI testing followed by various specialised testing.

Component Testing One must define the approach and test plan for testing web application individually at client side and at server side. One may have to devise simulators to replace corresponding components. When server is tested, we may need a client simulator while testing of client may need a server simulator. Network testing is also required.

Integration Testing Successfully tested servers and clients are brought together to form the web system and system test cases are executed. Communication between client and server are tested in integration testing.

There are regular testings like functionality testing and user interface testing to ensure that system meets the requirement specifications and design specifications correctly. In addition to this, there are several special testings involved in web application. Some of these are mentioned below.

Performance Testing System performance is tested as huge number of clients may be communicating with server simultaneously. Similarly, volume testing and stress testing may be used for testing these applications. Since there is no possibility of definition of maximum number of users, system requirement specifications must define maximum and normal load conditions so that they can be tested. Simulators are used extensively for such testing.

Concurrency Testing It may be possible that multiple users may be accessing same records at a time, and concurrency testing is required to understand the behavior of a system under such circumstances. Probability of concurrency is very high as number of users is very large.

Disaster Recovery/Business Continuity Testing When the machine is communicating with the web server, there exists a possibility of breaking of communication due to various reasons like breaking of connectivity, client failure, and server failure. Testing for disaster recovery and business continuity involves testing the scenario of such failures and actions taken by the system in each case. The requirement specifications must describe the possible expectations of system behavior in case of any failure. MTTR (Mean Time To Repair) and MTBF (Mean Time Between Failures) are very important tests for web applications.

Testing for Extended Periods In case of web applications, generally, the server is never shut down. It is expected to run 24×7 over extended period of time, and there must be a provision of alternate server (hot recovery/mirroring) if requirements specify the same. One needs to conduct testing over an extended period to understand if service level of server deteriorates over a time due to some reasons like memory leakage.

Security Testing As the communication is through virtual network, security becomes an important issue. Applications may use communication protocols, coding and decoding mechanisms, and schemes to maintain security of system. System must be tested for possible weak areas called 'vulnerabilities' and possible intruders trying to attack the system called 'perpetrators'.

Compatibility Testing Web applications may be put in different environments when the users are using them in production. Servers may be in different hardware, software, or operating system environment than the recommended one. Client browsers may differ significantly from the expected environmental variables. Testing must ensure that performance is maintained on the range of hardware and software configurations, and users must be adequately protected in case of configuration mismatch. Similarly, any limiting factors must be informed to prospective user.

12.10 MOBILE APPLICATION TESTING (PDA DEVICES)

Now-a-days, pocket devices are used widely for communication and computing. Pocket devices can be put into many uses due to mobility offered by them. There is tremendous increase in memory levels and technologies adopted by such appliances. Because of developing technologies like Blue tooth and Wi Fi, many PDAs are replacing desktop computers as they are more convenient for usage. New technologies have converted normal communication device into internet-driven palm tops.

12.10.1 TESTING LIMITATIONS OF PDAS

Scenarios designed for Web testing do not necessarily translate in the same way to PDA testing. There is a change in behavior and usage patterns on PDA. There are few limitations on PDA due to various factors.

- Hardware and software used on PDAs vary significantly from one another as there is less standardisation, and manufacturers have different preferences. Usability testing on PDAs needs to take into account variability in hardware and software as well as configurations.
- Memory available with PDAs is limited in comparison to desktops. Now there is tremendous increase in memory availability for PDAs but still that is much less than the memory availability as compared to normal computers.
- Even after invention of touchpad and joystick, usability of PDAs is limited with respect to normal desktops.
- Convenience of keyboard and mouse is difficult in PDAs.
- Data input has many limitations as single key may mean different inputs depending upon the number of times it is pressed. There is limitation for providing help and maintaining operability of PDA due to its small size.
- Lighting may be available for limited time as it has direct relationship with battery life. Similarly, battery capacity may be limited.
- Bandwidth available with Blue tooth and Wi Fi may be another challenge faced by PDAs as it is much less than bandwidth availability by other means like optical cables.

12.10.2 INTERFACE DESIGN OF PDAS

- The smaller size and resolution of the PDA screen presents usability challenges to testers. Reading from the screen is not easy. Scrolling up and down is very inconvenient.

- Instructions and other text must be used sparingly and only when necessary, as they may occupy the screen and scrolling may become necessary. Instructions cause content to be pushed below the fold, which can then be missed by users.
- Links must be very brief and containing only necessary key words. Generally, specific options must be presented before general options to improve usability.

12.11 eBUSINESS/eCOMMERCE TESTING

Due to change in way business is done and improvements in technology, eBusiness/eCommerce applications are becoming very famous now-a-days. There is a small difference between eBusiness and eCommerce application. While eCommerce concerns mainly with money transactions, eBusiness concerns with all aspects of business including money, advertising, and sales.

12.11.1 DISTINCT PARTS OF eBUSINESS

- **Information Access** by the common user is very important from business point of view. All the products available with enterprise must be accessible—to users who wish to buy them—with their information including price. Quick links may be provided to reach the areas of interest to buyer faster like grouping items into some categories and carting.
- **Self Services** are provided to users where the user can select an item and quantity, and make online payment accordingly. They are expected to select things they wish to purchase by going to different areas. Carting arrangements are used extensively.
- **Shopping Services** including advertisements, carting, and billing are done online where users can avail self-service arrangement. Users must be able to do all transactions as if they are buying something in a real shop.
- **Interpersonal Communication Services** can be used to tell users about the billing amount, or tell shopkeeper/user about the stocks availability and demand trends. It can be used to give more information about the products available for sale.
- **eBusiness** represents a virtual enterprise where all the actions are done by users through internet which replaces actual shopping activities happening in physical transactions.

12.11.2 TESTING APPROACH FOR eBUSINESS/eCOMMERCE

- Software applications are challenged to meet expectations of usability, performance and reliability which are critical success factors for such applications. People may not use the application or perform any transaction, if it is not user friendly. If performance is bad, application may be rejected as users may not use it, and business is directly affected. Similarly, it must give consistent results again and again.
- Generally, these applications are controlled by regulatory and statutory requirements imposed by various governments at different places and at different times. Regulatory and statutory requirements are very important in eBusiness and eCommerce as violation of such requirements is a legal offence.

Security and privacy are very important in eBusiness and eCommerce applications and these may be attached as a part of statutory requirements. Users must feel that their personal information and information about transactions are not disclosed to any third party and anonymity is maintained.

Non-functional requirements such as performance of a system, user interfaces, and online help are more important in addition to functional requirements. Look and feel, working speed, and disaster recovery ability are important aspects from user's perspective.

12.11.3 eCOMMERCE QUALITY CHALLENGES

- Stringent quality standards are associated with eBusiness sites and applications that survive for a long time. If people like the site and have a positive feeling about it, they may use it again and again. If there are several problems, business will be in problem as there is no physical store in existence.
- If the visitor experience about an application is negative due to slow response times, outright crashes, and violations of privacy then consumer confidence will be lost and users will not feel confident to use this site for transactions. If users are not confident, system usage and business may be in trouble.

12.11.4 eBUSINESS/eCOMMERCE DEVELOPMENT

Development of eBusiness/eCommerce is a specific way of doing things as required by business or consumers. It may not follow waterfall or iterative development. Agile methodologies and spiral development models are used extensively. It is characterised by the following.

- Rapid and easy assembly of application modules is essential as the system size increases as defined by spiral methodology. Initially, some parts of an application are put in use, and as per user response and expectations, the latter parts are then added at multiple instances.
- Testing of component functionality and performance at each increment is required to ensure correct integration. Huge regression testing cycles may be required.
- Designing models to simulate real-world scenario where normal users will be using the application is essential as there may not be real-time testing. Simulation is most important for testing and is also a major risk.
- Generally deployment of such application is into a distributed environment, $24 \times 7 \times 365$ for an extended period, and there is no (as less as possible) downtime allowable. MTTR and MTBF are very important parameters for assuring service to users.
- Monitoring performance and transactions over an extended period is essential to understand if there is any deterioration of service level over a time span.
- Analysing effectiveness of system and gathering business intelligence may be essential to maximise sale or maximise profit as the case may be.

12.11.5 INCORPORATING LEGAL STANDARDS

Statutory and legal requirements are one of the most important parts for eBusiness and eCommerce applications. One must understand all these requirements and implement them in an application. Any violation of these requirements is punishable under law. More importantly, they keep on changing from time to time as per policy and strategy decisions of government and categories of industry that they are applied to.

- All critical business functions with respect to common users are identified and evaluated on the basis of their criticality.
- Mechanisms must be in place to ensure connectivity and handling of loss of connections, or disaster recovery procedures as well as business continuity procedures are developed.
- Systems are tested to assess the security of online transactions. User information must be protected and privacy practices must be applied stringently.
- Privacy audits must be conducted to confirm that strategies for protecting customer privacy and confidentiality have worked as expected.
- Vulnerabilities are analysed to prevent hacker attacks and virus attacks. Users may not use the site, if there are possible vulnerabilities which are exposed to threats.

- Disaster avoidance measures are developed, such as redundant systems, alternative routing, precise change controls, encryption, capacity planning, load and stress testing, and access control.

12.12 AGILE DEVELOPMENT TESTING

'Agile development' is becoming a famous word in software development. Agile development talks about agile manifesto which works of some principles. Many organisations adopt agile development without going into the details of what an agile development is. Even after knowing and understanding agile approach of development, one may have to assess the organisation's readiness to adopt such approach.

Some of the agile principles are given below.

- Delivering working software to users in place of getting requirements signed off from customer.
- Adopting and embracing changes in requirements in place of scope definitions and change management to deliver what is required for the project.
- More stress on communicating effectively between various stakeholders.

12.12.1 AGILE TESTING

There are various ways under the umbrella of agile development such as scrum, extreme programming, feature-driven development, and test-driven development etc. One may have to adopt test plans to fit the methodologies used and purpose. Also, one may have to keep the basic agile principle of delivering working software at faster speed.

12.12.2 CRITICAL POINTS FOR AGILE TESTING

Testing is very critical from agile perspective. Typically in scrum, there are many changes and test team must be capable of handling huge regression testing cycle as one progresses from iteration to iteration. Understanding of requirements, creation of reusable test cases, integration testing along with regression testing are key factors in successful testing.

- **Competencies/Maturity of Agile Development and Test Team** For undertaking agile, one must have the teams, customer and management who psychologically accept agile approach. It talks about ability to change very fast, build good working product and communicate with team members and stakeholders effectively as well as efficiently. Agile implementation may prefer generalist approach as against specialist approach. It needs people with very high maturity as well as technical competence to adapt to changing needs of customer.
- **Development and Test Process Variability** Every process has an inborn variability. One may have to attack the generic reasons of variations while there may be some controls to identify special causes of variations. One must be able to plot development and test process, and try to remove personal factor from the processes.
- **Change Management and Communication** Change is inevitable in agile. It flows from customer to development team and goes back to customer. There must be a very close communication between development team, test team, customer, and other stakeholders to adapt to changing scenario. Requirement change must be welcomed and all people together must decide how customer can be served best.
- **Test Process Flexibility** Change is must in agile, and one may have to adapt to the changes. Test process is not an exception to it. Different parts of software need different strategies of testing. There may be

different test plans or one may keep flexibility in a test plan to adapt to these changes. There may be changes in focus in each iteration. Initially, there may be heavy unit testing, then it may have integration testing where different iterations come together. It may be followed by heavy regression testing.

- Focus on Business Objective** There is always a time pressure in agile development. Pressure may come from stakeholders to deliver things faster or it may come from development, if they get delayed. One may have to focus on business while defining test process. Cost-benefit analysis may be done when it comes to defect fixes and release of software. Nobody can find all defects but user must be protected from any accidental failure. Testing has to achieve both extremes.

- Stakeholder Maturity/Involvement** Agile development also needs a good maturity from stakeholders. Internal and external service providers must understand and work with time pressure. Good process of development and testing must be supported by tools and techniques required for agile implementation. There may be some specific requirements of stakeholders, and these requirements must be rearranged to suite agile development.

12.13 DATA WAREHOUSING TESTING

A data warehouse is a repository of an organisation's electronically stored data. Data warehouses are designed to facilitate reporting and analysis.

This classic definition of data warehouse focuses on data storage. However, the means to retrieve and analyse data, to extract, transform and load data, and to manage the data dictionary are also considered essential components of a data warehousing system. The important factor leading to the use of a data warehouse is that a data analyst can perform complex queries and analysis (data mining) on the information within data warehouse without slowing down the operational systems.

12.13.1 DATA WAREHOUSE DEFINITION

Data in data warehouse is subjected to few definitions.

- Subject-oriented** Subject-oriented data warehouses are designed to help the user in analysing data. The data is organised so that all the data elements relating to the same real-world event or object are linked together.

Illustration 12.1

To learn more about company's sales data, one can build a warehouse that concentrates on sales. Using this warehouse, one can answer questions like, "Who was the best customer for this item last year?" This ability to define a data warehouse by subject matter makes the data warehouse subject-oriented.

- Integrated** Integration of database is closely related to subject orientation. Data warehouses must put data from different sources into a consistent format. The database contains data from most or all of an organisation's operational applications and is made consistent.

- **Time-variant** The changes to the data in the database are tracked and recorded to produce reports on data changed over time. In order to discover trends in business, analysts need large amounts of data. A data warehouse's focus on change over time is what is meant by the term 'time variant'.

- **Non-volatile** Data in the database is never over-written or deleted, once committed—the data is static and read-only but retained for future reporting. Once entered into the warehouse, data should not change. This is logical because the purpose of a data warehouse is to enable one to analyse what has occurred.

12.13.2 BENEFITS OF DATA WAREHOUSING

Some of the benefits that a data warehouse provides are given below.

- A data warehouse provides a common data model for all data of interest regardless of its source. This makes it easier to report and analyse information than it would be if multiple data models were used to retrieve information.
- Prior to loading data into the data warehouse, inconsistencies are identified and resolved. This greatly simplifies reporting and analysis.
- Information in the data warehouse is under the control of data warehouse users so that, even if the source system data is purged over time, the information in the warehouse can be stored safely for extended periods of time.
- Because they are separate from operational systems, data warehouses provide retrieval of data without slowing down operational systems.
- Data warehouses facilitate decision support system applications such as trend reports, exception reports and reports that show actual performance versus goals.

12.13.3 TESTING PROCESS FOR DATA WAREHOUSE

Testing for a data warehouse consists of requirements testing, unit testing, integration testing followed by acceptance testing.

Requirements Testing The main aim for performing requirements testing is to check stated requirements for completeness. In a data warehouse, the requirements are mostly around reporting. Hence, it becomes more important to verify whether these reporting requirements can be catered using the data available. Successful requirements are structured closely to business rules, and address functionality and performance expected by users.

Unit Testing Unit testing will involve the following.

- whether the application is accessing and picking up right data from right source as expected by the users
- All the data transformations are correct according to the business rules and data warehouse is correctly populated with the transformed data.
- Testing the rejected records that do not fulfill transformation rules.

Integration Testing After unit testing is complete, it should form the basis of starting integration testing. Integration testing should test initial and incremental loading of the data warehouse. Integration testing will involve the following.

Verify Report Data with Source Although the data present in a data warehouse will be stored at an aggregate level yet it must be compared to source systems. Here, test team must verify the granular data stored in data warehouse against the source data available.

• **Field Level Data Verification** Test team must understand the linkages for the fields displayed in the report and must trace back and compare that with the source systems.

• **Creating Queries** Create queries to fetch and verify the data from source and target. Sometimes, it is not possible to do the complex transformations done in ETL. In such a case, the data can be transferred to some file and calculations can be performed.

• **Data Completeness** Basic test of data completeness is to verify that all expected data loads into the data warehouse. This includes validating that all records, all fields and the full contents of each field are loaded. This may cover,

- Comparing record counts between source data, data loaded to the warehouse and rejected records.
- Comparing unique values of key fields between source data and data loaded to the warehouse.
- Utilising a data profiling tool that shows the range and value distributions of fields in a data set.
- Populating the full contents of each field to validate that no truncation occurs at any step in the process.
- Testing the boundaries of each field to find any database limitations.

• **Data Transformation** Validating that data is transformed correctly from database is based on business rules. This can be the complex part of testing.

- Create a spreadsheet of scenarios of input data and expected results, and validate these with the business customer.
- Create test data that includes all scenarios.
- Utilise data profiling results to compare range and distribution of values in each field between source and target data.
- Validate correct processing.
- Validate that data types in the warehouse are as specified in the design and/or the data model.
- Set up data scenarios that test referential integrity between tables.
- Validate parent-to-child relationships in the data.

• **Data Quality** Data quality rules are defined during design of data warehouse application. It may include,

- Reject the record if a certain decimal field has non-numeric data.
- Substitute null if a certain decimal field has non-numeric data.
- Duplicate records.

Performance and Scalability As the volume of data in a data warehouse grows, load times can be expected to increase and performance of queries can be expected to degrade. The aim of performance testing is to point out any potential weaknesses in the design, such as reading a file multiple times or creating unnecessary intermediate files.

- Load the database with peak expected production volumes to ensure that this volume of data can be loaded within the agreed-upon time. This can be a part of SLA definition.
- Compare these loading times to loads performed with a smaller amount of data to anticipate scalability issues.
- Monitor the timing of the reject process and consider how large volumes of rejected data will be handled.
- Perform simple and multiple join queries to validate query performance on large database volumes.



Tips of special testing (different types of systems)

- Understand the requirement statement clearly. Understand the scope of testing, type of application, and customer's line of business before devising any test.
- Understand the support documentation, tools required for doing testing, and different tests expected by users at different levels of development life cycle. Performance testing, load testing, and stress testing cannot be done without tools unless the requirements are not very stringent.
- Understand the types of users, their abilities and purpose for using such applications. Identify the risks connected with each type of system and create test strategy accordingly.

Summary

In this chapter, we have seen evolution of new technologies and new techniques of software development and how testing gets affected due to them. The chapter also covers what is meant by new technology, and the possible risks due to introduction of such technologies. It also deals with organisational process maturity with respect to evolving technologies. The testing approaches of new technology covered following methodologies and business applications.

- Object-oriented development
- Internal controls
- Commercially Of The Shelf (COTS) softwares
- Client-server testing
- Web application
- PDAs/Mobile applications
- eCommerce and eBusiness
- Data warehouse systems

- 1) Describe the risk associated with new technology usage.
- 2) Explain technology process maturity.
- 3) Explain the test process for new technology.
- 4) Explain the process of testing object-oriented development.
- 5) Explain the process of testing of internal controls.
- 6) Explain how transaction processing controls can be tested.
- 7) Explain the process of testing security controls.
- 8) What are the attributes of a good control?
- 9) Why software organisations buy 'COTS' softwares?
- 10) Explain 'COTS' testing process.
- 11) What are the challenges in COTS testing?
- 12) Differentiate between evaluation and assessment of COTS.
- 13) Describe a process of client-server testing.
- 14) Describe the testing process for web application.
- 15) Describe the mobile applications (PDA) testing process.
- 16) Describe the critical quality issues of eCommerce and eBusiness.
- 17) Describe the process of testing data warehouse systems.

