

[Cover](#)

[About the Author](#)

[Copyright](#)

[Acknowledgements](#)

[ISTQB Copyright](#)

[Preface](#)

[Introduction](#)

[Chapter 1. Test Basics](#)

[1.1. Introduction](#)

[1.2. Testing in the Software Lifecycle](#)

[1.3. Sequential Lifecycle Models](#)

[1.4. Iterative or Incremental Lifecycle Models](#)

[1.5. Spiral Lifecycle Models](#)

[1.6. Test Levels](#)

[1.7. Specific Systems](#)

[1.8. Metrics and Measurement](#)

[1.9. Ethics](#)

[1.10. Sample Exam Questions](#)

[Chapter 2. Testing Processes](#)

[2.1. Introduction](#)

[2.2. Test Process Models](#)

[2.3. Test Planning and Control](#)

[2.4. Test Analysis and Design](#)

[2.5. Test Implementation and Execution](#)

[2.6. Evaluating Exit Criteria and Reporting](#)

[2.7. Test Closure Activities](#)

[2.8. Sample Exam Questions](#)

[Chapter 3. Test Management](#)

[3.1. Introduction](#)

[3.2. Risk-Based Testing and Failure Mode and Effects Analysis](#)

3.3. Test Management Documentation and Test Plan Documentation Templates

3.4. Test Estimation

3.5. Scheduling Test Planning

3.6. Test Progress Monitoring and Control

3.7. Business Value of Testing

3.8. Distributed, Outsourced, and Insourced Testing

3.9. Test Management Issues

3.10. Nonfunctional Testing Issues

3.11. Sample Exam Questions

Chapter 4. Test Techniques

Chapter 5. Tests of Software Characteristics

Chapter 6. Reviews

6.1. Introduction

6.2. The Principles of Reviews

6.3. Types of Reviews

6.4. Introducing Reviews

6.5. Success Factors for Reviews

6.6. Sample Exam Questions

Chapter 7. Incident Management

7.1. Introduction

7.2. When Can a Defect Be Detected?

7.3. Defect Lifecycle

7.4. Defect Fields

7.5. Metrics and Incident Management

7.6. Communicating Incidents

7.7. Sample Exam Questions

Chapter 8. Standards and Test Process Improvement

8.1. Introduction

8.2. Standards Considerations

8.3. Test Improvement Process

[8.4. Improving the Test Process](#)

[8.5. Improving the Test Process with TMM](#)

[8.6. Improving the Test Process with TPI](#)

[8.7. Improving the Test Process with CTP](#)

[8.8. Improving the Test Process with STEP](#)

[8.9. Capability Maturity Model Integration, CMMI](#)

[8.10. Test Improvement Process Exercise](#)

[8.11. Test Improvement Process Exercise Debrief](#)

[8.12. Sample Exam Questions](#)

[Chapter 9. Test Tools and Automation](#)

[9.1. Introduction](#)

[9.2. Test Tool Concepts](#)

[9.3. Test Tool Categories](#)

[9.4. Sample Exam Questions](#)

[Chapter 10. People Skills and Team Composition](#)

[10.1. Introduction](#)

[10.2. Individual Skills](#)

[10.3. Test Team Dynamics](#)

[10.4. Fitting Testing within an Organization](#)

[10.5. Motivation](#)

[10.6. Communication](#)

[10.7. Sample Exam Questions](#)

[Chapter 11. Preparing for the Exam](#)

[11.1. Learning Objectives](#)

[11.2. ISTQB Advanced Exams](#)

[Appendix A. Bibliography](#)

[Advanced Syllabus Referenced Standards](#)

[Advanced Syllabus Referenced Books](#)

[Other Referenced Books](#)

[Other References](#)

[Appendix B. HELLOCARMS The Next Generation of Home Equity](#)

Lending

System Requirements Document

Table of Contents

II Versioning

III Glossary

000 Introduction

001 Informal Use Case

003 Scope

004 System Business Benefits

010 Functional System Requirements

020 Reliability System Requirements

030 Usability System Requirements

040 Efficiency System Requirements

050 Maintainability System Requirements

060 Portability System Requirements

A Acknowledgement

Appendix C. Answers to Sample Questions

Index



Advanced Software Testing Vol. 2: Guide to the ISTQB Advanced Certification as an Advanced Test Manager

by Rex Black

Publisher: **Rocky Nook**

Pub Date: **December 28, 2008**

Print ISBN: **978-1-933952-36-9**

Web ISBN: **1-933952-36-9**

Pages: **570**

Overview

This book teaches test managers what they need to know to achieve advanced skills in test estimation, test planning, test monitoring, and test control. Readers will learn how to define the overall testing goals and strategies for the systems being tested.

This hands-on, exercise-rich book provides experience with planning, scheduling, and tracking these tasks. You'll be able to describe and organize the necessary activities as well as learn to select, acquire, and assign adequate resources for testing tasks. Learn how to form, organize, and lead testing teams. Master the organizing of communication among the members of the testing teams, and between the testing teams and all the other stakeholders. Additionally, you'll learn how to justify decisions and provide adequate reporting information where applicable. Target Audience: Software Testers, Programmers, System Designers, IT Managers

About the Author



With a quarter-century of software and systems engineering experience, **Rex Black** is President of RBCS (www.rbcus-us.com), a leader in software, hardware, and systems testing. For over a dozen years, RBCS has delivered services in consulting, outsourcing, and training for software and hardware testing. Employing the industry's most experienced and recognized consultants, RBCS conducts product testing; builds and improves testing groups; and hires testing staff for hundreds of clients worldwide. Ranging from Fortune 20 companies to start-ups, RBCS's clients save time and money through improved product development, decreased tech support calls, improved corporate reputation, and more.

As the leader of RBCS, Rex is the most prolific author practicing in the field of software testing today. His popular first book, *Managing the Testing Process*, has sold over 30, 000 copies around the world, including Japanese, Chinese, and Indian releases. His three other books on testing, *Critical Testing Processes*, *Foundations of Software Testing*, and *Pragmatic Software Testing*, have also sold tens of thousands of copies, including Hebrew, Indian, Chinese, -Japanese and Russian editions. He has written over twenty-five articles, presented hundreds of papers, workshops, and seminars, and given about thirty keynote speeches at conferences and events around the world. Rex is the President of the International Software Testing Qualifications Board (ISTQB) and a Director of the American Software Testing Qualifications Board (ASTQB).

Copyright

Rex Black

rex_black@rbcs-us.com

Editor: Jimi DeRouen

Copyeditor: Judy Flynn, Santa Barbara, USA

Layout and Type: Gerry Ichikawa, Santa Barbara

Indexer: Seth Maisin, Potomac Indexing

Proofreader: Carey Feldstein

Cover Design: Helmut Kraus, www.exclam.de

Printed in the United States of America

ISBN-13: 978-1-933952-36-9

1st Edition © 2009 by Rex Black

16 15 14 13 12 11 10 09 08 1 2 3 4 5

Rock Nook Inc.

26 West Mission Street Ste 3

Santa Barbara, CA 93101-2432

www.rockynook.com

Distributed by O'Reilly Media

1005 Gravenstein Highway North

Sebastopol, CA 95472-2811

All product names and services identified throughout this book are trademarks or registered trademarks of their respective companies. They are used throughout this book in editorial fashion only and for the benefit of such companies. No such uses, or the use of any trade name, is intended to convey endorsement or other affiliation with the book. No part of the

material protected by this copyright notice may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

This book is printed on acid-free paper.

Acknowledgements

A complete list of people who deserve thanks for helping me along in my career as a test professional would probably make up its own small book. Here I'll confine myself to those who had an immediate impact on my ability to write this particular book.

First of all, I'd like to thank my colleagues on the American Software Testing Qualifications Board, the International Software Testing Qualifications Board, and especially the Advanced Syllabus Working Party, who made this book possible by creating the process and the material from which this book grew. Not only has it been a real pleasure sharing ideas with and learning from each of the participants, but I have had the distinct honor of twice being elected president of both the American Software Testing Qualifications Board and the Inter-national Software Testing Qualifications Board. As I approach the end of my statutory last term as president of the International Software Testing Qualifications Board, I look back with pride at our accomplishments so far, I look forward with pride to what we'll accomplish together in the future, and I hope this book serves as a suitable expression of the gratitude and professional pride I feel toward what we have done for the field of software testing.

Next, I'd like to thank the people who helped me create the material that grew into this book. The materials in this book, our Advanced Test Manager instructor-led training course, and our Advanced Test Manager e-learning course were reviewed, re-reviewed, and polished with hours of dedicated assistance by José Mata, Judy McKay, Jamie Mitchell, Paul Jorgensen, and Pat -Masters.

Now, once I had created the materials, the task of assembling the first rough draft of this book from scripts, slides, the syllabus, and a rough framework fell to Dena Pauletti, RBCS's extremely competent and meticulous technical administrator. This book would have taken literally months longer to prepare without her intrepid and timely assistance.

Of course, the Advanced syllabus could not exist without a foundation, - specifically the ISTQB Foundation syllabus. I had the honor of working with that Working Party as well. I thank them for their excellent work over the years, creating the fertile soil from which the Advanced syllabus and thus this book sprang.

In the creation of the training courses and the materials that make up this

book, I have drawn on all the experiences I have had as an author, practitioner, consultant, and trainer. So I have benefited from individuals too numerous to list. I thank each of you who have bought one of my previous books, for you contributed to my skills as a writer. I thank each of you who have worked with me on a project, for you have contributed to my abilities as a test manager, test analyst, and technical test analyst. I thank each of you who have hired me to work with you as a consultant, for you have given me the opportunity to learn from your organizations. I thank each of you who have taken a training course from me, for you have collectively taught me much more than I taught each of you. I thank my readers, colleagues, clients, and students and hope that my contributions to each of you have repaid the debt of gratitude that I owe you.

For over a dozen years, I have run a testing services company, RBCS. From humble beginnings, RBCS has grown into an international consulting, training, and outsourcing firm with clients on six continents. While I have remained a hands-on contributor to the firm, over 100 employees, subcontractors, and business partners have been the driving force of our ongoing success. I thank all of you for your hard work for our clients. Without the success of RBCS, I could hardly avail myself of the luxury of writing technical books, which is a source of great pride but not a whole lot of money. Again, I hope that our mutual successes together have repaid the debt of gratitude that I owe each of you.

Finally, I thank my family, especially my wife, Laurel, and my daughters, Emma and Charlotte. The hectic work schedule entailed in running a global testing services company means little time for my family, and my insatiable habit of writing books reduces that time further. To Laurel, Emma, and - Charlotte, know that I am aware that I can never fully repay the debt that I owe you for all that you give to me, but also know that my love for each of you is much greater than the time we get to share together.

ISTQB Copyright

This book is based on the ISTQB Advanced Syllabus version 2007. It also references the ISTQB Foundation Syllabus version 2007. It uses terminology definitions from the ISTQB Glossary version 2.0. These three documents are copyrighted by the ISTQB and used by permission.

Preface

I've worked as a software engineer for 25 years. When I started as a programmer, I learned to unit test my code as I wrote it. When I became a professional tester 20 years ago, I tested my own code and other people's code.

I received my degree in software engineering in the 1980s at the University of California, Los Angeles, which is a prestigious university. However, I learned very little about testing there. So when I became a professional tester, I found that I had to teach myself the skills, both from books and from experience.

In the 1980s, software testing was just emerging as a separate discipline within software engineering. It's no surprise that universities, even prestigious universities, would not provide much education on the topic of software testing at that time.

Unfortunately, software testing education at universities and colleges has not advanced much in the intervening 20 years. Training on software testing from private firms, including my own company, RBCS, has helped fill that gap. However, practitioners often obtain education in a piecemeal fashion, without continuity between subjects. Therefore, many practitioners of software testing remain self-taught, or educated in disparate concepts without a unifying set of ideas to help them understand that whole field. This has created a situation in which many practitioners do not understand the foundational and advanced concepts of software testing.

The International Software Testing Qualifications Board (ISTQB) is addressing this situation by defining a set of syllabi (or bodies of knowledge, if you prefer) that suggest what practitioners should know at various points in their careers as software testing professionals. I have served as president of the International Software Testing Qualifications Board and the American Software Testing Qualifications Board. I have also served on the ISTQB's Foundation Level Working Party and Advanced Level Working Party to help create these syllabi.

I previously wrote a book, with Isabel Evans, Dorothy Graham, and Erik van Veenendaal, called *Foundations of Software Testing*. That book addressed the Foundation Level syllabus.

This book addresses the portion of the Advanced Level syllabus targeted for test managers. Here, I will cover the concepts that practitioners should

apply as they reach an advanced point in their careers as software testing professionals. It is my hope that, after reading this book, you will feel more comfortable in your understanding and application of these concepts. In the next 20 years, before I retire from this field, I intend to help us move beyond a piecemeal, disparate, inchoate understanding of software testing to a point where software testing becomes a true, respected, professionally practiced specialty. I hope that this book will help you be among its respected practitioners.

Introduction

This is a book on advanced software testing for test managers. By that I mean that I address topics that a practitioner who has chosen to manage software testing as a career should know. I focus on those skills and techniques related to test analysis, test design, test execution, and test results evaluation. I assume that you know the basic concepts of test engineering, test design, test tools, testing in the software development lifecycle, and test management. You are ready to mature your level of understanding of these concepts and to apply these mature, advanced concepts to your daily work as a test professional.

This book follows the International Software Testing Qualifications Board's (ISTQB's) Advanced Level syllabus, with a focus on the material and learning objectives for the advanced test manager. As such, this book can help you prepare for ISTQB Advanced Test Manager exam. You can use this book to self-study for those exams or as part of an e-learning or instructor-led course on the topics covered in those exams. If you are taking an ISTQB-accredited Advanced Level Test Manager training course, this book is an ideal companion text for that course.

However, even if you are not interested in the ISTQB exams, you will find this book useful to prepare yourself for advanced work in software testing. If you are a test manager, test director, test analyst, technical test analyst, automated test engineer, manual test engineer, programmer, or in any other field where a sophisticated understanding of software test management is needed, then this book is for you.

This book focuses on test management. The book consists of 11 chapters, addressing the following material:

- 1. Basic aspects of software testing**
- 2. Testing processes**
- 3. Test management**
- 4. Test techniques**
- 5. Testing of software characteristics**
- 6. Reviews**

7. Incident (defect) management

8. Standards and test process improvement

9. Test tools and automation

10. People skills (team composition)

11. Preparing for the exam

Since that structure follows the structure of the ISTQB Advanced syllabus, some of the chapters address the material in great detail because they are central to the test manager role. Some of the chapters address the material in less detail because the test manager need only be familiar with it. For example, we cover test management in detail in this book because that is central to what a test manager does, while we spend less time on test techniques.

If you also read the companion volume to this book, which is for test analysts, you'll find parallel chapters that each address the material in detail, but with different emphasis. For example, test analysts need to know quite a bit about incident management. Test analysts spend a lot of time creating incident reports, and you need to know how to do that well. Test managers also need to know a lot about incident management, but they focus on how to keep incidents moving through their reporting and resolution lifecycle and how to gather metrics from such reports.

What should a test manager be able to do? Or, to ask the question another way, what should you have learned to do—or learned to do better—by the time you finish this book?

- Define the overall testing goals and strategy for the systems being tested
- Plan, schedule, and track the tasks
- Describe and organize the necessary activities
- Select, acquire, and assign the adequate resources to the tasks
- Select, organize, and lead testing teams
- Organize the communication between the members of the testing teams and between the testing teams and all the other stakeholders

- Justify the decisions and provide adequate reporting information where applicable

In this book, we focus on these main concepts. I suggest that you keep these high-level objectives in mind as we proceed through the material in each of the following chapters.

In writing this book and the companion volume on test analysis, I've kept foremost in my mind the question of how to make this material useful to you. If you are using this book to prepare for an ISTQB Advanced Level Test Manager exam, then I recommend that you read [chapter 11](#) first, then read the other 10 chapters in order. If you are using this book to expand your overall understanding of testing to an advanced level but do not intend to take an ISTQB Advanced Level Test Manager exam, then I recommend that you read [chapters 1 through 10](#) only. If you are using this book as a reference, then feel free to read only those chapters that are of specific interest to you.

Each of the first 10 chapters is divided into sections. For the most part, I have followed the organization of the ISTQB Advanced syllabus to the point of section divisions, but subsections and sub-subsection divisions in the syllabus might not appear. You'll also notice that each section starts with a text box describing the learning objectives for the section. If you are curious about how to interpret those K2, K3, and K4 tags in front of each learning objective, and how learning objectives work within the ISTQB syllabus, read [chapter 11](#).

Software testing is in many ways similar to playing the piano, cooking a meal, or driving a car. How so? In each case, you can read books about these activities, but until you have practiced, you know very little about how to do it. So, I've included practical, real-world exercises for the key concepts. I encourage you to practice these concepts with the exercises in the book. Then, make sure you take these concepts and apply them on your projects. You can become an advanced software test management professional only by managing software testing.

Chapter 1. Test Basics

"Oh, we don't drink while we test."

"Right, Larry, we drink because we test!"

—Two directors of large testing groups reacting to my story about a group of testers at an Internet start-up who did testing work while drinking beer.

The first chapter of the Advanced syllabus is concerned with contextual and background material that influences the remaining chapters. There are five sections:

- 1.** Introduction
- 2.** Testing in the Software Lifecycle
- 3.** Specific Systems
- 4.** Metrics and Measurement
- 5.** Ethics

Let's look at each section and how it relates to test management.

1.1. Introduction

Learning objectives

Recall of content only

This chapter, as the name implies, introduces some basic aspects of software testing. These central testing themes have general relevance for testing professionals.

ISTQB Glossary

software lifecycle: The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software lifecycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. Note that these phases may overlap or be performed iteratively.

There are four major areas:

- Lifecycles and their effects on testing
- Special types of systems and their effects on testing
- Metrics and measures for testing and quality
- Ethical issues

Many of these concepts are expanded upon in later chapters. While you might suspect this material and the material in the Foundation syllabus to be redundant, based on the name of this chapter, it actually expands on ideas introduced there.

1.2. Testing in the Software Lifecycle

Learning objectives

(K2) Describe how testing is a part of any software development and maintenance activity.

(K4) Analyze software lifecycle models and outline the most appropriate tasks and test activities to be executed, distinguishing between test and development activities.

Chapter 2 in the Foundation syllabus discusses integrating testing into the software lifecycle. As with the Foundation syllabus, at the Advanced level, you should understand that testing must be integrated into the software lifecycle to succeed. This statement is true whether the particular lifecycle chosen is sequential, incremental, iterative, or spiral.

Proper alignment between the testing process and other processes in the life-cycle is critical for success; this is especially true at key interfaces and hand-offs. The following list includes examples of these hand-offs:

- Requirements engineering and management
- Project management
- Configuration and change management
- Software development and maintenance
- Technical support
- Technical documentation

Let's look at two examples of alignment.

In a sequential lifecycle model, a key assumption is that the project team will define the requirements early in the project and then manage the (hopefully limited) changes to those requirements during the rest of the project. In such a situation, if the team follows a formal requirements process, an independent test team in charge of the system test level can follow an analytical requirements-based test strategy.

Using such a strategy in a sequential model, the test team would start

planning and designing tests early in the project, following an analysis of the requirements specification to identify test conditions. This planning, analysis, and design work might identify defects in the requirements, making testing a preventive activity. Failure detection would start much later in the lifecycle, once system test execution began.

However, suppose the project follows an incremental lifecycle model, adhering to one of the agile methodologies like Scrum. Now, the test team won't receive a complete set of requirements early in the project. Instead, the test team will receive requirements at the beginning of each 30-day "sprint".

Rather than analyzing requirements at the outset of the project, the best the test team can do is to identify and prioritize key quality risk areas; i.e., they can follow an analytical risk-based test strategy. Specific test designs and implementation will occur immediately before test execution, potentially reducing the preventive role of testing. Defect detection starts very early in the project, at the end of the first sprint, and continues in repetitive, short cycles throughout the project. In such a case, testing activities in the fundamental testing process overlap and are concurrent with each other as well as with major activities in the software lifecycle.

No matter what the lifecycle—and indeed, especially with the more fast-paced agile lifecycles—good change management and configuration management are critical for testing. A lack of proper change management results in an inability for the test team to keep up with what the system is and what it should do. A lack of proper configuration management, as was discussed in the Foundation syllabus, leads to loss of changes, an inability to say what was tested at what point in time, and severe lack of clarity around the meaning of the test results.

At this point, let's quickly review the major types of lifecycles mentioned in the syllabus: sequential, incremental, iterative, and spiral.

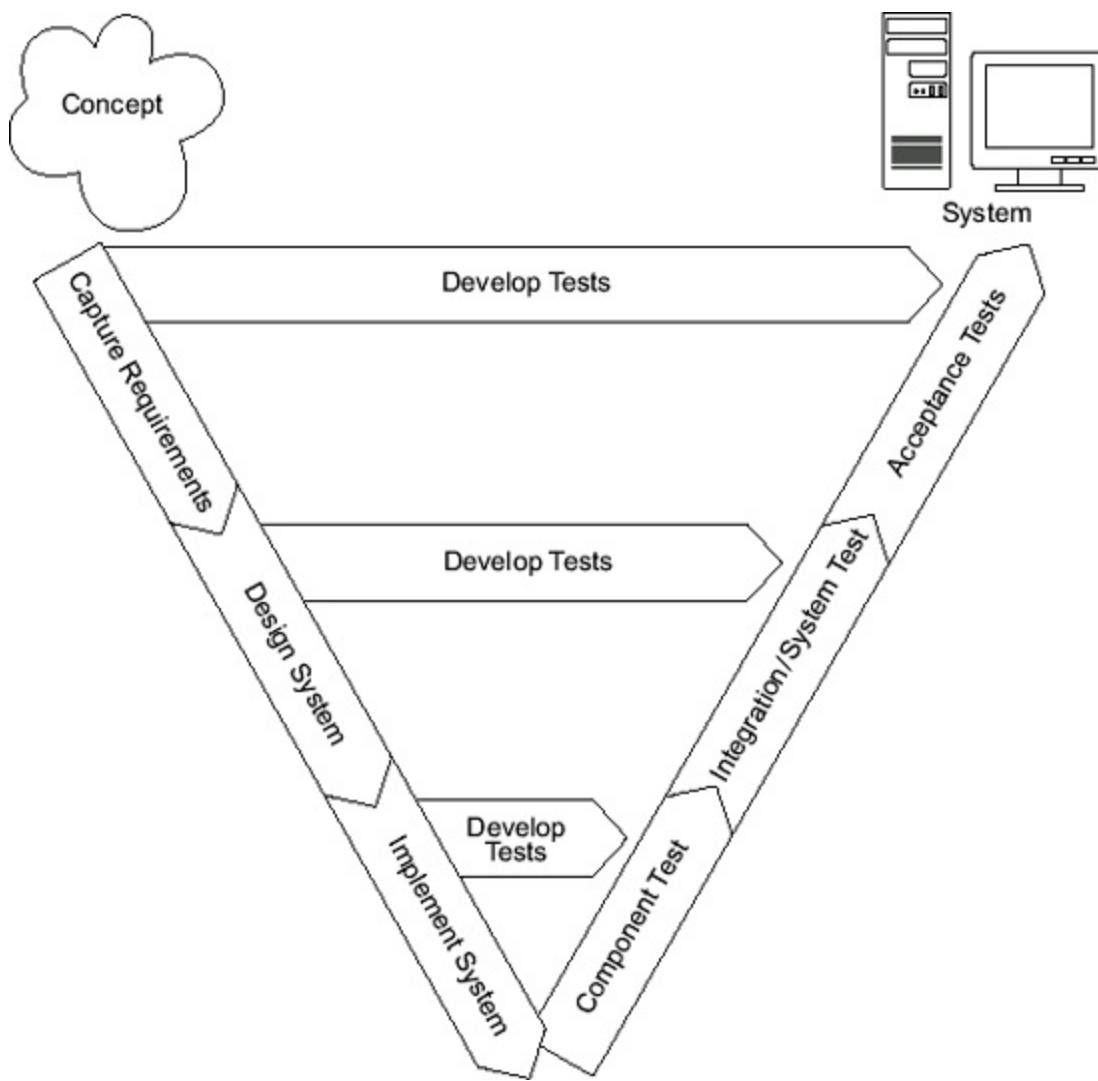
1.3. Sequential Lifecycle Models

Sequential models like the V-model shown in [figure 1-1](#) are called sequential because the entire system is built at one time, in a sequence of activities that successively define, then implement, then test. The two most common examples are the waterfall and the V-model.

You might have encountered the W-model too. The W-model depicts the test development activities for each test level—shown in the V-model as crossbars on the V—instead as a parallel track next to the requirements, design, and implementation activities. The W-model explicitly shows bug fixing associated with the test levels as a parallel track next to the testing activities. Some people find it clearer than the V-model, but I actually find diagrams of the W-model more confusing than the V-model because the W-model is more visually complicated.

The use of sequential lifecycle models creates certain issues for testing that the test manager must manage. The first and most infamous issue is that of schedule compression during testing at the end of the project. This "crunch" tends to set up an unpleasant trade-off between quality and delivery dates during the always-hectic end game of the project. When the test manager is put in the position of "quality cop"—that is, the person who is supposed to approve a release—a no-win situation occurs where the test manager is subjected to immense pressure to approve the release followed by howls of condemnation when the release proves bug-ridden in the field.

Figure 1-1. V-model



The second issue is the common problem of development groups, likewise pressured to achieve dates, delivering unstable and often untestable systems to the test team. This problem causes significant portions of the test schedule to be consumed by what is, effectively, retroactive unit testing. When compounded with a lack of adequate configuration management, this situation can render the high-level testing almost pointless, because what is delivered doesn't work and isn't even understood in terms of composition.

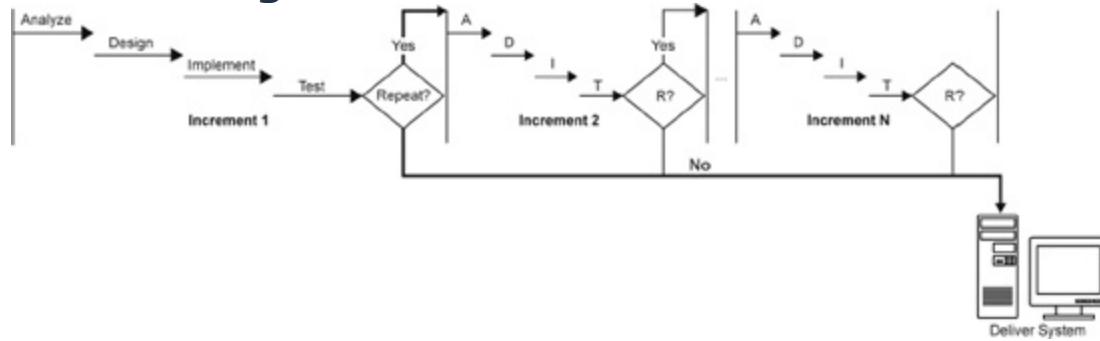
A third issue is the common failure to include the activities shown in the crossbars of the V-model in [figure 1-1](#); instead, due to other projects or a lack of management awareness, the test team is involved late. Very little preparation time is allowed. Testing typically devolves to an ad hoc or at best reactive strategy, with no defect prevention, no clear coverage, and limited value.

These sequential lifecycle issues are surmountable, but they require careful test management. You'll learn more about how to manage such issues in this book.

1.4. Iterative or Incremental Lifecycle Models

Iterative or incremental models are those where the system is built and tested iteratively in chunks, as shown in [figure 1-2](#). The grouping of functions and capabilities into chunks can be done based on risk, in that the functions and capabilities that are most likely to fail get built in the first chunk, then the next most likely to fail, and so forth. The grouping into chunks can be done based on customer priority so that the functions and capabilities most desirable to customers get built first, the least desirable last, and the others at some chunk in between. The grouping into chunks can also be influenced by regulatory requirements, design requirements, and other constraints.

Figure 1-2. Incremental model



There are myriad examples of these models, including evolutionary, incremental, and agile. There is tremendous variation in the size of the chunks, the duration of the iterations, and the level of formality. The common element is that fully integrated, working systems—albeit with only a portion of the functions and capabilities—are created earlier in the lifecycle than they would be in a sequential project.

The availability of testable systems earlier in the lifecycle would seem to be a benefit to the test manager, and it can be. However, the iterative lifecycle models create certain test issues for the test manager.

The first issue is the need, in each increment after the first one, to be able to regression test all the functions and capabilities provided in the previous increments. Because the most important functions and capabilities are typically provided in the earlier increments, you can imagine how important it is that these functions and capabilities not be broken. However, given the frequent and large changes to the code base—every increment being likely to introduce as much new and changed code as the previous increment—the risk of regression is high. This risk tends to lead to attempts to automate

regression tests, with varying degrees of success.

The second issue is the common failure to plan for bugs and how to handle them. This failure manifests itself when business analysts, designers, and programmers are assigned to work full-time on subsequent increments while testers are testing the current increment. In other words, you allow the activities associated with increments to overlap rather than requiring that each increment complete entirely before the next one starts; this can seem efficient at first. However, once the test team starts to locate bugs, an overbooked situation for the business analysts, designers, and programmers who must address them occurs.

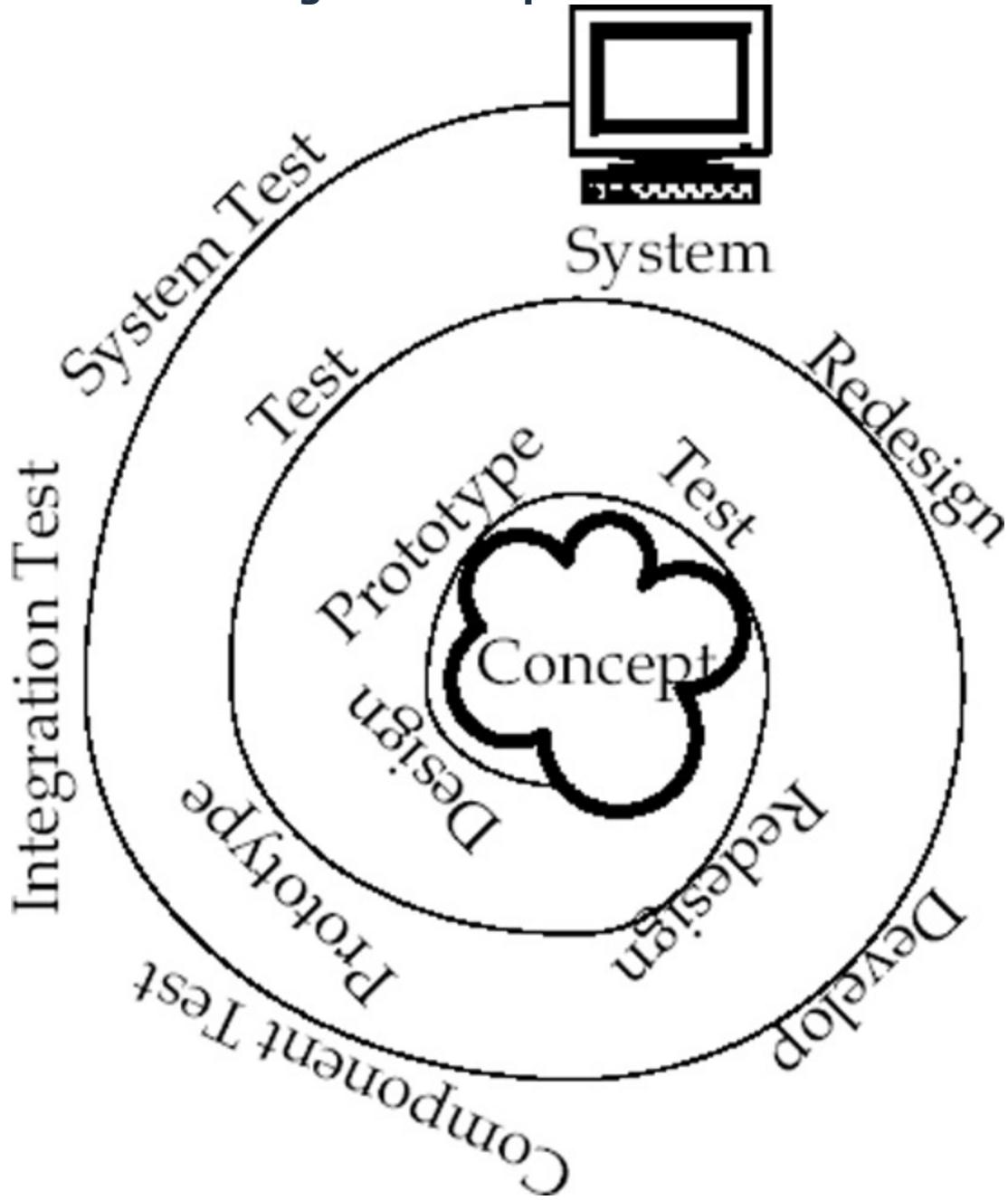
The final common issue, which is particularly common in the agile world, is the lack of rigor in and respect for testing. This issue is not universal—RBCS has clients that follow excellent practices in testing and use agile methodologies, particularly Scrum. These clients have found a way to integrate formal testing into agile lifecycles. However, some leading lights in the agile community are dismissive of formal testing and the infrastructure critical to it. For example, one proponent of agile methodologies has disparaged the formal tracking of bugs, which is an overwhelming proposition on any real-sized project.

Again, these are all surmountable issues, but the test manager must manage them carefully, in conjunction with the project management team. We'll address ways to manage these issues in this book.

1.5. Spiral Lifecycle Models

Finally, we have the spiral models, where early prototypes are used to design the system. The development work goes through a sequence of prototypes that are tested, then redesigned and reprototyped, and retested, until all of the risky design decisions have been proven (or disproven and rejected) through testing. The only example of this approach is the spiral model (figure 1-3), put forth by Victor Basili.

Figure 1-3. Spiral model



This model has seen service on the largest, most complex of projects, such as the United States missile defense system, which is under development. I have also used the spiral model myself on small projects to develop e-

learning packages with new technologies. It is quite useful when it is applied to projects with a large number of unknowns.

However, there are test issues created by the spiral model that the smart test manager must manage. The first issue is that, by the model's very nature, the designs of the system will change. Flexibility is paramount in all test case, test data, test tool, and test environment decisions early in the project. If the test manager commits too heavily to a particular way of generating test data, for example, and the structure of the system data repository changes dramatically, say from a relational database to XML files, serious rework of testing work products will occur.

The second issue is the unique, experimental mode of early testing. Testing of early prototypes is all about finding out what we don't know, subjecting questionable designs to rigorous testing to see what might not work. Confidence building is not an objective, typically. These different test objectives for earlier testing, evolving into a more typical role of testing in the final stages, require the test manager to change plans and strategies as the project progresses. Again, flexibility is key.

The final issue is that, given the spiral model's attempt to deal with unknowns through repeated prototyping, schedules can be quite unpredictable. Estimating and planning the testing work can be difficult, particularly if other projects are active at the same time.

Again, these are surmountable issues, but they are quite troublesome if not dealt with properly by the test manager.

1.6. Test Levels

The Foundation syllabus cited four typical test levels:

- Unit or component
- Integration
- System
- Acceptance

The Foundation syllabus mentioned some reasons for variation in these levels, especially with integration and acceptance.

Integration testing can mean component integration testing—integrating a set of components to form a system, testing the builds throughout that process. Or it can mean system integration testing —integrating a set of systems to form a system of systems, testing the system of systems as it emerges from the conglomeration of systems.

ISTQB Glossary

system of systems: Multiple heterogeneous, distributed systems that are embedded in networks at multiple levels and in multiple domains and are interconnected, addressing large-scale interdisciplinary common problems and purposes.

Acceptance test variations, discussed at the Foundation level, included user acceptance test and regulatory acceptance test.

Along with these four levels and their variants, at the Advanced level you need to keep in mind additional test levels that you might need for your projects. These levels would include the following:

- Hardware-software integration testing
- Feature interaction testing
- Customer product integration testing

You should expect to find most, if not all, of the following for each level:

- Clearly defined test goals and scope
- Traceability to the test basis (if available)
- Entry and exit criteria, as appropriate both for the level and for the system lifecycle
- Test deliverables, including results reporting that will be expected
- Test techniques that will be applied, as appropriate for the level, for the team, and for the risks inherent in the system
- Measurements and metrics
- Test tools, where applicable and as appropriate for the level
- And, if applicable, compliance with organizational or other standards

When RBCS performs assessments of test teams, we often find organizations use test levels but perform them in isolation—this is usually very inefficient and confusing. We discuss these topics in more detail in

chapter 3. You'll see then that using documents like test policies and coordinating closely with test-related staff can coordinate the test levels to reduce gaps, overlap, and confusion about results.

Let's take a closer look at this concept of alignment. We'll use the V-model ([figure 1-1](#)) as an example. We'll further assume that we are talking about the system test level.

In the V-model, with a well-aligned test process, test planning occurs concurrently with project planning. In other words, the moment of involvement of testing is at the very start of the project.

Once the test plan is approved, test control begins. Test control continues through to test closure. Analysis, design, implementation, execution, evaluation of exit criteria, and test results reporting are carried out according to the plan. Deviations from the plan are managed.

Test analysis starts immediately after or even concurrently with test planning. Test analysis and tests design happen concurrently with requirements, high-level design, and low-level design. Test implementation, including test environment implementation, starts during system design and completes just before test execution begins.

Test execution begins when the test entry criteria are met. More realistically, test execution starts when most entry criteria are met and any outstanding entry criteria are waived. In V-model theory, the entry criteria would include successful completion of both component test and integration test levels. Test execution continues until the test exit criteria are met, though again some of these will often be waived.

Evaluation of test exit criteria and reporting of test results occur throughout test execution. Test closure activities occur after test execution is declared complete.

This kind of precise alignment of test activities with each other and with the rest of the system lifecycle absolutely *will not* happen automatically. Nor can you expect to be able to instill this alignment continuously throughout the process, without any forethought.

Rather, for each test level, no matter what the selected software lifecycle and test process, the test manager must perform this alignment. Not only must this alignment happen during the test and project planning, but test control must include acting to ensure ongoing alignment.

No matter what test process and software lifecycle are chosen, each project has its own quirks. This is especially true for complex projects such as the systems of systems projects common in the military and among RBCS's larger clients. In such a case, the test manager must plan not only to align test processes, but also to modify them. Off-the-rack process models, whether for testing alone or for the entire software lifecycle, don't fit such complex projects well.

1.6.1. Two Case Studies of Integrating Testing into the Lifecycle

Two important concepts in the ISTQB approach are the ideas of testing integrated throughout the lifecycle and testing as a preventive activity. Let's look at an example of both, applied to an Internet appliance project. On this project, performance and scalability were potential major risks. If the servers would not scale to handle the load of thousands of Internet appliances, a number of financial and customer-satisfaction disasters would ensue.

So, on this project, we had four major activities to assure performance and scalability of the delivered system:

- During the design phase, we did static performance tests of proposed designs, which basically consisted of reviews of spreadsheet models of load and performance.
- Once those tests were done—but still during the design phase—we did static performance analysis, using commercial simulation tools, of the proposed designs.
- As programming progressed, programmers were responsible for doing unit performance tests as they implemented the design. The harnesses used for these tests were also used to build system performance test tools.
- Finally, once we had a feature-complete system, we started system test. (We followed a sequential lifecycle on this project.) At this point, we did system performance tests of the implemented design.

This approach reduced the number of serious performance problems found during system test to a manageable number. That's not to say we didn't find performance defects—we found over a dozen—but they were all of the sort that could be fixed quickly and without major changes to the design, or the

schedule.

Let's look at an example of how development and test tasks interact in the lifecycle. This example is from a large, complex project to develop a system of systems providing an entertainment network in North America. [figure 1-4](#) shows the entry criteria for the System Integration Test phase, the last of the formal test phases before delivery of the system for operation.

Figure 1-4. Example of entry criteria

1. Pre-Integration Test has exited.
2. Final, approved versions of the primary specification documents exist and have been provided to the individuals responsible for performing System Integration Testing.
3. This document [System IntegrationTest Plan] and the System IntegrationTestSuite documents are both final and approved.
4. The Network Team configures the [entire] live system for testing... In addition, all integrated components, hardware and software, must be of the correct major and minor releases...
5. All Development Teams provide revision-controlled, complete systems for Integration Test, with these products having completed at least one cycle of the applicable System Test.
6. All Development Teams provide all custom documentation needed [for] the software mentioned in entry criteria 5.
7. All Development Teams have plans in place to resolve all known "must-fix" bugs.

- Entry criterion 1 requires that the preintegration test phase has exited. This was an informal test phase design to check, on a risk-driven basis, whether various intersystem interfaces were working properly.
- Entry criterion 2 requires that we have final, approved versions of the primary specification documents. This has a lifecycle assumption embedded in it, which is that we are following a lifecycle that would produce such documents.
- Entry criterion 3 requires approval of the system integration test plan and the system integration tests. The development and project management teams were to review these documents and provide approval, which was to align development and test activities.
- Entry criterion 4 requires that the live system be set up for testing. The lifecycle assumption here is that we would have access to the live system and that it would be ready well in advance of going into production so that we could use it for testing. Obviously, this is a

unique criterion, one that only people working on deploying completely new systems would be able to have.

- Entry criterion 5 requires that the development teams provide us with revision-controlled, complete systems, which imposes a level of formalism on configuration management. And, we further require that these systems have completed at least one cycle of the applicable system test. That's not to say that the system tests necessarily all pass, but at least enough of the system tests have been run that we're satisfied that the systems will provide all major functions.
- Entry criterion 6 requires that not only do we get systems, and tested systems, but also that we get whatever documentation is needed.
- Finally, entry criterion 7 requires that there be a plan to resolve all known "must-fix" bugs. This criterion imposes a requirement on development and the lifecycle that we not enter integration testing with no end in sight.

As you can see, these entry criteria provide a tight linkage between Integration Testing, previous testing activities, and the overall development lifecycle.

1.6.2. Testing in the Software Lifecycle Exercise

Read the HELLOCARMS Systems Requirements Document. Note that some of the sections are still to be completed, with the notation [TBD], meaning to be determined.

Assume that the project will follow an iterative lifecycle model. There will be five iterations. The features and capabilities built in each iteration are based on the priority of the underlying requirement.

Following the ISTQB fundamental test process, outline the test activities required for this project, including each iteration.

An overview of one possible solution is shown in the following debrief, though many solutions are possible.

1.6.3. Testing in the Software Lifecycle Debrief

First, because we're following the ISTQB fundamental test process, let's refresh our memory on that process:

1. Test Planning and Control
2. Test Analysis and Design
3. Test Implementation and Execution
4. Evaluating Exit Criteria and Reporting
5. Test Closure Activities

So, there are activities in each of these five areas. However, some iterations might not include all areas.

Your solution could be structured like this:

1. Initial Project Planning Period
 - 1.1. Test Planning and Control
 - 1.1.1. Create system integration test plan
 - 1.1.2. Participate in overall project planning
 - 1.2. Test Analysis and Design
 - 1.2.1. Perform quality risk analysis
 - 1.2.2. Outline test suites needed in each iteration
- Iteration One (Very High Priority Features)
 - 2.1. Test Planning and Control
 - 2.1.1. Adjust plan as needed for this iteration
 - 2.1.2. Guide testing work during this iteration
 - 2.2. Test Analysis and Design
 - 2.2.1. Adjust quality risk analysis
 - 2.2.2. Design test suites and cases for this iteration
 - 2.3. Test Implementation and Execution
 - 2.3.1. Implement test suites for this iteration

2.3.2. Execute test suites for this iteration

2.4. Evaluating Exit Criteria and Reporting

2.4.1. Check test results against exit criteria in test plan

2.4.2. Report test results to project management team

Iteration Two (High Priority Features)

3.1. Test Planning and Control

3.1.1. Adjust plan as needed for this iteration

3.1.2. Guide testing work during this iteration

3.2. Test Analysis and Design

3.2.1. Adjust quality risk analysis

3.2.2. Design test suites and cases for this iteration

Test Implementation and Execution

4.1.1. Implement test suites for this iteration

4.1.2. Execute test suites for this iteration

2. Evaluating Exit Criteria and Reporting

5.1.1. Check test results against exit criteria in test plan

5.1.2. Report test results to project management team

3. Iteration Three (Medium Priority Features)

6.1. Test Planning and Control

6.1.1. Adjust plan as needed for this iteration

6.1.2. Guide testing work during this iteration

6.2. Test Analysis and Design

6.2.1. Adjust quality risk analysis

6.2.2. Design test suites and cases for this iteration

6.3. Test Implementation and Execution

6.3.1. Implement test suites for this iteration

6.3.2. Execute test suites for this iteration

6.4. Evaluating Exit Criteria and Reporting

6.4.1. Check test results against exit criteria in test plan

6.4.2. Report test results to project management team

4. Iteration Four (Low Priority Features)

7.1. Test Planning and Control

7.1.1. Adjust plan as needed for this iteration

7.1.2. Guide testing work during this iteration

7.2. Test Analysis and Design

7.2.1. Adjust quality risk analysis

7.2.2. Design test suites and cases for this iteration

7.3. Test Implementation and Execution

7.3.1. Implement test suites for this iteration

7.3.2. Execute test suites for this iteration

7.4. Evaluating Exit Criteria and Reporting

7.4.1. Check test results against exit criteria in test plan

7.4.2. Report test results to project management team

5. Iteration Five (Very Low Priority Features)

8.1. Test Planning and Control

8.1.1. Adjust plan as needed for this iteration

8.1.2. Guide testing work during this iteration

8.2. Test Analysis and Design

8.2.1. Adjust quality risk analysis

8.2.2. Design test suites and cases for this iteration

8.3. Test Implementation and Execution

8.3.1. Implement test suites for this iteration

8.3.2. Execute test suites for this iteration

8.4. Evaluating Exit Criteria and Reporting

8.4.1. Check test results against exit criteria in test plan

8.4.2. Report test results to project management team

6. Post Project Period

9.1. Test Planning and Control

9.1.1. Document variances from plan

9.1.2. Participate in project retrospective

9.2. Test Closure Activities

9.2.1. Finalize testware for archiving or hand-off

9.2.2. Document test environment configuration

You should include one more level of detail below this, showing the particular test environments, test suites, and test data needed for each iteration.

1.7. Specific Systems

Learning objectives

(K2) Explain by giving examples of the specifics of testing systems of systems.

(K2) Explain why the three main outcomes of testing safety-critical systems are required to demonstrate compliance to regulations.

Systems of systems are independent systems tied together to serve a common purpose. Since they are independent and tied together, they often lack a single, coherent user or operator interface, a unified data model, compatible external interfaces, and so forth.

Such projects might include the following characteristics and risks:

- The integration of commercial off-the-shelf (COTS) software, along with some amount of custom development, often taking place over a long period.
- Significant technical, lifecycle, and organizational complexity and heterogeneity. This organizational and lifecycle complexity can include issues of confidentiality, company secrets, and regulations.
- Different development lifecycles and other process among disparate teams, especially—as is frequently the case—when distributed work, insourcing, and outsourcing are involved.
- Serious potential reliability issues due to intersystem coupling, where one inherently weaker system creates ripple-effect failures across the entire system of systems.
- System integration testing, including interoperability testing, is essential. Well-defined interfaces for testing are needed.

At the risk of restating the obvious, systems of systems projects are more complex than single-system projects. This complexity increase applies organizationally, technically, process-wise, and team-wise. Good project management, formal development lifecycles and processes, configuration management, and quality assurance become more important as size and complexity increases.

Let's focus for a moment on the lifecycle implications of systems of systems projects. As mentioned before, with systems of systems projects, we are typically going to have multiple levels of integration. First, we will have component integration for each system, and then we'll have system integration as we build the system of systems.

We will also typically have multiple version management and version control systems and processes, unless all the systems happen to be built by the same (presumably large) organization and that organization follows the same approach throughout its software development team. This setup is not something my associates and I commonly see during assessments of large companies, by the way.

Projects tend to have a long duration. I have seen them planned for as long as five to seven years. A system of systems project that involves integrating five or six systems might be considered relatively short and relatively small if it lasted "only" a year and involved "only" 40 or 50 people. Across this project, there are multiple test levels, usually owned by different parties.

Because of the size and complexity of the project, it's easy for hand-offs and transfers of responsibility to break down. So we need formal information transfers among project members, especially at milestones.

Even when we're integrating purely off-the-shelf systems, these systems are evolving—that's all the more likely to be true with custom systems. We have the test management challenge of coordinating development of the individual systems and the test analyst challenge of proper regression tests at the system of systems level when things change.

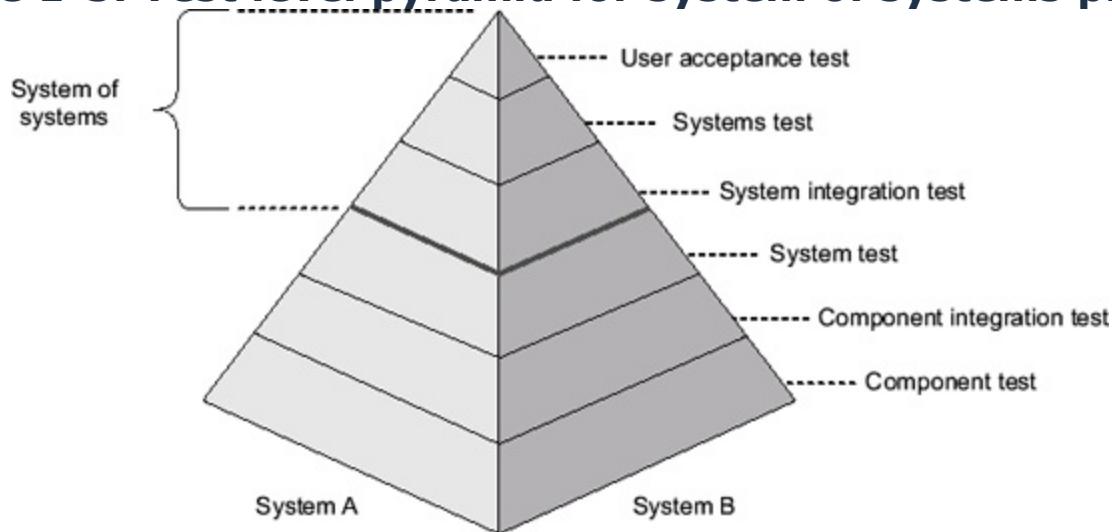
Especially with off-the-shelf systems, maintenance testing can be triggered—sometimes without much warning—by external entities and events such as obsolescence, bankruptcy, or upgrade of an individual system.

If you think of the fundamental test process, in a system of systems project, the progress of levels is not two-dimensional. Instead, imagine a sort of pyramidal structure, as shown in [figure 1-5](#).

At the base, we have component testing. A separate component test level exists for each system. Moving up the pyramid, we have component integration testing. A separate component integration test level exists for each system. Next, we have system testing. A separate system test level exists for each system.

Note that, for each of these test levels, we have separate organizational ownership, if the systems come from different vendors. We also probably have separate team ownership, since multiple groups often handle component, integration, and system test.

Figure 1-5. Test level pyramid for system of systems project



Continuing to move up the pyramid, we come to system integration testing. Now, finally, we are talking about a single test level across all systems. Next above that we find systems testing, focusing on end-to-end tests that span all the systems. Finally, we have user acceptance testing. For each of these test levels, while we have single organizational ownership, we probably have separate team ownership.

1.7.1. Case Study of a System of Systems Project

Table 1-1 shows an example of what we called "preintegration testing" on a complex system of systems project to deliver an entertainment system in North America. The system of systems consisted of seven main systems, labeled A through G in this table:

- The interactive voice response server hardware itself, a custom server
- The local and wide area networks tying the systems together
- The interactive voice response server application, being the software running on the IVR servers that actually interacted with customers
- The customer service application, which ran in the call center, allowing agents to handle calls from customers, particularly if they had trouble

using the system

- The call center infrastructure, such as the computer-telephony integration suites
- The data warehouse that stored transaction information on the customers, along with other business intelligence information
- The content management software that managed all the content on the servers

Table 1-1. Backbone integration in a system of systems

System	BB0	BB1	BB2	BB3	BB4	BB5
A. IVR HW	Alpha		Beta	Quad		Prod
B. Network	LAN					WAN
C. IVR App	C1-E1	C2-D1	C3-D2 C3-G3	C4-D3 C5-F1 C6-E4	C7-F2 C8-D6	C9-D8
D. Customer Service App		D1-C2	D2-C3	D3-C4 D4-G4 D5-E5	D6-C8 D7-F3	D8-C9 D9-F5
E. Call Center	E1-C1 E1-G1	E2-G2	E3-G4	E4-C6 E5-D5		
F. Data Warehouse				F1-C5 F3-D7 F4-G5	F2-C7	F5-D9
G. Content Mgmt	G1-E1	G2-E2	G3-C3 G4-E3	G4-D4	G5-F4	

Preintegration testing was the name we gave to the phase of testing that

preceded system integration testing. In this phase, we identified six main builds, which we called backbones. Those are identified in the BB labels across the top, from BB0 to BB5. Each build allowed testing of one or more interfaces between the systems. The interfaces are identified in the table with a letter/number dash letter/number combination. The letter corresponds to the system; the number is a sequence number for the interface. Notice that some interfaces are used by one system to interact with more than one other system. For example, the call center's E1 interface interacts with the IVR application and the content management system in backbone zero.

ISTQB Glossary

safety-critical system: A system whose failure or malfunction may result in death or serious injury to people, or loss or severe damage to equipment, or environmental harm.

The objective of this exercise, as you probably have guessed, was to test as many critical interfaces as possible before formal system integration testing. This early testing would allow us to find dangerous—and potentially design-threatening—defects before formal integration testing started so we could eliminate them.

1.7.2. Safety-Critical Systems

Simply put, safety-critical systems are those systems upon which lives depend. Failure of such a system—or even temporary performance or reliability degradation or undesirable side effects as support actions are carried out—can injure or kill people.

Safety-critical systems, like systems of systems, have certain associated characteristics and risks, including the following:

- Because defects can cause death, and deaths can cause civil and criminal penalties, so proof of adequate testing can be and often is used to reduce liability.
- For obvious reasons, various regulations and standards often apply to safety-critical systems. The regulations and standards can constrain the process, the organizational structure, and the product. Unlike the usual constraints on a project, though, these are constructed specifically to increase the level of quality rather than to enable trade-offs to enhance schedule, budget, or feature outcomes at the expense of quality. Overall, there is a focus on quality as a very important project priority.
- There is typically a rigorous approach to both development and testing. Throughout the lifecycle, traceability extends all the way from regulatory requirements to test results, and helps demonstrate compliance. It requires extensive, detailed documentation but provides

high levels of auditability, even by non-test experts.

Audits are not unusual if regulations are imposed. Demonstrating compliance can involve tracing from the regulatory requirement through development to the test results. An outside party typically performs the audits. So, the traceability gathering up-front and the audit/compliance activities at the back end affect management, development, testing, and the competent authorities, from both a people and a process point of view.

During the lifecycle—often as early as design—the project team uses safety analysis techniques to identify potential problems. Single points of failure are often resolved through system redundancy.

In some cases, safety-critical systems are complex systems or even systems of systems. In other cases, non-safety-critical components or systems are integrated into safety-critical systems or systems of systems. For example, networking or communication equipment is not inherently a safety-critical system, but if integrated into an emergency dispatch or military system, it becomes part of a safety-critical system.

Formal quality risk management is essential in these situations. Fortunately, a number of such techniques exist, such as: failure mode and effect analysis; failure mode, effect, and criticality analysis; hazard analysis; and software common cause failure analysis. Let's look at an example.

One of RBCS's clients builds regulated medical devices. The United States Food and Drug Administration rules apply to this organization and its products. These regulations have the following implications for testing:

- Every one of its test cases is traceable to the requirements specification. The test case is the way in which the organization can prove that a requirement is met.
- Periodically, there is an audit of the organization to ensure compliance. This audit includes tests and test results.
- In part to support this audit, the organization stores not only tests but also evidence of test results, including screen shots.

ISTQB Glossary

metric: A measurement scale and the method used for measurement.

measurement scale: A scale that constrains the type of data analysis that can be performed on it.

measurement: The process of assigning a number or category to an entity to describe an attribute of that entity.

measure: The number or category assigned to an attribute of an entity by making a measurement.

While this sounds quite formal, this organization uses the most lightweight possible documentation. The organization follows the Scrum lifecycle, an agile lifecycle model. However, the organization has modified the lifecycle to support the documentation required.

1.8. Metrics and Measurement

Learning objectives

(K2) Describe and compare the standard testing-related metrics

(K3) Monitor testing activities by measuring the test object(s) and the test process.

Throughout this book, we use metrics and measurement to establish expectations and guide testing by those expectations. You can and should apply metrics and measurements throughout the software development lifecycle because well-established metrics and measures, aligned with project goals and objectives, will enable test analysts to report and track test and quality results to management in a consistent and coherent way.

A lack of metrics and measurements leads to purely subjective assessments of quality and testing, and to disputes over the meaning of test results toward the end of the lifecycle. It also results in a lack of clearly perceived and communicated value, effectiveness, and efficiency for testing.

Not only must we have metrics and measurements, we also need baselines. What is a "good" result for a given metric? An acceptable result? An unacceptable result? Without defined baselines, successful testing is usually impossible. In fact, when my associates and I perform assessments for our clients, we often find ill-defined (or nonexistent) metrics of test team effectiveness and efficiency, with no baselines and thus bad and unrealistic expectations (which of course aren't met).

There's just about no end to what can be subjected to a metric and tracked through measurement. Consider the following:

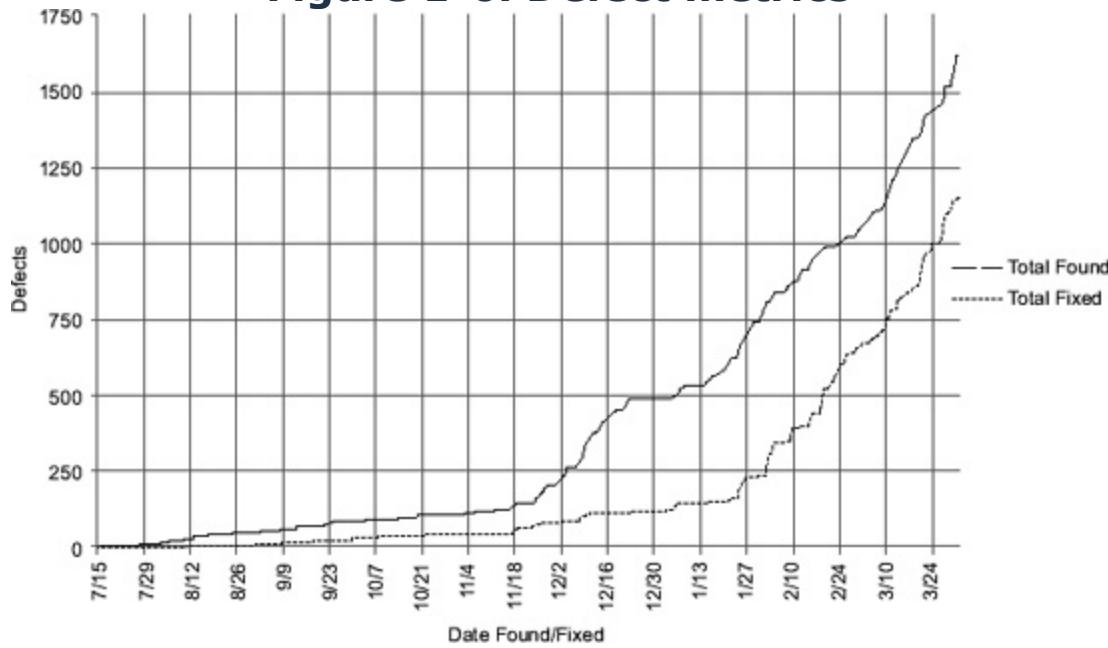
- Planned schedule and coverage
- Requirements and their schedule, resource, and task implications for testing
- Workload and resource usage
- Milestones and scope of testing
- Planned and actual costs

- Risks, both quality and project risks
- Defects, including total found, total fixed, current backlog, average closure periods, and configuration, subsystem, priority, or severity distribution

During test planning, we establish expectations, which I mentioned as the baselines previously. As part of test control, we can measure actual outcomes and trends against these expectations and adjust our approach as indicated. As part of test reporting, we can consistently explain to management various important aspects of the process, product, and project, using objective, agreed-upon metrics with realistic, achievable targets.

Let's look at an example. [figure 1-6](#) shows the total number of defects found and fixed on the system of systems project I've used as an example a few times in this book. As you can see, this project is in a lot of trouble at this point. There are a very large number of total defects, there is a significant backlog of defects, and the weekly defect discovery rate remains very high. The only good news, apparent in this graph, is that the weekly defect fix rate is about the same as the discovery rate. So, while the discovery rate isn't declining, at least the backlog is not growing.

Figure 1-6. Defect metrics



When you're thinking about a testing metrics and measurement program, there are three main areas to consider: definition, tracking, and reporting. Let's start with definition.

In a successful testing metrics program, you define a useful, pertinent, and concise set of quality and test metrics for a project. You avoid too large a set of metrics, as this will prove difficult and perhaps expensive to measure while often confusing rather than enlightening the viewers and stakeholders.

You also want to ensure uniform, agreed-upon interpretations of these metrics, to minimize disputes and divergent opinions about measures of outcomes, analyses, and trends. There's no point in having a metrics program if everyone has an utterly divergent opinion about what particular measures mean.

Finally, define metrics in terms of objectives and goals for a process or task, for components or systems, and for individuals or teams.

Victor Basili's well-known Goal Question Metric technique is one way to evolve meaningful metrics. In this technique, we proceed from the goals of the effort—in this case, testing—to the questions we'd have to answer to know if we were ultimately meeting those goals to the specific metrics.

For example, one typical goal of testing is to build confidence. One natural question that arises in this regard is, "How much of the system has been tested?" Metrics for coverage include the percentage of requirements covered by tests, branches and statements covered by tests, interfaces covered by tests, and so forth.

Let's move on to tracking. Because tracking is a recurring activity in a metrics program, the use of automated tool support can reduce the time required to capture, track, analyze, report, and measure the metrics.

Be sure to apply objective and subjective analyses for specific metrics over time, especially when trends emerge that could allow for multiple interpretations of meaning. Try to avoid jumping to conclusions, or delivering metrics that encourage others to do so.

Be aware of and manage the tendency for people's interests to affect the interpretation they place on a particular metric or measure. Everyone likes to think they are objective—and, of course, right as well as fair!—but usually people's interests affect their conclusions.

Finally, let's look at reporting. Most importantly, reporting of metrics and measures should enlighten management and other stakeholders, not confuse or misdirect them. In part, this is achieved through smart definition

of metrics and careful tracking, but it is possible to take perfectly clear and meaningful metrics and confuse people with them through bad presentation. Edward Tufte's series of books is a treasure trove of ideas about how to develop good charts and graphs for reporting purposes.^[1]

[1] Tufte's series of books includes *The Graphical Display of Quantitative Information*, *Envisioning Information*, and *Visual Explanations*.

Tufte would surely agree that good testing reports based on metrics should be easily understood, not overly complex and certainly not ambiguous. They should also draw the viewer's attention toward what matters most, not toward trivialities. In that way, good testing reports based on metrics and measures will help management guide the project to success.

Not all types of graphical displays of metrics are equal—or equally useful. A snapshot of data at a moment in time, as shown in a table, might be the right way to present some information, such as the coverage planned and achieved against certain critical quality risk areas. A graph of a trend over time might be a useful way to present other information, such as the total number of defects reported and the total number of defects resolved since the start of testing. An analysis of causes or relationships might be a useful way to present other information, such as a scatter plot showing the correlation (or lack thereof) between years of tester experience and percentage of bug reports rejected.

1.8.1. Metrics and Measurement Exercise

Assume you are using a risk-based testing strategy for the HELLOCARMS project. Describe one or more metrics that you could use to report on the remaining level of quality risk during test execution. Some of the possible answers are listed on the next page.

1.8.2. Metrics and Measurement Debrief

There are a number of ways to report on the remaining level of quality risk during test execution:

- A table that lists the major risk items, along with the number of test cases planned, the number of test cases passed, and the number of test cases failed for each risk item
- The same table, but with two added numbers, the number of defects reported against the risk item and the number of those defects that are

now closed

- A histogram graph showing the planned versus actual coverage of risk items by test cases
- The same histogram graph, with an additional set of bars representing the percentage of defects found for each risk item

Notice that you need traceability from risk items to test cases and/or defect reports for many of these metrics to work.

ISTQB Glossary

ethics: *Not defined in the ISTQB glossary.*

1.9. Ethics

Learning objectives

Recall of content only

Many professions have ethical standards. In the context of professionalism, ethics are "rules of conduct recognized in respect to a particular class of human actions or a particular group, culture, etc."^[1]

[1] Definition from [dictionary.com](https://www.dictionary.com).

Because testers often have access to confidential and privileged information, ethical guidelines can help them use that information appropriately. In addition, testers should use ethical guidelines to choose the best possible behaviors and outcomes for a given situation, given their constraints. "Best possible" means for everyone, not just the tester.

Let me give you an example of ethics in action. I am an officer of three related international software testing consultancies: RBCS, RBCS NZ, and PureTesting. I also serve on the ISTQB and ASTQB boards of directors. As such, I might and do have insight into the direction of the program that our competitors in the software testing consultancy business don't have.

In some cases, such as helping to develop syllabi, I have to make those business interests clear to people, but I am allowed to help. I helped write both the Foundation and Advanced syllabi.

In other cases, such as developing exams, I agreed, along with my colleagues on the ASTQB, that I should not participate. Direct access to the exams would make it all too likely that, consciously or unconsciously, I would warp our training materials to "teach the exam."

As you advance in your career as a tester, more and more opportunities to show your ethical nature—or to be betrayed by a lack of it—will come your way. It's never too early to inculcate a strong sense of ethics.

At the Advanced level, the ISTQB expects certificate-holders to adhere to the following code of ethics:

- PUBLIC-Certified software testers shall act consistently with the public interest. For example, if you are working on a safety-critical system and

are asked to quietly cancel some defect reports, that's an ethical problem.

- **CLIENT AND EMPLOYER**-Certified software testers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest. For example, if you know that your employer's major project is in trouble, and you short-sell the stock and then leak information about the project problems to the Internet, that's a real ethical lapse—and probably a criminal one, too.
- **PRODUCT**-Certified software testers shall ensure that the deliverables they provide (on the products and systems they test) meet the highest professional standards possible. For example, if you are working as a consultant and you leave out important details from a test plan so that the client has to hire you on the next project, that's an ethical lapse.
- **JUDGMENT**-Certified software testers shall maintain integrity and independence in their professional judgment. For example, if a project manager asks you not to report defects in certain areas due to potential business sponsor reactions, that's a blow to your independence and an ethical failure on your part if you comply.
- **MANAGEMENT**-Certified software test managers and leaders shall subscribe to and promote an ethical approach to the management of software testing. For example, favoring one tester over another because you would like to establish a romantic relationship with the favored tester's sister is a serious lapse of managerial ethics.
- **PROFESSION**-Certified software testers shall advance the integrity and reputation of the profession consistent with the public interest. For example, if you have a chance to explain to your child's classmates or your spouse's colleagues what you do, be proud of it and explain the ways software testing benefits society.
- **COLLEAGUES**-Certified software testers shall be fair to and supportive of their colleagues, and promote cooperation with software developers. For example, it is unethical to manipulate test results to arrange the firing of a programmer whom you detest.
- **SELF**-Certified software testers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession. For example, attending courses, reading books, and speaking at conferences about what you do

help to advance you—and the profession. Participation like this is called doing well while doing good, and fortunately, it is very ethical!

1.10. Sample Exam Questions

To end each chapter, you can try one or more sample exam questions to reinforce your knowledge and understanding of the material and to prepare for the ISTQB Advanced Level Test Manager exam.

Q: Which of the following is an example of testing as part of the requirements specification phase of a project?

- 1.** A requirements review meeting
- 2.** A business analyst eliciting requirements
- 3.** A database administrator designing a table
- 4.** A test results report showing requirements coverage

Q: Assume you are a test manager working on a project to create a programmable thermostat for home use to control central heating, ventilation, and air conditioning (HVAC) systems. This project is following a sequential lifecycle model, specifically the V-model. Currently, the system architects have released a first draft design specification based on the approved requirements specification released previously.

Identify all of the following that are appropriate test tasks to execute at this time:

- 1.** Design tests from the requirements specification
- 2.** Analyze design-related risks
- 3.** Execute unit test cases
- 4.** Write the test summary report
- 5.** Design tests from the design specification

Q: You are the manager of a bank's quality assessment group, in charge of independent testing for banking applications. You are working on a project to implement an integrated system that will use three off-the-shelf systems to manage a bank's accounts-receivable system.

Identify all of the following that are test levels that you would expect to *directly* manage:

- 1.** Component testing for each system
- 2.** Component integration testing for each system
- 3.** System testing for each system
- 4.** Contract acceptance testing for each system
- 5.** System integration testing

Q: Which of the following is generally applicable to demonstrating compliance to regulations for safety critical systems?

- 1.** Test traceability
- 2.** ISO 61508
- 3.** Usability testing
- 4.** Employee evaluations

Q: 5. Which of the following is an accurate statement that captures a difference between a trend chart showing the total number of bugs discovered and resolved during test execution and a trend chart showing the total number of test cases passed and failed during test execution?

- 1.** The bug trend chart will reveal test progress problems while the test case trend chart will not.
- 2.** The bug trend chart will show test coverage while the test case trend chart will not.
- 3.** The test case trend chart will reveal test progress problems while the bug trend chart will not.
- 4.** The test case trend chart will show test coverage while the bug trend chart will not.

Q: Considering the typical objectives of testing that are identified in the

Foundation syllabus, which of the following metrics can we use to measure the effectiveness of the test process in achieving one of those objectives?

- 1.** Average days from defect discovery to resolution
- 2.** Lines of code written per developer per day
- 3.** Percentage of test effort spent on regression testing
- 4.** Percentage of requirements coverage

Chapter 2. Testing Processes

Drill Sergeant: Gump! Why did you put that weapon together so quickly, Gump?

Forrest Gump: You told me to, Drill Sergeant?

Drill Sergeant: [Expressing surprise and looking at a stopwatch.] This is a new company record! If it wouldn't be such a waste of a fine enlisted man, I'd recommend you for OCS! You are gonna be a general some day, Gump. Now disassemble your weapon and continue!

—Forrest Gump displays an innate ability to follow a process accurately and quickly in a scene the movie *Forrest Gump*, screenplay by Eric Roth, from the novel by Winston Groom.

The second chapter of the Advanced syllabus is concerned with the process of testing and the activities that occur within that process. It establishes a framework for all the subsequent material in the syllabus and allows you to visualize organizing principles for the rest of the concepts. There are seven sections:

- 1.** Introduction
- 2.** Test Process Models
- 3.** Test Planning and Control
- 4.** Test Analysis and Design
- 5.** Test Implementation and Execution
- 6.** Evaluating Exit Criteria and Reporting
- 7.** Test Closure Activities

Let's look at each section and how it relates to test management.

2.1. Introduction

Learning objectives

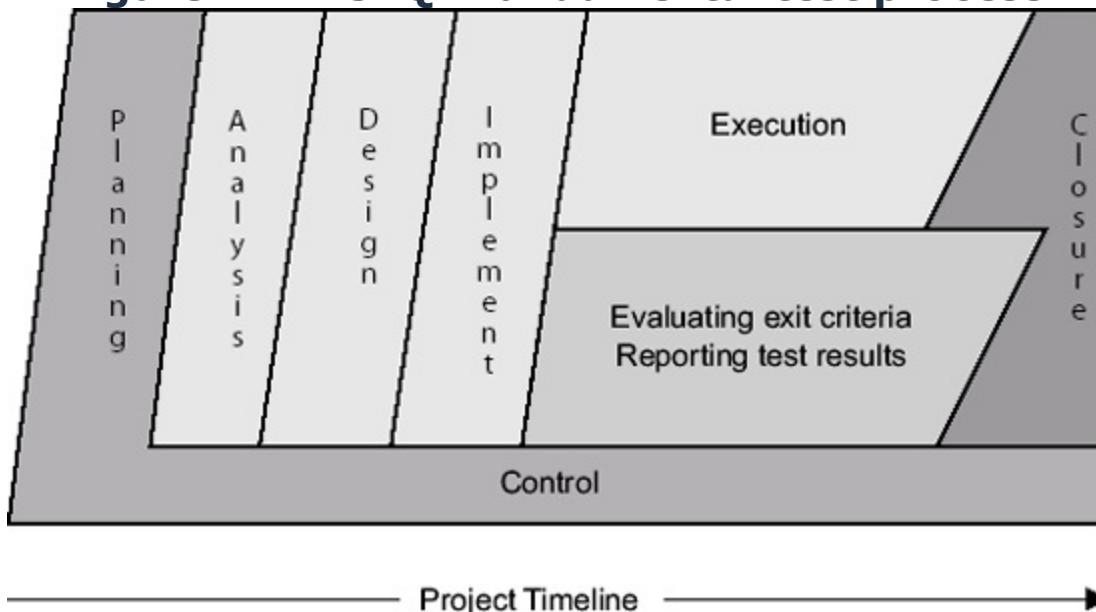
Recall of content only

The ISTQB Foundation syllabus describes the ISTQB fundamental test process. It provides a generic, customizable test process, shown in [figure 2-1](#). That process consists of the following activities:

- Planning and control
- Analysis and design
- Implementation and execution
- Evaluating exit criteria and reporting
- Test closure activities

As test managers, we don't need to spend much time on analysis and design, but we do need to look at the other activities, especially planning and control.

Figure 2-1. ISTQB fundamental test process



2.2. Test Process Models

Learning objectives

Recall of content only

The concepts in this section are covered in more detail in [chapter 8](#). Therefore, there are no learning objectives defined for test managers in this section. If you're studying for the ISTQB Advanced Test Manager exam, remember that the contents of the ISTQB Advanced syllabus are covered at a K1 level and thus are examinable. Therefore, read this section in [Chapter 2](#) of the Advanced syllabus for general recall and familiarity only.

2.3. Test Planning and Control

Learning objectives

(K2) Describe, giving examples, how test strategies affect test planning.

(K2) Compare test work products and explain by giving examples of the relationships between development and testing work products.

(K2) Classify related test control activities to determine if test mission, strategies, and objectives have been achieved.

In an analytical test process, such as that described by the ISTQB fundamental test process, test planning occurs at the beginning. Ideally, it proceeds in parallel with overall project planning. It addresses the implementation of the test strategy. For example, in a risk-based analytical test strategy, quality risk analysis will determine the prioritization and effort allocation for the testing. The test plan will then describe the test work that must be done to carry out the tests according to that priority and with the planned amount of effort. We'll cover risk analysis, risk-based testing, and test plan documentation thoroughly in [chapter 3](#).

ISTQB Glossary

test planning: The activity of establishing or updating a test plan.

test plan: A document describing the scope, approach, resources, and schedule of intended test activities. It identifies, amongst others, test items, the features to be tested, the testing tasks, who will do each task, the degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used and the rationale for their choice, and any risks requiring contingency planning. It is a record of the test planning process.

Test planning is not only concerned with planning a single level of testing but should address all the levels of testing that must occur. In other words, there should be a master test plan that spans the various levels and ensures that the right amount of testing happens at the right time. For example, if we consider a particular quality risk area as very dangerous, the master test plan should include coverage of that risk area early in the lifecycle and probably at multiple levels in the lifecycle.

During test planning, we also set a framework for deriving test cases, test conditions, and test procedures from the test basis. The test basis can include the requirements specifications, the design specifications, the quality risks, and other items. A good test plan, properly executed, helps to ensure that we actually do test what we set out to test during test execution. This sounds simple, but the plan must deal with the complex relationship between, on the one hand, the test basis as it exists during the test planning for the project, and on the other hand, the test cases, conditions, and procedures as they exist during the test execution for the project.

Once we have established the test plan, test control begins. Test control is an ongoing activity. In test control, we compare actual progress against the plan. We report the status of executing the plan, including any deviations that arise. Because projects are not static, but rather are evolving, test control is necessary to guide the testing to fulfill the mission, strategies, and objectives of testing.

Test control must respond to the changes in the mission, strategies, and objectives of testing, or to changes in the project. For example, we might discover that our risk analysis was mistaken in terms of defect likelihood in one area, which would cause us to re-allocate the remaining effort and reprioritize the tests. Or, we might have to deal with a reduced amount of time for test execution, leading us to use risk to guide a reduction in the total number of tests.

ISTQB Glossary

test case: A set of input values, execution preconditions, expected results, and execution postconditions developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.

test condition: An item or event of a component or system that could be verified by one or more test cases (e.g., a function, transaction, feature, quality attribute, or structural element).

test execution: The process of running a test on the component or system under test, producing actual result(s).

test procedure: A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script.

test control: A test management task that deals with developing and applying a set of corrective actions to get a test project on track when monitoring shows a deviation from what was planned.

Test planning and control often involve the creation or updating of documents. Test documentation for test planning and control are covered in chapter 3.

2.3.1. Test Planning and Control Case Study

Let's use an example to illustrate the test planning and control activities. When RBCS went through a recent project to redesign our website, we hired a marketing firm to help us create the content and a web development company to implement the site. We had a statement of work, included in the contract, with the web development company. This statement of work included the requirements for the site.

So, as part of the project, we planned to do an acceptance test. The primary objective for this acceptance testing, as with most testing at this level, was to demonstrate readiness for release. Since the requirements formed the contract between RBCS and the web development company, we used a

requirements-based testing strategy.

Based on this strategy, we developed a plan to cover all the stated and implied requirements. This plan involved documented test cases, though they were documented at the logical level, not the concrete level. We also used some amount of exploratory testing in areas where the requirements were vague or implied.

In [figure 2-2](#) you can see an example of how two paragraphs in one section of the requirements specification became four test cases. Notice that the first two test cases are relatively clear and unambiguous. There are a number of different test conditions possible in these test cases, which cause a lot of variation in terms of what was tested here. However, the test oracle—how we can determine expected behavior—is clear: If a returning customer has to reenter billing or shipping information after signing in, or cannot sign in at all, that is a bug.

Figure 2-2. Translating requirements to test cases

<i>Requirements excerpt</i>	<i>Test suite excerpt</i>
<p>Returning customers can use the sign in page to avoid having to re-enter all of their shipping and billing information when they place an order. The sign in and customer account management features have a structure and functionality similar to what is becoming standard on most major e-commerce sites.</p> <p>[Vendor] will provide a letter certifying PCI compliance, suitable for forwarding to Paymentech and/or credit card companies if needed.</p>	<ol style="list-style-type: none">1. Returning customers can use the sign in page to avoid re-entering their shipping information2. Returning customers can use the sign in page to avoid re-entering their billing information when they place an order3. The sign in and customer account management features have a structure and functionality similar to...major e-commerce sites.4. [Vendor] will provide a letter certifying PCI compliance

The third test case is, comparatively, quite ambiguous. What is the expected behavior? There are millions of e-commerce sites. Which one defines the standard? In practice, we used [Amazon.com](#) as the reference site. You could say that it really didn't matter which major site we picked as long as we picked a major site, and I would agree.

The fourth test case is clear on the face of it. Either there's a compliance letter, and it's professional in appearance, or there is not. However, if you ask, "Well, what does PCI compliance require?" then that opens up the question of what should this letter's content be? Given that the vendor was

also the host of the site—and thus the vendor is at risk if it didn't comply—we decided to let the vendor's staff worry about those details.

In [Table 2-1](#) you can see part of the test control mechanism used for this test. Generally, simple test projects can utilize simple test controls. Later in this book, you'll see examples of complex test status reporting and control mechanisms. However, on this simple project, tracking of test case status and defect status was sufficient.

Table 2-1. Test control based on defect status

Pass Two Defect Status	
Verified	47
To be verified	1
Need RBCS approval	5
Deferred	21
Failed verification	3
Not a defect	8
New	18
Total	103

[Table 2-1](#) shows the defect status at the end of the second pass of testing. We ran one pass of our tests, running each test case once, and found defects. We then let the vendor fix the defects and ran another pass. Ultimately, three passes of testing were required to fulfill the objective of compliance with requirements.

That said, the acceptance test failed in its objectives in a way. A typical acceptance test should not detect defects. Certainly, in this case, since we had given the vendor our tests, we were highly disappointed in the test results. Failure of an acceptance test in a situation like this is tantamount to shifting costs of failure onto the customer. We'll cover costs of quality and the test management implications of them later in this book, but at this point, I will say we are not a referenceable client for this web company.

To close this section, let's look at metrics and measurements for test planning and control. To measure completeness of this portion of the test process, we can measure the following:

- Risk and test coverage
- Defect discovery rates and other defect information
- Planned versus actual hours to develop testware and execute test cases

A work breakdown structure, established during test planning, will be useful for test control to determine whether we are proceeding according to the estimate and schedule.

2.4. Test Analysis and Design

Learning objectives

Recall of content only

The concepts in this section apply primarily for test analysts. There are no learning objectives defined for test managers in this section.

If you're studying for the ISTQB Advanced Test Manager exam, remember that the contents of the ISTQB Advanced syllabus are covered at a K1 level and thus are examinable. Therefore, read this section in [Chapter 2](#) of the Advanced syllabus for general recall and familiarity only.

ISTQB Glossary

test implementation: The process of developing and prioritizing test procedures, creating test data, and, optionally, preparing test harnesses and writing automated test scripts.

2.5. Test Implementation and Execution

Learning objectives

(K2) Explain the preconditions for test execution.

(K2) Explain by giving examples the advantages and disadvantages of early test implementation considering different testing techniques.

(K2) Explain the reasons why users and/or customers might be included in test execution.

(K2) Describe how the degree of test logging might vary depending on test level.

During test implementation, the test manager must ensure that we have handled all remaining tasks required to start test execution. This includes making sure that the test procedures are organized and accessible to the testers. This includes determining who will run tests when and what test data, test cases, test environments, and test tools they will use so that we can finalize the work schedule appropriately. And, this includes checking both explicit entry criteria (which you documented in your test plan) and implicit entry criteria (which might be in the project plan or just in people's heads).

Two test management challenges for implementation are to understand the sources of test data and to use those sources to obtain test data and make that data available to testers. You might need to arrange for the use of production or live data, which can lead to issues related to data privacy. You might need to acquire a tool for test data generation, especially if you need to do load, volume, or performance testing. While test analysts will create and use the data, the test manager must assure that they have the data and tools available.

Another test management challenge relates to the test environments. These can be large and complex, in some projects, and significant planning work is required to ensure that they will be available. Some defects can be found only in production-like test environments, which are often expensive to procure and difficult to configure and manage. The test manager has to manage this work during implementation. This management includes checking test environment dependencies, such as when a test requires exclusive use of a test environment. The test analysts must configure and verify the test environments, and the test manager must make sure that

they have. Otherwise, tests will be blocked or the results will be invalid.

A good rule of thumb for testers is, "Find the scary stuff first." This means that we have to pick the right tests and run them in the right order. So test prioritization is important. In risk-based testing strategies, we prioritize tests based on the associated likelihood and risk of problems.

As testers finalize the test cases and test procedures, an important management question is, "What is the right level of documentation detail?" The answer is subject to a variety of variables.

Sometimes the answer is easy to determine. If you are managing a project subject to standards or regulations, those standards or regulations might tell you the required level of detail. Usually, the answer in such situations is that you must be very detailed, which can create a lot of implementation work.

Sometimes the answer is hard to determine. If there are no rules for the level of detail, then you must understand what is needed. What are the skill levels of your testers? How important is reproducibility of test results? Will the test procedures be automated, now or eventually?

This question of level of detail is an important test management question, but it's one that few test managers effectively manage. I would encourage you to assess the current level of details of your test cases and take steps to bring it under control.

Speaking of automation, test implementation includes automation scripting. Automated, scripted tests are, of course, very detailed since they must tell the test tool exactly what to do. Proper implementation of automated test scripts is critical for good return on the automation investment.

ISTQB Glossary

test script: Commonly used to refer to a test procedure specification, especially an automated one.

How early should test implementation occur in the test project? This can be another complex question, especially when your project requires detailed tests. On the one hand, the more detailed your test procedures, the more effort it takes to implement the tests, pressing you to start earlier. On the other hand, the more detailed your test procedures, the more sensitive your test procedures are to last-minute changes in user interfaces, requirements, and so forth, pressing you to start later. When you can use logical test cases rather than concrete test cases, it will allow you to start implementation earlier and proceed at a more measured pace.

Finally, it is often a good idea to use blended test strategies that include analytical and dynamic elements. For analytical tests, test analysis, design, and implementation precede test execution. For dynamic tests, significant amounts of test analysis, design, and implementation work occur during test execution. The test manager must understand which tests arise from analytical strategies and which tests arise from dynamic strategies.

2.5.1. Test Execution

Test execution can start once the test team has received the test object and the entry criteria have been satisfied. The test manager needs to manage both of these areas to prevent premature entry into a test level and unnecessary delays.

During test execution, the testers execute the tests per the test procedures. Depending on the strategy, there is often some amount of exploratory latitude granted to testers running manual tests. This latitude again is a matter of test management, because the test manager must ensure that this happens to the degree desired, neither more nor less. The test manager must ensure that appropriate controls assure the capture of necessary information for exploration that occurs during test procedures since that exploratory testing is not defined in the procedures nor is it inherently traceable to the test basis without the capture of some additional

information.

As testers run the test procedures, they compare the actual results with the expected results. The test manager must ensure that the level of test result misinterpretation is kept to a tolerable minimum. False positives occur when a test is marked as failed when the behavior was correct. False negatives occur when a test is marked as passed when the behavior was incorrect. Both situations represent failure of the test process to generate correct information. So the test manager should track the number of such incidents and try to keep the number low.

When actual and expected results do not match, the tester has observed an anomaly. This is the first step in the identification and reporting of incidents. To ensure that a false positive has not occurred, testers must investigate incidents carefully and with the right degree of skepticism toward the tests themselves. An important test management concern is whether the number of false positives that occur arises from excessive and unmanaged change to the test basis, such as the requirements specification.

As the testers execute tests, they should log the results. The test manager should ensure that these logs provide an adequate record of relevant details for tests and events. This is especially true if events occurred during testing that slowed or even stopped test execution. The test manager can expect to be called on by management to explain these kinds of delays because test execution is on the critical path for release. The test manager should also ensure that adequate configuration management is occurring so that the test team can clearly and unambiguously link the logs to the particular version of the test object under test. And, of course, the test manager should ensure that the test logs contain the necessary data for the test results metrics that will be reported.

The level of detail of test logging, like the level of detail of test documentation, tends to vary depending on the level of testing and the strategy. For early levels of testing like unit tests, logging tends to be quite sparse. For later levels of testing, especially if regulatory standards apply, logging tends to be more verbose.

2.5.2. Case Study of Test Execution Preconditions

Let's look at some examples of preconditions for starting test execution. In figure 2-3 you can see the entry criteria for a test project that RBCS ran.

These entry criteria focus on three typical test preconditions:

- Readiness of the test environment
- Configuration and release management for the system under test
- The readiness of a defect management and a test management system

Another typical test precondition, the readiness of the test cases, test data, and so forth, is not explicitly measured for this project in the test plan. On this project, we followed a risk-based analytical test strategy, so we implemented test cases early during the project, well before entering system test. This proved to have both good and bad aspects.

Figure 2-3. Case study of test execution entry criteria

1. Bug tracking and test tracking systems are in place.
2. All components are under formal, automated configuration management and release management control.
3. The Operations team has configured the System Test server environment, including all target hardware components and subsystems. The Test Team has been provided with appropriate access to these systems.

Shortly before system test was to start, the user interface team worked over a weekend. During that frenetic weekend, they changed the entire user interfaces. They did not communicate this action to the test team, either verbally or in email.

So, the next Monday morning we arrived and attempted to continue our tests, which we were then running informally against early versions of the system. Since we didn't know why all the tests were failing in terms of the user interface, we suspected a problem with configuration management. The entire test team—all eight people—spent the whole day researching this before a chance hallway conversation revealed what had actually happened.

The impact of this unmanaged change on our testing effort was twofold. Most immediately, there was the loss of 1 schedule day and 64 person-

hours chasing down these false positives we thought at first were bugs. The longer and more significant impact was the approximately 2 person-weeks of effort spent updating the tests.

If I had to manage this test project again, knowing what would happen, would I still choose to have the testers implement the tests early in the project? Yes, I would. Here's why: The early test implementation revealed some important requirements specification defects. Without the detailed test implementation work we did, I don't think we would have found as many of those problems.

On this same project, we also had a beta testing level. As explained in the Foundation syllabus, beta testing is a form of acceptance testing done by actual users for a mass-market product. The product marketing team instituted this beta testing effort. In [figure 2-4](#) you can see the objectives as defined by the marketing staff for this program.

Because the product was designed for noncomputer users, it was a challenge to find such people using typical beta program methods. So to try to replicate the approximate skill and computer experience level of the target market, the marketing team encouraged employees to recruit their friends and family as the beta testers.

This effort was a bit Janus-faced in its objectives, as you can see if you read the excerpt carefully. For one thing, typical of a beta test, the goal includes finding defects that the test team might find difficult to discover with formal system testing. That was a valuable objective from the test team's point of view because one thing we could not cover well were different phone line conditions for the modems.

Figure 2-4. The objective of a beta test

Objective

Have 500... employee sponsored people test the stability of [the product] and our network and to provide constant, detailed feed back on the product and service. This feedback will help us make improvements to our product before we go nationwide. The [Beta test] marks our product launch in to the market – therefore it is extremely important that every [employee] participate to the max! The results and feedback from this program **will define** our success in the market.

However, the beta program also included a marketing component. The marketing team wanted to gather testimonials and other useful marketing collateral from the project. For this reason, the marketing team—rather than the test team—managed the effort.

In [Figure 2-5](#) you see an example of test logging from the beta testers on this project. As you can see, this would not qualify as professional test logging by a team of professional testers doing, say, a system test. However, it's hard to enforce standards on volunteers. Similar problems often exist with user acceptance test logging and with unit test logging. In this case, unlike with many beta tests, there was a large volume of data from the testers. However, it was sometimes hard to interpret due to the ambiguity of some of the comments.

In an attempt to understand how these results fit with what we had found in formal testing, the marketing and technical support teams performed a comparison with the test results. That effort was not very conclusive, but the idea is a good one. It would be especially important in a user

acceptance test to compare results against the system testing done previously.

Figure 2-5. Results logging from beta testers

Contents news financial and sports coverage are excellent...

I haven't found any bugs yet, but I just got up and running. I'll keep you posted.

I just browsed through the weather maps and was very impressed.

your weather content needs some work
1) it always seems to be about 5 hours behind so it needs to be more "current"
2) would it be possible to get a satellite view of a more precise area such as where we live?

2.5.3. BS 7925/2 Standard

The British Standards Institute produces the BS 7925/2 standard. It has two main sections: test design techniques and test measurement techniques. It reviews a wide range of test design techniques, including black-box, white-box, and others. It covers three black-box techniques that were also covered in the Foundation syllabus: equivalence partitioning, boundary value analysis, and state transition testing. It also covers a black-box technique called cause-effect graphing, which is a graphical version of a decision table, and a black-box technique called syntax testing.

For white-box techniques, it covers statement testing and branch/decision

testing, which were also covered in the Foundation syllabus. It also covers some additional white-box techniques that were covered only briefly or not at all in the Foundation: data flow testing, branch condition testing, branch condition combination testing, modified condition decision testing, and Linear Code Sequence and Jump (LCSAJ) testing.

Rounding out the list are two sections, "Random Testing" and "Other Testing Techniques." Random testing was not covered in the Foundation—and won't be covered in this book, though I will address it in the volume on technical test analysis. The section on other testing techniques doesn't provide any examples but merely talks about rules on how to select them.

If you intend to take the Advanced Test Manager exam, you might be thinking, "Hey, wait a minute, that was too fast. Which of those do I need to know for the exam?" Any test design technique that was on the Foundation syllabus, you had better know it; it's fair for the Advanced Test Manager exam.

BS 7925/2 provides one or more coverage metrics for each of the test measurement techniques. The organization of the document is curious indeed because there is no clear reason why the coverage metrics weren't covered at the same time as the design techniques. However, from the point of view of the ISTQB fundamental test process, perhaps it is easier that way. For example, our entry criteria might require some particular level of test coverage, as it would if we were testing safety-critical avionics software subject to the United States Federal Aviation Administration's standard DO-178B (a standard covered later in this book). So, during test design, we would employ the test design techniques. During test implementation, we'd use the test measurement techniques to ensure adequate coverage.

In addition to these two major sections, this document also includes two annexes. Annex B brings the dry material in the first two major sections to life by showing an example of applying them to realistic situations. Annex A covers process considerations, which is perhaps closest to our area of interest here. It discusses the application of the standard to a test project, following a test process given in the document.

We can map that process to the ISTQB fundamental test process. Test analysis and design along with test implementation in the ISTQB process is equivalent to test specification in the BS-7925/2 process. BS 7925/2 test execution, logically enough, corresponds to test execution in the ISTQB

process. Note, though, that the ISTQB process includes test execution as part of a larger activity, test implementation and execution. Note also that the ISTQB process includes test logging as part of test execution, while BS 7925/2 has a separate test recording process. Finally, BS 7925/2 has checking for test completion as the final step in its process. That step corresponds roughly to the ISTQB's evaluating test criteria and reporting.

To close this section, let's look at the metrics and measurements we can use for the test implementation and execution of the ISTQB fundamental test process. Different people use different metrics, of course.

Typical metrics during test implementation are the percentage of test environments configured, the percentage of test data records loaded, and the percentage of test cases automated. During test execution, typical metrics look at the percentage of test conditions covered, test cases executed, and so forth.

A work-breakdown-structure is also useful in determining whether we are proceeding according to the estimate and schedule.

Note that here we are discussing metrics to measure the completeness of these processes, i.e., the progress we have made. You should use a different set of metrics for test reporting.

2.6. Evaluating Exit Criteria and Reporting

Learning objectives

(K2) Summarize the information necessary to collect during the test process to support accurate reporting and evaluation against exit criteria.

The evaluation of exit criteria and reporting of results is a test management activity. We'll introduce ways to analyze, graph, present, and report test results as part of the test progress monitoring and control activities and cover that topic in detail in [chapter 3](#).

In terms of the test process, we need to consider two main areas. First, we need to collect the information necessary to support test management reporting of results. Second, we need to measure progress toward completion, and by extension, we need to detect deviation from the plan.

To measure completeness of the testing with respect to exit criteria, and to generate information needed for reporting, we can measure properties of the test execution process such as the following:

ISTQB Glossary

exit criteria: The set of generic and specific conditions, agreed upon with the stakeholders, for permitting a process to be officially completed. The purpose of exit criteria is to prevent a task from being considered completed when there are still outstanding parts of the task that have not been finished. Exit criteria are used to report against and to plan when to stop testing.

- Number of test conditions, cases, or test procedures planned, executed, passed, and failed
- Total defects, classified by severity, priority, status, or some other factor
- Change requests proposed, accepted, and tested
- Planned versus actual costs, schedule, effort
- Quality risks, mitigated and residual
- Lost test time due to blocking events
- Confirmation and regression test results

A work breakdown structure is also useful in determining whether we are proceeding according to the estimate and schedule.

[figure 2-6](#) shows, in miniature, four typical test results reporting graphics. Let's briefly cover each one and the data we need to capture during the test process to create them.

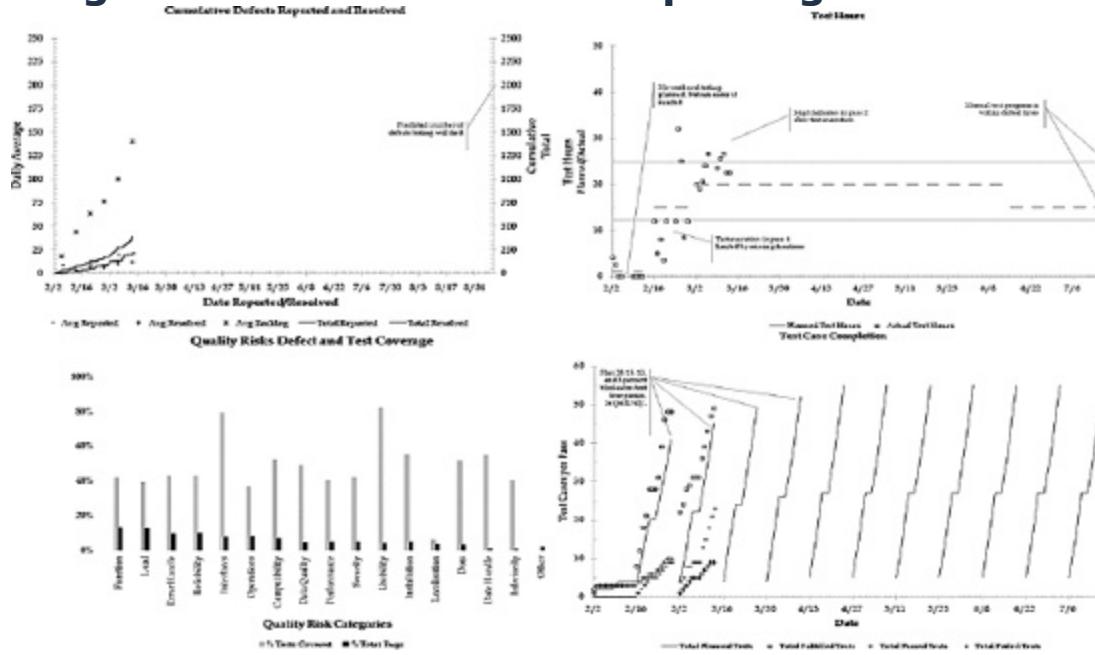
At the upper left is the open/closed, or bug convergence, chart. This chart measures the bug find rate and the bug backlog, which is a good way to determine how close we are to finishing the bug finding and bug fixing work for this project. This chart requires the open date and the close date for every defect.

Moving clockwise, at the upper right is the test progress chart. This measures whether the testing process is stable and efficient on a daily

basis. This chart requires the planned and actual number of test hours achieved for each day.

At the lower right is the test fulfillment chart. This chart measures how many of the planned test cases we have completed and the relative number passed and failed. This chart must show the total number of test cases we plan to run as of the end of each day, the actual number of test cases we ran, the actual number of test cases that passed, and the actual number of test cases that failed.

Figure 2-6. A test results reporting dashboard



Finally, at the lower left is the test coverage chart. This chart measures whether we have covered the test basis—in this example, the quality risks—and how many bugs we are finding relative to each major risk area. This chart requires the test cases and the bug reports to be traceable and analyzable in terms of the test basis.

All four of these charts (figure 2-6) and more such charts will be covered in great detail in chapter 3.

Let's close this section with a quick look at the IEEE 829 standard for test documentation, in this case, the template for test summary reports. A test summary report describes the results of a given level or phase of testing. The IEEE 829 template includes the following sections:

- Test summary report identifier

- Summary (e.g., what was tested, what the conclusions are, etc.)
- Variances (from plan, cases, procedures)

ISTQB Glossary

test closure: During the test closure phase of a test process, data is collected from completed activities to consolidate experience, testware, facts, and numbers. The test closure phase consists of finalizing and archiving the testware and evaluating the test process, including preparation of a test evaluation report.

test summary report: A document summarizing testing activities and results. It also contains an evaluation of the corresponding test items against exit criteria.

- Comprehensive assessment
- Summary of results (e.g., final metrics, counts)
- Evaluation (of each test item vis-à-vis pass/fail criteria)
- Summary of activities (resource use, efficiency, etc.)
- Approvals

The test manager can deliver these summaries during test execution, as part of a project status report or meeting. They can also be used at the end of a test level as part of test closure activities.

2.7. Test Closure Activities

Learning objectives

(K2) Summarize the four groups of test closure activities.

(K3) Generalize lessons learned in the test closure period in order to discover areas to improve or repeat.

Test closure work is the often forgotten but important work at the end of the test process to guarantee and capture the remaining value available to the test team and the larger project team and the organization as a whole. There are four main types:

- Ensuring that test work is concluded
- Delivering test work products
- Participating in project retrospectives
- Archiving test work products

As I mentioned, test closure work is often forgotten, or at least one or more of these activities are often forgotten. In some cases, they are not so much forgotten as simply omitted, whether due to premature dissolution of the test or project team, immediate reallocation of resources to respond to schedule pressures on subsequent projects, or simple team burnout. For contract development, the contract should specify the closure activities to make sure they are not forgotten.

Let's look at each of these. First is the simple act of checking to make sure we are really done. Given the complexity of managing testing—especially on large or distributed projects—it's easy to forget something. Smart test managers, as the testing draws to a close, double-check the test plan to make sure that everything planned was done. They check the test tracking information to make sure that all planned tests were either run to conclusion or were deliberately skipped (and approved for skipping). They check that all known defects were fixed (including the confirmation testing of the fix), deferred (with appropriate blessing from the project team), or accepted as permanent restrictions.

Next is looking for opportunities to reuse test work products. After a release, many of these work products can be useful to other people on the

project team, on support teams, or even to customers. For example, we can tell users, customers, or support people about known defects that were deferred or accepted. We can transfer tests and test environments to those who will perform maintenance testing. We can deliver automated or manual regression test packs to customers if they are integrating our system into a larger system of systems.

The most common type of test closure activity is the use of project retrospectives to document lessons learned (both good and bad) from the project and to plan to manage the events, surprises, and problems of future projects better. For example, during the analysis of a project retrospective, we might find that we saw many unexpected defect clusters during testing. This discovery could lead us to expand the list of quality risk analysis participants on future projects.

We might find that we missed our estimates significantly in a number of areas. This could lead us to evaluate the root causes for these deviations. Such causes could include more defects than expected, more test work than expected, and so forth. We can then adopt plans to resolve those causes in the future.

We might find that defect trends and causes reveal many opportunities to find the defects earlier, to reduce the number of defects, or to prevent defects entirely. We might locate areas where testing, or the project as a whole, was inefficient or ineffective. We can then identify process, team, or tool improvements.

Keep in mind that retrospectives should apply to testing as well as to the entire project and indeed the wider organization. Problems tend to be systemic, not isolated. When you have opportunities to improve, there are often enablers and disablers outside of the target improvement group.

Finally, test closure includes safeguarding the information products. Testing delivers intermediate and final test results, test log files, test status reports, and other documents and work products. We should place these into the configuration management system. That system must link the various test work products to the system under test (both in terms of names and versions). For example, we should capture and archive both the test plan and the project plan, along with the test cases, test procedures, test data, and test results.

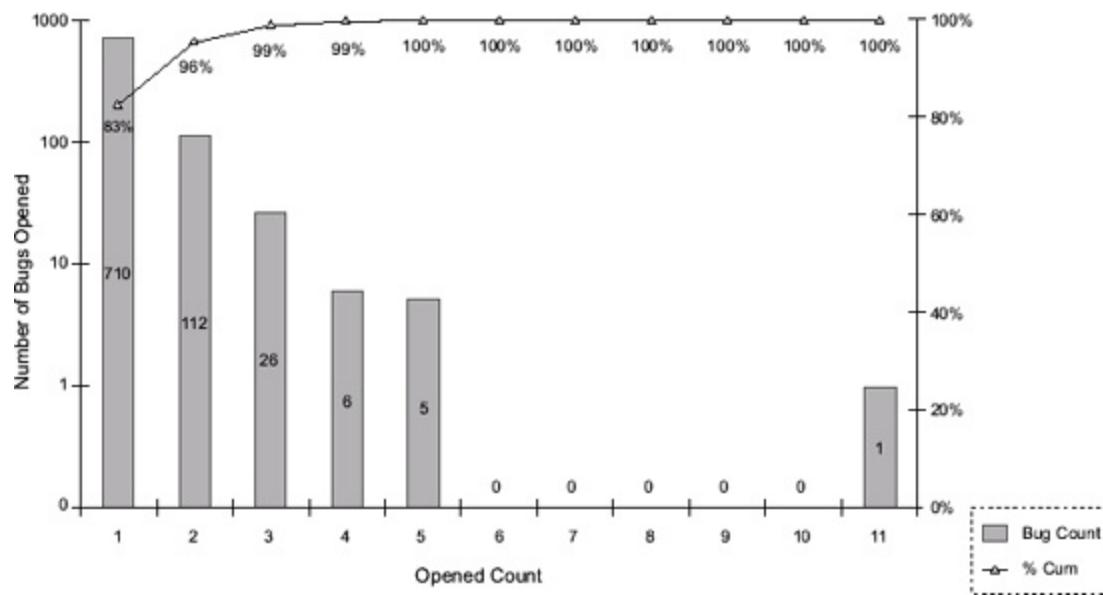
2.7.1. Two Case Studies of Test Closure

As an example of valuable test work deliverables given to another party after the project, consider this example from the Internet appliance project. Toward the end of the project, just before release, I assigned a tester the job of communicating all known issues to the technical support manager and key participants on the technical support team. This report included detailed information on all failed test cases, to the point of the specific step in the test that failed, the actual result as opposed to expected result, and, if possible, any workaround we had identified. It also included a complete review of all deferred or unresolved bugs. According to the technical support manager, this resulted in a significant time savings, as much as 300 to 400 percent, for calls related to known failed tests and defects.

On one occasion, my test team and I came in to clean up the testing on a second try of a failed project. My first step was to analyze test results for a retrospective. [figure 2-7](#) shows an example of a chart I created as part of this retrospective analysis. The testing went rather poorly, and a number of reasons were revealed by this analysis.

Figure 2-7. A high failure rate of confirmation testing

Case Study Banking Application Bugs
Confirmation Test Failure Analysis



In this chart, we can see the number of times that the various bug reports were opened. If a bug is discovered, reported, fixed by development, closed, and then never seen again, the bug count metric for that bug is one. If a bug is discovered, reported, allegedly fixed, reopened by test during confirmation testing, then actually fixed, closed, and never seen again, the bug count metric for that bug is two. Alternatively, if the bug is discovered,

reported, fixed, closed during confirmation testing, but then seen again later—due to regression, for example—and only the second time is it finally resolved, the bug count metric for that bug is also two. The scale, by the way, is logarithmic to prevent the large number of bugs from distracting attention to the key point of the chart.

The key point of the chart is the large number of bug reports that required attention more than once. As you can see, fully one-sixth (about 17 percent) of the bug reports were reopened at least once. Is this a big deal? Well, assume that each bug report requires at least one hour of effort to handle. In that case, we can read the total waste associated with incomplete resolution of bugs, from left to right on the chart, starting with the second bar on the histogram, as follows:

There were 112 bug reports opened twice. The second time was wasted time, so we can count 112 hours of waste for the bug reports opened twice.

There were 26 bug reports opened 3 times. The second and third time were wasted time, so we can count 2 times 26, or 52, hours of waste for the bug reports opened 3 times.

There were 6 bug reports opened 4 times. The second, third, and fourth time were wasted time, so we can count 3 times 6, or 18, hours of waste for the bug reports opened 4 times.

There were 5 bug reports opened 5 times. The second, third, fourth, and fifth time were wasted time, so we can count 4 times 5, or 20, hours of waste for the bug reports opened 5 times.

There was 1 bug report opened 11 times. The second through eleventh time were wasted time, so we can count 10 times 1, or 10, hours of waste for the bug reports opened 11 times.

That totals more than 200 person-hours or 5 person-weeks of wasted effort associated with having to process and reprocess bug reports. The moral of this story is clear: Fix once and fix right.

To close this section, let's look at metrics and measurements for test closure. To measure completeness of this portion of the test process, we can measure the following:

- Percentage of test cases run during test execution

- Percentage of test cases checked into repository
- Percentage of test cases automated
- Percentage of test cases for regression testing
- Percentage of deferred defect reports
- Percentage of work products archived

A work breakdown structure is also useful to compare our results against our estimate and schedule, which is good to know in the project retrospective.

2.7.2. Test Closure Activities Exercise

After the first release of the HELLOCARMS system, you hold a retrospective meeting. In researching the results, you discover the following:

- Twenty-seven percent of defects reported against reverse mortgage function were rejected as not actual problems.
- Twenty-five percent of defects reported against optional life insurance feature were rejected as not actual problems.
- Thirty-seven percent of defects reported against LoDoPS were rejected as not actual problems.

The average defect report rejection rate is 5 percent for your team. You consider that acceptable, but anything above 10 percent is unacceptable.

List at least five potential improvements to investigate for the second and subsequent releases.

2.7.3. Test Closure Activities Exercise Debrief

Your list should have included some of the following:

- Use training, hiring, or user assistance to increase reverse mortgage expertise in the test team.
- Evaluate whether the requirements associated with the particular rejected reverse mortgage defects were ambiguous or otherwise faulty.
- Check for a high incidence of field-reported defects associated with

rejected reverse mortgage defects.

- Use training, hiring, or user assistance to increase life insurance expertise in the test team.
- Evaluate whether the requirements associated with the particular rejected life insurance defects were ambiguous or otherwise faulty.
- Check for a high incidence of field-reported defects associated with rejected life insurance defects.
- Use training, hiring, or user assistance to increase LoDoPS expertise in the test team.
- Evaluate whether the requirements associated with the particular rejected LoDoPS defects were ambiguous or otherwise faulty.
- Check for a high incidence of field-reported defects associated with rejected LoDoPS defects.
- Set up a meeting with developers to discuss why particular problems were rejected and to brainstorm ways to improve communication between the two teams.
- Have a defect review process so that each defect report is checked by one or more testers before the report is entered into the tracking system.

You can probably think of others as well. Remember, the important thing when considering improvements, is to use data.

2.8. Sample Exam Questions

To end each chapter, you can try one or more sample exam questions to reinforce your knowledge and understanding of the material and to prepare for the ISTQB Advanced Test Manager exam.

Q: Assume you are a test manager working on a project to create a programmable thermostat for home use to control central heating, ventilation, and air conditioning (HVAC) systems. This project is following a sequential lifecycle model, specifically the V-model. Your test team is following a risk-based testing strategy augmented with a reactive testing strategy during test execution to ensure that key risks missed during risk analysis are caught.

Of the following statements, identify all that are true about the test plan.

1. The test plan should include risk analysis early in the process.
2. The test plan should list the requirements specification as an input to the risk analysis.
3. The test plan should assign specific bugs to specific developers.
4. The test plan should discuss the integration of reactive test techniques into test execution.
5. The test plan should specify overall project team metrics used to determine bonuses.

Q: 2. Which of the following is a test document in which you would expect to find the preconditions to start executing a level of testing?

1. Test plan
2. Test design specification
3. Incident report
4. Project plan

Q: Which of the following is the most likely reason a user might be

included in test execution?

- 1.** Their application domain knowledge
- 2.** Their technical expertise
- 3.** Their testing expertise
- 4.** Their management expertise

Q: You are the manager of a bank's quality assessment group, in charge of independent testing for banking applications. You have just concluded a project to implement an integrated system that will use three off-the-shelf systems to manage a bank's accounts-receivable system. During the project, you found that one of the vendor's systems, while comprising approximately the same amount of functionality and of roughly the same complexity as the other two systems, had significantly more defects. Making no assumptions other than ones based on the information provided here, which of the following is a reasonable improvement to the test process for subsequent projects involving this vendor?

- 1.** Impose retroactive financial penalties on this vendor for the number of bugs delivered on this project.
- 2.** Perform an acceptance test for all systems received, with particular rigor for this vendor.
- 3.** Cancel the contract with this vendor and put it on an industry blacklist.
- 4.** Require the vendor's developers to attend training to improve their ability to write quality code.

Chapter 3. Test Management

"The plan is for us to give your team a test release, and you'll run a cycle of tests against it. If you find any bugs, we'll fix them over the weekend, then give you a final release. After you confirm that everything works, we'll be ready to ship."

—An overly optimistic development manager describing his expectations for the test execution period to the author. In actuality, the test execution turned out somewhat differently, and they found quite a few more bugs than expected.

The third chapter of the Advanced syllabus is concerned with test management. It discusses test management activities from the start to the end of the test process and introduces the consideration of risk for testing. There are 11 sections:

- 1.** Introduction
- 2.** Test Management Documentation
- 3.** Test Plan Documentation Templates
- 4.** Test Estimation
- 5.** Scheduling Test Planning
- 6.** Test Progress Monitoring and Control
- 7.** Business Value of Testing
- 8.** Distributed, Outsourced, and Insourced Testing
- 9.** Risk-Based Testing
- 10.** Failure Mode and Effect Analysis
- 11.** Test Management Issues

Let's look at each section and how it relates to test management.

ISTQB Glossary

risk analysis: The process of assessing identified risks to estimate their impact and probability of occurrence (likelihood).

3.1. Introduction

Learning objectives

Recall of content only

This chapter, as the name indicates, focuses primarily on test management topics. Thus, it is of primary importance to Advanced Test Manager exam candidates. You should plan to spend considerably more time on this chapter than other chapters in the syllabus.

This chapter also covers test documentation templates for test managers. It focuses on the IEEE 829 standard.

We're going to take the topics in this chapter just a little bit out of order. I tend to follow a risk-based analytical testing strategy. In such a strategy, risk analysis precedes all the other testing tasks except for establishing the goals and context of testing.

So, we're going to start with the two sections on risk analysis. These are [sections 3.9](#) and [3.10](#) of the syllabus. Because the quality risk analysis exercise follows "Failure Mode and Effects Analysis", [section 3.10](#), which addresses that technique, I cover that technique together with [section 3.9](#). Once we finish this combined section, we return to the remaining sections of the chapter in order.

I also combine [sections 3.2](#) and [3.3](#) for similar reasons. The test planning exercise from [section 3.2](#) requires the use of the IEEE 829 test plan template, which is discussed in [section 3.3](#).

To help orient you with the syllabus, I've retained the section names from the syllabus. Don't be surprised when you see that the section names aren't in the same order as they are in the syllabus, and when some sections address multiple sections of the syllabus.

3.2. Risk-Based Testing and Failure Mode and Effects Analysis

Learning objectives

- (K2) Explain the different ways that risk-based testing responds to risks.
- (K4) Identify risks within a project and product, and determine an adequate strategy and test plan based on these risks.
- (K3) Execute a risk analysis for product from a tester's perspective, following the failure mode and effects analysis approach.
- (K4) Summarize the results from the various perspectives on risk typically held by key project stakeholders, and use their collective judgment in order to outline test activities to mitigate risks.
- (K2) Describe characteristics of risk management that require it to be an iterative process.
- (K3) Translate a given risk-based test strategy to test activities and monitor its effects during the testing.
- (K4) Analyze and report test results, including determining and reporting residual risks to enable project managers to make intelligent release decisions.
- (K2) Describe the concept of FMEA, and explain its application in projects and benefits to projects by example.

Risk is the possibility of a negative or undesirable outcome or event. A specific risk is any problem that might occur that would decrease customer, user, participant, or stakeholder perceptions of product quality or project success.

In testing, we're concerned with two main types of risks. The first type is product or quality risks. When the primary effect of a potential problem is on the quality of the product itself, the potential problem is called a product risk. A synonym for product risk, which I use most frequently myself, is quality risk. An example of a quality risk is a possible reliability defect that could cause a system to crash during normal operation.

ISTQB Glossary

product risk: A risk directly related to the test object.

project risk: A risk related to management and control of the (test) project, e.g., lack of staffing, strict deadlines, changing requirements, etc.

risk: A factor that could result in future negative consequences; usually expressed as impact and likelihood.

The second type of risk is project or planning risks. When the primary effect of potential problem is on the overall success of a project, those potential problems are called project risks. Some people also refer to project risks as planning risks. An example of a project risk is a possible staffing shortage that could delay completion of a project.

Not all risks are equal in importance. There are a number of ways to classify risks by level of risk. The simplest way is to look at two factors:

- The likelihood of the problem occurring
- The impact of the problem should it occur

Likelihood of a problem arises primarily from technical considerations, such as the programming languages used, the bandwidth of connections, and so forth. The impact of a problem arises from business considerations, such as the financial loss the business will suffer, the number of users or customers affected, and so forth.

In risk-based testing, we use the risk items identified during risk analysis, together with the level of risk associated with each risk item, to guide our testing. In fact, under a true analytical risk-based testing strategy, risk is the primary basis of testing.

Risk can guide testing in various ways, but there are three very common ones:

- First, during all test activities, test analysts and test managers allocate effort for each quality risk item proportional to the level of risk. Test analysts select test techniques in a way that matches the rigor and extensiveness of the risk.

the technique with the level of risk. Test managers and test analysts carry out test activities in reverse risk order, addressing the most important quality risks first and only at the very end spending any time at all on less important ones. Finally, test managers and test analysts work with the project team to ensure that the prioritization and resolution of defects is appropriate to the level of risk

ISTQB Glossary

risk level: The importance of a risk as defined by its characteristics impact and likelihood. The level of risk can be used to determine the intensity of testing to be performed. A risk level can be expressed either qualitatively (e.g., high, medium, low) or quantitatively.

risk management: Systematic application of procedures and practices to the tasks of identifying, analyzing, prioritizing, and controlling risk.

- Second, during test planning and test control, test managers carry out risk control for all significant, identified project risks. The higher the level of risk, the more thoroughly that project risk is controlled. We'll cover risk control options in a moment.
- Third, test managers and test analysts report test results and project status in terms of residual risks. For example, which tests have we not yet run or have we skipped? Which tests have we run? Which have passed? Which have failed? Which defects have we not yet fixed or retested? How do the tests and defects relate back to the risks?

When following a true analytical risk-based testing strategy, it's important that risk management not be something that happens only at the start of a project. The three responses to risk I just covered—along with any others that might be needed—should occur throughout the lifecycle. Specifically, we should try to reduce quality risk by running tests and finding defects and reduce project risk through mitigation and, if necessary, contingency actions. Periodically in the project, we should reevaluate risk and risk levels based on new information. This might result in our reprioritizing tests and defects, reallocating test effort, and other test control actions. This will be discussed further later in this section.

One metaphor sometimes used to help people understand risk-based testing is that testing is a form of insurance. In your daily life, you buy insurance when you are worried about some potential risk. You don't buy insurance for risks that you are not worried about. So, we should test the areas and test for bugs that are worrisome and ignore the ones that aren't.

One potentially misleading aspect of this metaphor is that insurance professionals and actuaries can use statistically valid data for quantitative risk

analysis. Typically, risk-based testing relies on qualitative analyses because we don't have the same kind of data that insurance companies have.

During risk-based testing, you have to remain aware of many possible sources of risks. There are safety risks for some systems. There are business and economic risks for most systems. There are privacy and data security risks for many systems. There are technical, organizational, and political risks too.

3.2.1. Characteristics and Benefits of Risk-Based Testing

What does an analytical risk-based testing strategy involve? What characteristics and benefits does it have?

For one thing, an analytical risk-based testing strategy matches the level of testing effort to the level of risk. The higher the risk, the more test effort we expend. This means not only the effort expended in test execution, but also the effort expended in designing and implementing the tests. We'll look at the ways to accomplish this later in this section.

For another thing, an analytical risk-based testing strategy matches the order of testing to the level of risk. Higher-risk tests tend to find more bugs, or tend to test more important areas of the system, or both. So the higher the risk, the earlier the test coverage. This is consistent with a rule of thumb for testing that often tell testers, which is to try to find the scary stuff first. Again, we'll see how we can accomplish this later in this section.

Because of this effort allocation and ordering of testing, the total remaining level of quality risk is systematically and predictably reduced as testing continues. By maintaining traceability from the tests to the risks and from the located defects to the risks, we can report test results in terms of residual risk. This allows project stakeholders to decide to declare testing complete whenever the risk of continuing testing exceeds the risk of declaring the testing complete.

Since the remaining risk is going down in a predictable way, this means that we can triage tests in risk order. Should schedule compression require that we reduce test coverage, we can do this in risk order, providing a way that is both acceptable and explainable to project stakeholders.

ISTQB Glossary

risk identification: The process of identifying risks using techniques such as brainstorming, checklists, and failure history.

For all of these reasons, an analytical risk-based testing strategy is more robust than an analytical requirements-based test strategy. Pure analytical requirements-based test strategies require at least one test per requirement, they don't tell us how many tests we need in a way that responds intelligently to project constraints. They don't tell us the order in which to run tests. In pure analytical requirements-based test strategies, the risk reduction throughout test execution is neither predictable nor measurable. Therefore, with analytical requirements-based test strategies, we cannot easily express the remaining level of risk if project stakeholders ask us whether we can safely curtail or compress testing.

That is not to say that we ignore requirements specifications when we use an analytical risk-based testing strategy. On the contrary, we use requirements specifications, design specifications, marketing claims, technical support or help desk data, and myriad other inputs to inform our risk identification and analysis process if they are available. However, if we don't have this information available or if we find such information of limited usefulness, we can still plan, design, implement, and execute our tests by using stakeholder input to the risk identification and assessment process. This ability to subsist on little or no documentation also makes an analytical risk-based testing strategy more robust than an analytical requirements-based strategy, because we reduce our dependency on upstream processes (which we may not control) like requirements gathering and design.

All that said, an analytical risk-based testing strategy is not perfect. Like any analytical testing strategy, we will not have all of the information we need for perfect risk assessment at the beginning of the project. Even with periodic reassessment of risk—which I will also discuss later in this section—we will miss some important risks. Therefore, an analytical risk-based testing strategy, like any analytical testing strategy, should blend reactive strategies during test implementation and execution so that we can detect risks that we missed during our risk assessment.

Let me be more specific and concise about the testing problems we often face and how analytical risk-based testing can help solve them.

First, as testers, we often face significant time pressures. There is seldom sufficient time to run the tests we would want to run, particularly when doing requirements-based testing. Ultimately, all testing is time-boxed. Risk-based testing provides a way to prioritize and triage tests at any point in the lifecycle.

When I say that all testing is time-boxed, I mean that we face a challenge in determining appropriate test coverage. If we measure test coverage as a percentage of what *could be tested*, any amount of testing yields a coverage metric of 0 percent because the set of tests that could be run is infinite for an real-sized system. So, risk-based testing provides a means to choose a smart subset from the infinite number of comparatively small subsets of tests we could run.

Further, we often have to deal with poor or missing specifications. By involving stakeholders in the decision about what not to test, what to test, and how much to test, risk-based testing allows us to identify and fill gaps in documents like requirements specifications that might result in big holes in our testing. It also helps to sensitize the other stakeholders to the difficult problem of determining what to test (and how much) and what not to test.

To return to the issue of time pressure, not only are they significant, they tend to escalate during the test execution period. We are often asked to compress the test schedule at the start of or even midway through test execution. Risk-based testing provides a means to drop tests intelligently while also providing a way to discuss with project stakeholders the risks inherent in doing so.

Finally, as we reach the end of our test execution period, we need to be able to help project stakeholders make smart release decisions. Risk-based testing allows us to work with stakeholders to determine an acceptable level of residual risk rather than forcing them—and us—to rely on inadequate, tactical metrics like bug and test counts.

3.2.2. The History of Risk-Based Testing

How did analytical risk-based testing strategies come to be? Understanding the history can help you understand where we are and where these strategies might evolve.

In the early 1980s, Barry Boehm and Boris Beizer each separately examined the idea of risk as it relates to software development. Boehm advanced the idea of

risk-driven spiral development lifecycle, which we covered in the Foundation syllabus. The idea of this approach is to develop the architecture and design in risk order to reduce the risk of development catastrophes and blind alleys late in the project.

Beizer advanced the idea of risk-driven integration and integration testing. In other words, it's not enough to develop in risk order, we need to assemble and test in risk order, too.^[1]

[1] See Beizer's book *Software System Testing and Quality Assurance*.

If you reflect on the implications of Boehm and Beizer's ideas, you can see that these are the precursors of iterative and agile lifecycles.

Now, in the mid 1980s, Beizer and Bill Hetzel each separately declared that risk should be a primary driver of testing. By this, they meant both in terms of efficiency and in terms of order. However, while giving some general ideas on this, they did not elaborate any specific mechanisms or methodologies for making this happen. I don't say this to criticize them. At that point, it perhaps seemed that just ensuring awareness of risk among the testers was enough to ensure risk-based testing.^[2]

[2] See Beizer's book *Software Testing Techniques* and Hetzel's book *The Complete Guide to Software Testing*.

However, it was not. Some testers have followed this concept of using the tester's idea of risk to determine test coverage and priority. For reasons we'll cover later, this results in testing devolving into an ill-informed, reactive bug hunt. There's nothing wrong with finding many bugs, but finding as many bugs as possible is not a well-balanced test objective.

So more structure was needed to ensure a systematic exploration of the risks. This brings us to the 1990s. Separately, Rick Craig, Paul Gerrard, Felix Redmill, and I were all looking for ways to systematize this concept of risk-based testing. I can't speak for Craig, Gerrard, and Redmill, but I know that I had become frustrated with requirements-based strategies for the reasons mentioned earlier. So in parallel and with very little apparent cross-pollination, the four of us—and perhaps others—developed similar approaches for quality risk analysis and risk-based testing. In this section, you'll learn these approaches.^[3]

[3] For more details on my approach, see my discussion of formal techniques in *Critical Testing Processes* and my discussion of informal techniques in *Pragmatic Software Testing*. For Paul Gerrard's approach, see *Risk-based e-*

Business Testing. Van Veenendaal discusses informal techniques in *The Testing Practitioner*.

So, where are we now? In the mid- to late 2000s, test practitioners widely used analytical risk-based testing strategies in various forms. Some still practice misguided, reactive, tester-focused bug hunts. However, many practitioners are trying to use analytical approaches to prevent bugs from entering later phases of testing, to focus testing on what is likely to fail *and* what is important, to report test status in terms of residual risk, and to respond better as their understanding of risk changes. By putting the ideas in this section into practice, you can join in this endeavor. As you learn more about how analytical risk-based testing strategies work—and where they need improvements—I encourage you to share what you've learned with others by writing articles, books, and presentations on the topic.

However, while we still have much to learn, that does not mean that analytical risk-based testing strategies are at all experimental. They are well-proven practices. I am unaware of any other test strategies that adapt as well to the myriad realities and constraints of software projects. They are the best thing going, especially when blended with reactive strategies.

Another form of blending that requires attention and work is the blending of analytical risk-based testing strategies with all the existing lifecycle models. My associates have used analytical risk-based testing strategies with sequential lifecycles, iterative lifecycles, and spiral lifecycles. These strategies work regardless of lifecycle. However, the strategies must be adapted to the lifecycle.

Beyond learning more through practice, another important next step is for test management tools to catch up and start to advance the use of analytical risk-based testing strategies. Some test management tools now incorporate the state of the practice in risk-based testing. Some still do not support risk-based testing directly at all. I encourage those of you who are working on test management tools to build support for this strategy into your tools and look for ways to improve it.

ISTQB Glossary

risk control: The process through which decisions are reached and protective measures are implemented for reducing risks to, or maintaining risks within, specified levels.

risk mitigation: See *risk control*.

3.2.3. How to Do Risk-Based Testing

Let's move on to the tactical questions about how we can perform risk-based testing. Let's start with a general discussion about risk management, and then we'll focus on specific elements of risk-based testing for the rest of this section.

Risk management includes three primary activities:

- Risk identification, figuring out what the different project and quality risks are for the project
- Risk analysis, assessing the level of risk—typically based on likelihood and impact—for each identified risk item
- Risk mitigation (which is really more properly called "risk control" because it consists of mitigation, contingency, transference, and acceptance actions for various risks)

In some sense, these activities are sequential, at least in when they start. They are staged such that risk identification starts first. Risk analysis comes next. Risk control starts once we have determined the level of risk through risk analysis. However, because we should continuously manage risk in a project, risk identification, risk analysis, and risk control are all recurring activities.

Everyone has their own perspective on how to manage risks on a project, including what the risks are, the level of risk, and the appropriate controls to put in place for risks. So risk management should include all project stakeholders.

Test analysts bring particular expertise to risk management due to their defect focused outlook. They should participate whenever possible. In fact, in many cases, the test manager will lead the quality risk analysis effort with test analysts providing key support in the process.

Let's look at these activities more closely. For proper risk-based testing, we need to identify both product and project risks. We can identify both kinds of risks using techniques like these:

- Expert interviews
- Independent assessments
- Use of risk templates
- Project retrospectives
- Risk workshops and brainstorming
- Checklists
- Calling on past experience

Conceivably, you can use a single integrated process to identify both project and product risks. I usually separate them into two separate processes since they have two separate deliverables and often separate stakeholders. I include the project risk identification process in the test planning process. In parallel, the quality risk identification process occurs early in the project.

That said, project risks—and not just for testing but also for the project as a whole—are often identified as by-products of quality risk analysis. In addition, you use a requirements specification, design specification, use cases, and the like as inputs into your quality risk analysis process, you should expect to find defects in those documents as another set of by-products. These are valuable products, which you should plan to capture and escalate to the proper person.

Previously, I encouraged you to include representatives of all possible stakeholder groups in the risk management process. For the risk identification activities, the broadest range of stakeholders will yield the most complete, accurate, and precise risk identification. The more stakeholder group representatives you omit from the process, the more risk items and even whole risk categories will be missing.

How far should you take this process? Well, it depends on the technique. In informal techniques, which I frequently use, risk identification stops at the risk items. The risk items must be specific enough to allow for analysis and assessment of each one to yield an unambiguous likelihood rating and an unambiguous impact rating.

Techniques that are more formal often look "downstream" to identify potential effects of the risk item if it becomes an actual negative outcome. These effects include effects on the system—or the system of systems if applicable—as well on potential users, customers, stakeholders, and even society in general. Failure Mode and Effect Analysis is an example of such a formal risk management technique, and it is commonly used on safety-critical and embedded systems.¹

¹ For a discussion of Failure Mode and Effect Analysis, see Stamatis's book *Failure Mode and Effect Analysis*.

Other formal techniques look "upstream" to identify the source of the risk. Hazard analysis is an example of such a formal risk management technique. I never used it myself, but I have talked to clients who have used it for safety-critical medical systems.

We'll look at some examples of various levels of formality in risk analysis a little later in this section.

The Advanced syllabus refers to the next step in the risk management process as risk analysis. I prefer to call it risk assessment, just because analysis would seem to include both identification and assessment of risk to me.

Regardless of what we call it, risk analysis or risk assessment involves the study of the identified risks. We typically want to categorize each risk item appropriately and assign each risk item an appropriate level of risk.

We can use ISO 9126 or other quality categories to organize the risk items. In my opinion, it doesn't matter so much what category a risk item goes into, usually, so long as we don't forget it. However, in complex projects and for large organizations, the category of risk can determine who has to deal with the risk. A practical implication of categorization like this will make the categorization important.

3.2.4. The Level of Risk

The other part of risk assessment or risk analysis is determining the level of risk. This determination often involves likelihood and impact as the two key factors. Likelihood arises from technical considerations, typically, while impact arises from business considerations. However, in some formalized approaches you use three factors, such as severity, priority, and likelihood of detection, or even subfactors underlying likelihood and impact. Again, we'll discuss this further later in the book.

So, what technical factors should we consider? Here's a list to get you started:

- Complexity of technology and teams
- Personnel and training issues
- Intrateam and interteam conflict/communication
- Supplier and vendor contractual problems
- Geographical distribution of the development organization, as with outsourcing
- Legacy or established designs and technologies versus new technologies and designs
- The quality—or lack of quality—in the tools and technology used
- Bad managerial or technical leadership
- Time, resource, and management pressure, especially when financial penalties apply
- Lack of earlier testing and quality assurance tasks in the lifecycle
- High rates of requirements, design, and code changes in the project
- High defect rates
- Complex interfacing and integration issues
- Lack of sufficiently documented requirements

And what business factors should we consider? Here's a list to get you started

- The frequency of use and importance of the affected feature
- Potential damage to image
- Loss of customers and business
- Potential financial, ecological, or social losses or liability
- Civil or criminal legal sanctions

- Loss of licenses, permits, and the like
- The lack of reasonable workarounds
- The visibility of failure and the associated negative publicity

Both of these lists are just starting points.

When determining the level of risk, we can try to work quantitatively or qualitatively. In quantitative risk analysis, we have numerical ratings for both likelihood and impact. Likelihood is a percentage, and impact is often a monet quantity. If we multiply the two values together, we can calculate the cost of exposure, which is called—in the insurance business—the expected payout or expected loss.

While it will be nice some day in the future of software engineering to be able do this routinely, typically the level of risk is determined qualitatively. Why? Because we don't have statistically valid data on which to perform quantitative quality risk analysis. So we can speak of likelihood being very high, high, medium, low, or very low, but we can't say—at least, not in any meaningful way—whether the likelihood is 90 percent, 75 percent, 50 percent, 25 percent, or 10 percent.

This is not to say—by any means—that a qualitative approach should be seen inferior or useless. In fact, given the data most of us have to work with, use of a quantitative approach is almost certainly inappropriate on most projects. The illusory precision thus produced misleads the stakeholders about the extent to which you actually understand and can manage risk. What I've found is that if you accept the limits of my data and apply appropriate informal quality risk management approaches, the results are not only perfectly useful, but also indeed essential to a well-managed test process.

Unless your risk analysis is based on extensive and statistically valid risk data, your risk analysis will reflect perceived likelihood and impact. In other words, personal perceptions and opinions held by the stakeholders will determine the level of risk. Again, there's absolutely nothing wrong with this, and I don't bring this up to condemn the technique at all. The key point is that project managers, programmers, users, business analysts, architects, and testers typically have different perceptions and thus possibly different opinions on the level of risk for each risk item. By including all these perceptions, we distill the collective wisdom of the team.

However, we do have a strong possibility of disagreements between

stakeholders. So the risk analysis process should include some way of reaching consensus. In the worst case, if we cannot obtain consensus, we should be able to escalate the disagreement to some level of management to resolve. Otherwise, risk levels will be ambiguous and conflicted and thus not useful as guide for risk mitigation activities—including testing.

3.2.5. Controlling the Risks

Part of any management role, including test management, is controlling risks that affect your area of interest. How can we control risks? We have four main options for risk control:

- Mitigation, where we take preventive measures to reduce the likelihood and/or the impact of a risk.
- Contingency, where we have a plan or perhaps multiple plans to reduce the impact if the risk becomes an actuality.
- Transference, where we get another party to accept the consequences of a risk.
- Finally, we can ignore or accept the risk and its consequences.

For any given risk item, selecting one or more of these options creates its own set of benefits and opportunities as well as costs and, potentially, additional risks associated with each option. Done wrong, risk control can make things worse, not better.

Analytical risk-based testing is focused on creating risk mitigation opportunities for the test team, especially for quality risks. Risk-based testing mitigates quality risks through testing throughout the entire lifecycle.

In some cases, there are standards that can apply. We'll look at a couple of risk-related standards shortly in this section.

3.2.6. Project Risks

While much of this section deals with product risks, test managers often identify project risks, and sometimes they have to manage them. Let's discuss this topic now so we can subsequently focus on product risks. A specific list of all possible test-related project risks would be huge, but includes issues like these:

- Test environment and tool readiness

- Test staff availability and qualification
- Low quality of test deliverables
- Too much change in scope or product definition
- Sloppy, ad hoc testing effort

Test-related project risks can often be mitigated or at least one or more contingency plans put in place to respond to the unhappy event if it occurs. A test manager can manage risk to the test effort in a number of ways.

We can accelerate the moment of test involvement and ensure early preparation of testware. By doing this, we can make sure we are ready to start testing when the product is ready. In addition, as mentioned in the Foundation syllabus and elsewhere in this course, early involvement of the test team allows our test analysis, design, and implementation activities to serve as a form of static testing for the project, which can serve to prevent bugs from showing up later during dynamic testing, such as during system test. Detecting an unexpectedly large number of bugs during high-level testing like system test, system integration test, and acceptance test creates a significant risk of project delay. This bug-preventing activity is a key project risk-reducing benefit of testing.

We can make sure that we check out the test environment before test execution starts. This early check can be paired with another risk-mitigation activity, that of testing early versions of the product before formal test execution begins. If we do this in the test environment, we can test the testware, the test environment, the test release and test object installation process, and many other test execution processes in advance before the first day of testing.

We can also define tougher entry criteria to testing. That can be an effective approach *if* the project manager will slip the end date of testing if the start date slips. Often, project managers won't do that, so making it harder to start testing while not changing the end date of testing simply creates more stress and put pressure on the test team.

We can try to institute requirements for testability. For example, getting the user interface design team to change editable fields into noneditable pull-down fields wherever possible—such as on date and time fields—can reduce the size of the potential user input validation test set dramatically and help automation efforts.

To reduce the likelihood of being caught unaware by really bad test objects, a

to help reduce bugs in those test objects, test team members can participate in reviews of earlier project work products, such as requirements specifications. We can also have the test team participate in problem and change management.

Finally, during the test execution effort—hopefully starting with unit testing a perhaps even before, but if not at least from day one of formal testing—we can monitor the project progress and quality. If we see alarming trends developing, we can try to manage them before they turn into end-game disasters.

In [figure 3-1](#), you see the test-related project risks for an Internet appliance project that serves as a recurring case study in this book. These risks were identified in the test plan and steps were taken throughout the project to manage them through mitigation or respond to them through contingency.

Figure 3-1. Test-related project risks example

<i>Risk</i>	<i>Mitigation/Contingency</i>
Unable to staff test team on time.	Reduce scope of test effort in reverse-priority order.
Release management not well-defined, resulting in a test cycle's results being invalidated.	Define a crisp release management process.
Test environment system administration support not available or proficient.	Identify system administration resources with pager/cell availability and appropriate ... skills.
Test environment shared with development.	[TBD]
Buggy deliverables impede testing progress.	Complete unit...testing. Adherence to test entry and exit criteria. Early auditing of vendor test and reliability plans and results.
Test and product scope and definition changes impede testing progress.	Change management or change control board.

Let's review the main project risks identified for testing on this project and the mitigation and contingency plans put in place for them.

We were worried, given the initial aggressive schedules, that we might not be able to staff the test team on time. Our contingency plan was to reduce scope test effort in reverse-priority order.

On some projects, test release management is not well-defined, which can result in a test cycle's results being invalidated. Our mitigation plan was to ensure a well-defined crisp release management process.

We have sometimes had to deal with test environment system administration support that was either unavailable at key times or simply unable to carry out the tasks required. Our mitigation plan was to identify system administration resources with pager and cell phone availability and appropriate UNIX, QNX, and network skills.

As consultants, my associates and I often encounter situations in which test environment are shared with development, which can introduce tremendous delays and unpredictable interruptions into the test execution schedule. In this case, we had not yet determined the best mitigation or contingency plan for this so it was marked "[TBD]".

Of course, buggy deliverables can impede testing progress. In fact, more often than not, the determining factor in test cycle duration for new applications (as opposed to maintenance releases) is the number of bugs in the product and how long it takes to grind them out. We asked for complete unit testing and adherence to test entry and exit criteria as mitigation plans for the software. If the hardware component, we wanted to mitigate this risk through early audit of vendor test and reliability plans and results.

It's also the case that frequent or sizeable test and product scope and definition changes can impede testing progress. As a contingency plan to manage this should it occur, we wanted a change management or change control board to be established.

3.2.7. Two Industry Standards and Their Relation to Risk

You can find an interesting example of how risk management, including quality risk management, plays into the engineering of complex and/or safety-critical systems in the ISO/IEC standard 61508, which is mentioned in the Advanced syllabus. It is designed especially for embedded software that controls systems with safety-related implications, as you can tell from its title: "Functional safety of electrical/electronic/programmable electronic safety-related systems".

The standard focuses on risks. It requires risk analysis. It considers two primary factors to determine the level of risk: likelihood and impact. During a project, the standard directs us to reduce the residual level of risk to a tolerable level, specifically through the application of electrical, electronic, or software improvements to the system.

The standard has an inherent philosophy about risk. It acknowledges that we can't attain a level of zero risk—whether for an entire system or even for a single risk item. It says that we have to build quality, especially safety, in from

the beginning, not try to add it at the end, and thus must take defect-prevent actions like requirements, design, and code reviews.

The standard also insists that we know what constitutes tolerable and intolerable risks and that we take steps to reduce intolerable risks. When those steps are testing steps, we must document them, including a software safety validation plan, software test specification, software test results, software safety validation verification report, and software functional safety report. The standard is concerned with the author-bias problem, which, as you should recall from the Foundation syllabus, is the problem with self-testing, so it calls for tester independence, indeed insisting on it for those performing any safety-related tests. And, because testing is most effective when the system is written to be testable, that's also a requirement.

The standard has a concept of a safety integrity level (SIL), which is based on the likelihood of failure for a particular component or subsystem. The safety integrity level influences a number of risk-related decisions, including the choice of testing and QA techniques.

Some of the techniques are ones I discuss in the companion volume on Advanced Test Analyst, such as the various functional and black-box testing design techniques. Many of the techniques are ones I discuss in the companion volume on Advanced Technical Test Analyst, including probabilistic testing, dynamic analysis, data recording and analysis, performance testing, interface testing, static analysis, and complexity metrics. Additionally, because thorough coverage including during regression testing, is important to reduce the likelihood of missed bugs, the standard mandates the use of applicable automated test tool

Again, depending on the safety integrity level, the standard might require various levels of testing. These levels include module testing, integration testing, hardware-software integration testing, safety requirements testing, and system testing. If a level is required, the standard states that it should be documented and independently verified. In other words, the standard can require auditing outside reviews of testing activities. Continuing in that vein of "guarding the guards", the standard also requires reviews for test cases, test procedures, and test results, along with verification of data integrity under test conditions.

The 61508 standard requires structural testing as a test design technique. So structural coverage is implied, again based on the safety integrity level. Because the desire is to have high confidence in the safety-critical aspects of the system, the standard requires complete requirements coverage not once but multiple times, at multiple levels of testing. Again, the level of test coverage required

depends on the safety integrity level.

Now, this might seem a bit excessive, especially if you come from a very informal world. However, the next time you step between two pieces of metal that can move—e.g., elevator doors—ask yourself how much risk you want to remain in the software that controls that movement.

Let's look at another risk-related testing standard. The United States Federal Aviation Administration provides a standard called DO-178B for avionics systems. In Europe, it's called ED-12B.

Table 3-1. FAA-DO 178B mandated coverage

Criticality	Potential Failure Impact	Required Coverage
Level A: Catastrophic	Software failure can result in a catastrophic failure of the system.	Modified Condition/Decision, Decision, and Statement
Level B: Hazardous/Severe	Software failure can result in a hazardous or severe/major failure of the system.	Decision and Statement
Level C: Major	Software failure can result in a major failure of the system.	Statement
Level D: Minor	Software failure can result in a minor failure of the system.	None
Level E: No effect	Software failure cannot have an effect on the system.	None

The standard assigns a criticality level based on the potential impact of a failure, as shown in [Table 3-1](#). Based on the criticality level, the DO-178B standard requires a certain level of white-box test coverage.

- Criticality level A, or Catastrophic, applies when a software failure can result in a catastrophic failure of the system. For software with such criticality, the standard requires Modified Condition/Decision, Decision, and Statement coverage.
- Criticality level B, or Hazardous and Severe, applies when a software failure can result in a hazardous, severe, or major failure of the system. For software with such criticality, the standard requires Decision and Statement coverage.

coverage.

- Criticality level C, or Major, applies when a software failure can result in a major failure of the system. For software with such criticality, the standard requires only Statement coverage.
- Criticality level D, or Minor, applies when a software failure can result in a minor failure of the system. For software with such criticality, the standard does not require any level of coverage.
- Finally, criticality level E, or No effect, applies when a software failure can have an effect on the system. For software with such criticality, the standard does not require any level of coverage.

This standard makes a certain amount of sense. You should be more concerned about software that affects flight safety, such as rudder and aileron control modules, than you are about software that doesn't, such as video entertainment systems. Of course, lately there has been a trend toward putting all of the software, both critical and noncritical, on a common network in the plane, which introduces enormous potential risks for inadvertent interference and malicious hacking.

However, I consider it dangerous to use a one-dimensional white-box measure as the sole stick to determine how much confidence we should have in a system. Coverage metrics are a measure of confidence, it's true, but we should use multiple coverage metrics, both white-box and black-box.^[1]

[1] You might be tempted to say, "Well, why worry about this? It seems to work for aviation software?" Spend a few moments on the Risks Digest at www.risks.org and peruse some of the software-related aviation near misses. You might feel less sanguine. There is also a discussion of the Boeing 787 design issue that relates to the use of a single network for all onboard systems, both safety critical and non-safety critical.

By the way, if you found this material a bit confusing, note that the white-box coverage metrics used in this standard were discussed in the Foundation syllabus in [chapter 4](#). If you don't remember these coverage metrics, you should go back and review that material in that chapter of the Foundation syllabus.

3.2.8. Risk-Based Testing and Failure Mode and Effect Analysis

Exercise 1

Review the HELLOCARMS System Requirements Document, shown in [appendix B](#).

focusing specifically on identifying test-related project risks. You can also look other project risks if you'd like.

Document the test-related project risks, along with the control option(s) you would choose to take.

If you are working in a group setting, discuss.

3.2.9. Risk-Based Testing and Failure Mode and Effect Analysis

Exercise 1 Debrief

Here are the test-related and other project risks I identified in my review of the HELLOCARMS System Requirements Document. For the test-related project risks, I have identified control activities.

3.2.9.1. Test-Related Project Risks

Here are some test-related project risks inherent in the requirements, together with risk-control options for each one.

- 1.** Unable to secure a true replica of the production environment (see section 000) for testing, leading to restrictions/blockages in testing.
 - Mitigation: Identify quality risks that require replica of production environment for testing purposes, report to project management.
 - Mitigation: As test environment is assembled, identify tests that cannot be run due to discrepancies between test and production environment report to project management.
 - Mitigation: Explore options for outsourcing performance and reliability testing, because the application architecture is common.
 - Transfer: If test environment discrepancies remain during test execution, escalate the test coverage holes inherent in these limitations to the project team.
 - Contingency: If certain interoperating systems will not be available (e.g. Scoring Mainframe), arrange for test harness that will allow simulation of these interactions.
- 2.** Unable to acquire and deploy adequate and workable performance and reliability testing tools (see sections 020 and 040).

- Mitigation: Determine budgetary and other constraints during test planning and use these constraints during tool evaluation.
 - Mitigation: Immediately start tool evaluation in parallel with test planning.
 - Transfer: If performance and reliability cannot be tested due to lack of tools, escalate the test coverage holes inherent in these limitations to the project team.
- 3.** Unable to run tests, including basic functionality and interoperability tests start of formal testing due to insufficient quality (including uninstallability the test releases).
- Mitigation: Use of an iterative lifecycle model with relatively short iterations, focused on achieving basic functionality and interoperability the first iteration.
 - Mitigation: Continuous integration, nightly builds, and automated smoke tests by the development team.
 - Mitigation: Test installations of test releases into the test environment regularly during test implementation (i.e., prior to test execution) to evaluate the test environment and the installability/testability of the test releases prior to the start of test execution.
 - Mitigation: Design tests to cover both end-to-end, complex workflows and relatively simple (e.g., screen-by-screen) capabilities to make tests robust in the face of poor-quality software.
 - Contingency/transfer: If tests are blocked on entry, triage test cases based on risk priority and escalate reduction in coverage to the project team.
- 4.** Changes in product in second and later iterations require extensive retest invalidate test results, and require test updates.

- Mitigation: Use maintainable automated tests to regression-test existing functionality as new iterations occur, adding new automated tests to the regression suite for each iteration (note that this mitigation plan has automated test design implications as well as test team staffing implications).

- Mitigation: For manual testing, use testers with domain and testing expertise, limit extent of test case documentation to reduce test update effort, and maintain a significant (10-20%) mix of exploratory test execution with test charters used to document exploratory coverage a execution.
 - Contingency/transfer: Where regression testing is not possible, triage test effort based on risk and escalate reduction in coverage to the project team.
- 5. Blank spots in requirements specification (e.g., section 020) completed too late to allow for adequate test preparation.**
- Mitigation: Work backwards from start of first test execution cycle to identify date by which all key specification areas must be complete to avoid impact to test, work with project management to ensure completion by that time.
 - Mitigation: Make reasonable assumptions about the requirements that would apply where not currently specified, define lightweight tests, allowing for the possibility of test updates during test execution.
 - Mitigation: Appoint one test team member to monitor requirements changes/completions so the appropriate test cases can be identified for changes/maintenance.
 - Transfer: Where insufficient information is provided to define any test whatsoever, escalate holes in coverage to the project team.
- 6. Misconfigurations of complex test environment result in a large number of false positives (i.e., incident reports related to the environment and not the software itself).**
- Mitigation: Ensure adequate skills and availability in the assigned system administration support staff.
 - Mitigation: Test installations of test releases into the test environment regularly during test implementation (i.e., prior to test execution) to evaluate the test environment and the installability/testability of the test releases prior to the start of test execution [note: repeat of mitigation for item 3].
 - Contingency: Monitor environment-related false positives during test

execution, increase test system administration skills through contractors/consultants if needed.

- 7.** Gaps in test coverage result in an unacceptable defect detection percentage (i.e., less than 95%) with a high rate of field failures.
 - Mitigation: Use quality risk analysis to guide the testing.
 - Contingency: Use a significant (10-20%) mix of exploratory test execution (including in areas not considered key risks), use exploratory test results to check for any gaps in the risk analysis and thus the test cases, and refactor the tests based on any gaps discovered.
 - Transfer: Escalate gaps in testing related to key (high likelihood and/or impact) risks to management.
- 8.** Delays in the start of test execution without corresponding delays in the end date result in compression of test periods.
 - Mitigation: Adhere to iterative lifecycle model, reduce number of iterations rather than compressing iteration times through reduced test scope.
 - Contingency/transfer: Where test compression does occur, triage test effort based on risk and escalate reduction in coverage to the project team.
- 9.** Budget or staffing constraints make it impossible to assemble fully suitable test team and/or environment.
 - Mitigation: Explore outsource possibilities to reduce costs (e.g., performance, reliability, and compatibility testing).
 - Mitigation: Structure test effort so that discrete, separable test areas can be stripped away while maintaining an ability to do some useful testing in a realistic time frame.
 - Contingency/transfer: Where test compression does occur, triage test effort based on risk and escalate reduction in coverage to the project team.
- 10.** Use of the test environment for debugging causes unpredictable delays, loss of test time/effort, and ultimately test compression for the test team.

- Mitigation: Work with project management to ensure adequate debugging environments for the development team (which would also have the positive side effect of lowering the risk of installability problems if the software were first installed in the development environment).
- Contingency/transfer: Where test compression does occur, triage test effort based on risk and escalate reduction in coverage to the project team.

3.2.9.2. Other Project Risks

Here are some non-test-related project risks inherent in the requirements.

1. Given the lack of clarity around security requirements (see 010-040-010) there's a strong chance that the necessary infrastructure won't be in place needed to support this project's schedule.
2. A higher-than-expected number of escalations to Senior Telephone Banks (see sections 001 and 004) undermines the ability of the system to service the expected number of customers.
3. Insufficient availability of developers and other Globobank employees assigned to other interoperating systems (e.g., Scoring Mainframe, LoDoP etc.) to resolve cross-system problems, leading to delays in defect resolution and production environment setup, delaying first release.
4. Changes/upgrades to interoperating systems that are not coordinated with this release.

3.2.10. Risk Identification and Assessment Techniques

Various techniques exist for performing quality risk identification and assessment. These techniques range from informal to semiformal to formal.

You can think of risk identification and assessment as a structured form of project and product review. In a requirements review, we focus on what the system should do. In quality risk identification and assessment sessions, we focus on what the system might do that it should not. Thus, we can see quality risk identification and assessment as the mirror image of the requirements, the design, and the implementation.

As with any review, as the level of formality increases, so does the cost, the defect removal effectiveness, and the extent of documentation associated with

You'll want to choose the technique you use based on constraints and needs for your project. For example, if you are working on a short project with a very tight budget, adopting a formal technique with extensive documentation doesn't make much sense.

Let's review the various techniques for quality risk identification and assessment, from informal to formal, and then some ways in which you can organize the sessions themselves.

In many successful implementations of projects, we use informal methods for risk-based testing. These methods can work just fine. In particular, it's a good way to start learning about and practicing risk-based testing because excessive formality and paperwork can create barriers to successful adoption of risk-based testing.

In informal techniques, we rely primarily on history, stakeholder domain and technical experience, and checklists of risk category to guide us through the process. These informal approaches are easy to put in place and to carry out. They are lightweight in terms of both documentation and time commitment. They are flexible from one project to the next because the amount of documented process is minimal.

However, because we rely so much on stakeholder experience, these techniques are participant dependent. The wrong set of participants means a relatively poor set of risk items and assessed risk levels. Because we follow a checklist, if the checklist has gaps, so does our risk analysis. Because of the relatively high level at which risk items are specified, they can be imprecise both in terms of the items and the level of risk associated with them.

That said, these informal techniques are a great way to get started doing risk-based testing. If it turns out that a more precise or formal technique is needed, the informal quality risk analysis can be expanded and formalized for subsequent projects. Even experienced users of risk-based testing should consider informal techniques for low-risk or agile projects. You should avoid using informal techniques on safety-critical or regulated projects due to the lack of precision and the tendency toward gaps.

3.2.11. Categories of Quality Risks

I mentioned that informal risk-based testing tends to rely on a checklist to identify risk items. What are the categories of risks that we would look for? In part, that depends on the level of testing we are considering. Let's start with the early levels of testing, unit or component testing. In the following lists, I'm going to focus on the types of risks that are most common at these levels.

to pose these checklist risk categories in the form of questions, to help stimulate your thinking about what might go wrong.

- States: Does the unit handle state-related behaviors properly? Do transitions from one state to another occur when the appropriate events occur? Are the correct actions triggered? Are the correct events associated with each input?
- Transactions: Can the unit handle the transactions it should handle, correctly, without any undesirable side effects?
- Code Coverage: What statements, branches, conditions, complex conditions, loops, and other paths through the code might result in failures?
- Data Flow Coverage: What flows of data into or out of the unit—whether through parameters, objects, global variables, or persistent data structures like files or database tables—might result in immediate or delayed failures including corrupted persistent data (the worst kind of corrupted data).
- Functionality: Is the functionality provided to the rest of the system by the component incorrect, or might it have invalid side effects?
- User Interface: If this component interacts with the user, might users have problems understanding prompts and messages, deciding what to do next, or feeling comfortable with color schemes and graphics?
- Mechanical Life: For hardware components, might this component wear out or fail after repeated motion or use?
- Signal Quality: For hardware components, are the signals correct and in the correct form?

As we move into integration testing, additional risks arise, many in the following areas:

- Component or Subsystem Interfaces: Are the interfaces between components well defined? What problems might arise in direct and indirect interaction between components?
- Functionality: Again, what problems might exist in terms of actions and side effects, particularly as a result of component interaction?
- Capacity and Volume: Are the static data spaces such as memory and disk space sufficient to hold the information needed? Are the dynamic volume conduits such as networks going to provide sufficient bandwidth?

- Error/Disaster Handling and Recovery: Will the integrated component respond correctly under typical and extremely adverse conditions? Can the component recover to normal functionality after such a condition?
- Data Quality: Can the system store, load, modify, archive, and manipulate data reliably, without corruption or loss of data?
- Performance: What problems might exist in terms of response time, efficient resource utilization, and the like?
- User Interface: Again, for this integration collection of components, if a user interface is involved, might users have problems understanding prompts and messages, deciding what to do next, or feeling comfortable with color schemes and graphics?

Similar issues apply for system integration testing, but we would be concerned with the integration of systems, not components.

Finally, what kinds of risk might we consider for system and user acceptance testing?

- Functionality: Again, we need to consider functionality problems. At these levels, the issues we are concerned with are systemic. Do end-to-end functions work properly? Are deep levels of functionality and combinations of related functions working?
- User Interface and Usability: In terms of the whole system interface to the user, are we consistent? Can the user understand the interface? Do we mislead or distract the user at any point? Trap the user in dead-end interfaces?
- States and Transactions: Overall, does the system handle various states correctly? Considering states the user or objects acted on by the system might be in, are there potential problems here?
- Data Quality: Considering the entire set of data that the system uses—including data it might share with other systems—can the system store, load, modify, archive, and manipulate that data reliably, without corrupting or losing it?
- Operations: Complex systems often require administration. Databases, networks, and servers are examples. Operations these administrators perform can include essential maintenance tasks. For example, might there

be problems with backing up and restoring files or tables? Can you migrate the system from one version or type of database server or middleware to another? Can storage, memory, or processor capacity be added?

- Performance, Stress, Capacity, and Volume: Are there potential issues with response time? With behavior under combined conditions of heavy load and low resources? Insufficient static space? Insufficient dynamic capacity and bandwidth?
- Reliability, Availability, and Stability: Will the system fail under normal, exceptional, or heavy load conditions? Might the system be unavailable when needed? Might it prove unstable with certain functions?
- Installation, Cutover, Setup, and Configuration: What installation, data migration, application migration, configuration, or initial conditions might cause problems?
- Error and Disaster Handling and Recovery: Will the system respond correctly under typical and extremely adverse conditions? Can it recover to normal functionality after such a condition? Might its response to such conditions create consequent conditions that negatively affect interoperating or cohabiting applications?
- Date and Time Handling: Might certain date- or time-triggered events fail? Do related functions that use dates or times work properly together? Could situations like leap years or daylight saving time transitions cause problems? What about time zones?
- Localization: In terms of the various languages we need to support, will some of those character sets or translated messages cause problems? Might currency differences cause problems?
- Networked and Distributed: Do latency, bandwidth, or other factors relate to the networking or distribution of processing and storage cause potential problems?
- Compatibility: Might the system be incompatible with various environments it has to work in? Might the system be incompatible with interoperating or cohabiting applications in some of the supported environments?
- Standards: What standards apply to our system, and might the system violate some of those standards?

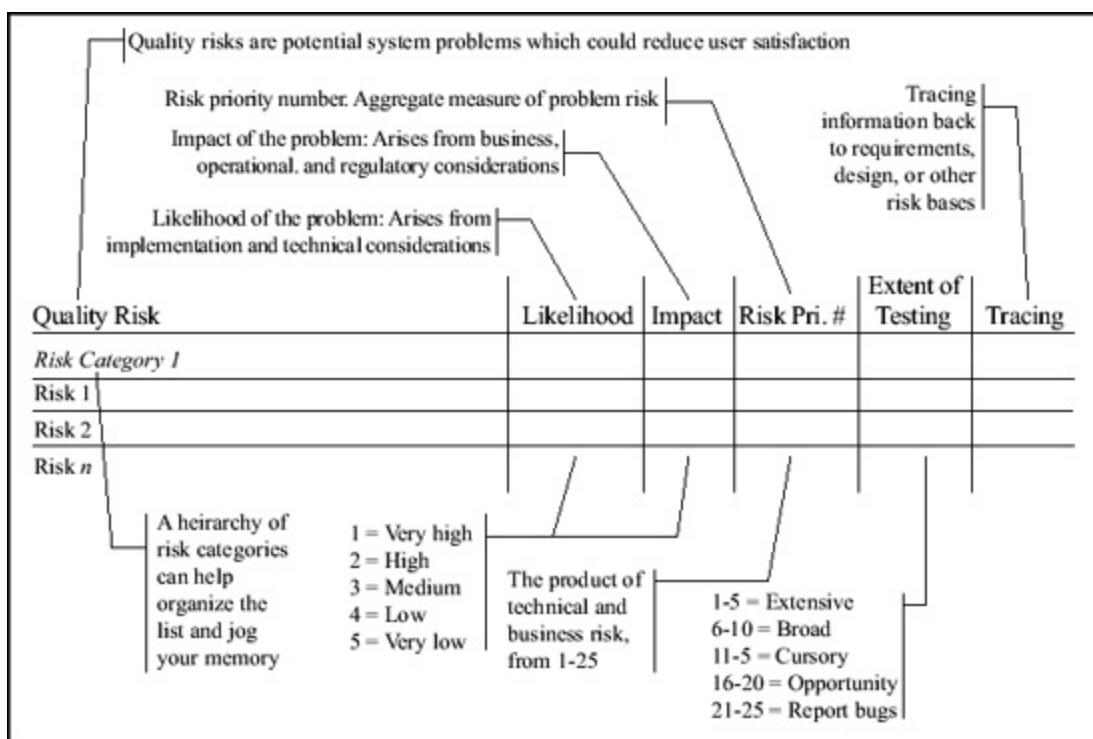
- Security: Is it possible for users without proper permission to access functions or data they should not? Are users with proper permission potentially denied access? Is data encrypted when it should be? Can security attacks bypass various access controls?
- Environment: For hardware systems, might normal or exceptional operating environments cause failures? Will humidity, dust, or heat cause failures, either permanent or intermittent?
- Power: Are there problems with power consumption for hardware systems? Do normal variations in the quality of the power supplied cause problems? Is battery life sufficient?
- Shock, Vibration, and Drop: For hardware systems, might foreseeable physical shocks, background vibrations, or routine bumps and drops cause failures?
- Documentation and Packaging: Is the documentation incorrect, insufficient, or unhelpful? Is the packaging sufficient?
- Maintainability: Can we upgrade the system? Apply patches? Remove or add features from the installation media?

There are certainly other potential risk categories, but this list forms a good starting point. You'll want to customize this list to your particular systems if you use it.

3.2.12. Documenting Quality Risks

In [figure 3-2](#), you see a template that can be used to capture the information you identify in quality risk analysis. In this template, you start by identifying the risk items, using the categories just discussed as a framework. Next, for each risk item, you assess its level of risk in terms of the factors of likelihood and impact. You then use these two ratings to determine the overall priority of testing and the extent of testing. Finally, if the risks arise from specific requirements or design specification elements, you establish traceability back to these items. Let's look at these activities and how they generate information to populate this template.

Figure 3-2. A template for capturing quality risk information



First, remember that quality risks are potential system problems that could reduce user satisfaction. We can use the risk categories to organize the list and to jog people's memory about risk items to include. Working with the stakeholders, we identify one or more quality risk item for each category and populate the template.

Having identified the risks, we can now go through the list of risk items and assess the level of risk because we can see the risk items in relation to each other. An informal technique typically uses main factors for assessing risk.

The first is the likelihood of the problem, which is determined mostly by technical considerations. I sometimes call this "technical risk" to remind me of that fact. The second is the impact of the problem, which is determined mostly by business or operational considerations. I sometimes call this "business risk" to remind me of that fact.

Both likelihood and impact can be rated on an ordinal scale. A three-point ordinal scale is high, medium, and low. I prefer to use a five-point scale, from very high to very low.

Given the likelihood and impact, we can calculate a single, aggregate measure of risk for the quality risk item. A generic term for this measure of risk is *risk priority number*. One way to do this is to use a formula to calculate the risk priority number from the likelihood and impact. First, translate the ordinal scale into a numerical scale, as in this example:

1 = Very high

2 = High

3 = Medium

4 = Low

5 = Very low

You can then calculate the risk priority number as the product of the two numbers. We'll revisit this issue in a later section of this chapter because this is just one of many ways to determine the risk priority.

The risk priority number can be used to sequence the tests. To allocate test effort, I determine the extent of testing. [Figure 3-2](#) shows one way to do this, dividing the risk priority number into five groups and using those to determine test effort:

1-5 = Extensive

6-10 = Broad

11-15 = Cursory

16-20 = Opportunity

21-25 = Report bugs only

We'll return to the matter of variations in the way to accomplish this later in this section.

As noted before, while you go through the quality risk analysis process, you are likely to generate various useful by-products. These include implementation assumptions that you and the stakeholders made about the system in assessing likelihood. You'll want to validate these, and they might prove to be useful suggestions. The by-products also include project risks that you discovered, which the project manager can address. Perhaps most importantly, the by-products include problems with the requirements, design, or other input documents. We can now avoid having these problems turn into actual system defects. Notice that all three enable the bug-preventive role of testing discussed earlier in this book.

In [Figure 3-3](#), you see an example of an informal quality risk analysis. We have

used six quality categories for our framework:

- Functionality
- Localization
- Usability and user interface
- Reliability
- Performance
- Supportability

These are the standard quality categories used by some groups at Hewlett Packard.

I've provided one or two example quality risk items for each category. Of course for a typical product there would be more like 100 to 500 total quality risks—perhaps even more for particularly complex products.

3.2.13. Quality Risk Analysis Using ISO 9126

We can increase the structure of an informal quality risk analysis—formalize it slightly, if you will—by using the ISO 9126 standard as the quality characteristics framework instead of the rather lengthy and unstructured list of quality risk categories given on the previous pages.

Figure 3-3. Informal quality risk analysis example

Quality Risk	Tech. Risk	Bus. Risk	Risk Priority	Extent of Testing
<i>Functionality</i>				
User login and authentication fails	2	1	2	Extensive
User can't view transactions in accounts	3	1	3	Extensive
<i>Localization</i>				
Spanish language translation doesn't fit fields	2	3	6	Broad
Spanish language messages wrong, insulting, etc.	4	3	12	Cursory
<i>Usability and User Interface</i>				
User gets "stuck" in screen flows, esp. on error	4	2	8	Broad
User finds interface too complex	4	1	4	Extensive
<i>Reliability</i>				
Less than 95% availability from 4:00 AM to 2:00 AM.	4	2	8	Broad
<i>Performance</i>				
System responses >0.5 seconds under full load	5	2	10	Broad
<i>Supportability</i>				
Patches and releases won't install correctly	5	4	20	Opportunity
Patches and releases require extensive downtime	5	3	15	Cursory

This standard has some strengths. The ISO 9126 standard provides a pre-defined and thorough framework. The standard itself—that is, the entire set of documents that the standard comprises—provides a predefined way to tailor it to your organization. If you use this across all projects, you will have a common basis for your quality risk analyses and thus your test coverage. Consistency in testing across projects provides comparability of results.

The use of ISO 9126 in risk analysis has its weaknesses too. For one thing, if you are not careful tailoring the quality characteristics, you could find that you are potentially over-broad in your analysis; that makes you less efficient. For another thing, applying the standard to all projects, big and small, complex or simple, could prove over-regimented and heavyweight from a process point of view.

I would suggest that you consider the use of ISO 9126 structure for risk analysis whenever a bit more formality and structure is needed, or if you are working on a project where standards compliance matters. I would avoid its use on atypical projects or projects where too much structure, process overhead, or paperwork is likely to cause a problem, relying instead on the lightest-weight informal process possible in such cases.

To refresh your memory on the ISO 9126 standard, here are the six quality characteristics:

- Functionality, which has the subcharacteristics of suitability, accuracy, interoperability, security, and compliance
- Reliability, which has the subcharacteristics of maturity (robustness), fault tolerance, recoverability, and compliance
- Usability, which has the subcharacteristics of understandability, learnability, operability, attractiveness, and compliance
- Efficiency, which has the subcharacteristics of time behavior, resource utilization, and compliance
- Maintainability, which has the subcharacteristics of analyzability, changeability, stability, testability, and compliance
- Portability, which has the subcharacteristics of adaptability, installability, coexistence, replaceability, and compliance

You should remember, too, that in the ISTQB taxonomy of black-box or behavioral tests, those related to functionality and its subcharacteristics are functional tests, while those related to reliability, usability, efficiency, maintainability, and portability and their subcharacteristics are nonfunctional tests.

3.2.14. Quality Risk Analysis Using Cost of Exposure

Another form of quality risk analysis is referred to as *cost of exposure*, a name derived from the financial and insurance world. The cost of exposure—or the expected payout in insurance parlance—is the likelihood of a loss times the average cost of such a loss. Across a large enough sample of risks for a long enough period, we would expect the total amount lost to tend toward the total costs of exposure for all the risks.

So, for each risk, we should estimate, evaluate, and balance the costs of testing versus not testing. If the cost of testing were below the cost of exposure for a risk, we would expect testing to save us money on that particular risk. If the cost of testing were above the cost of exposure for a risk, we would expect testing to be a smart way to reduce costs of that risk.

This is obviously a very judicious and balanced approach to testing. Where there's a business case, we test, where there's not, we don't. What could be more practical? Further, in the insurance and financial worlds, you're likely to find that stakeholders relate easily and well to this approach.

That said, it has some problems. In order to use this approach with any degree of confidence, we need enough data to make reasonable estimates of likelihood and cost. Furthermore, this approach uses monetary considerations exclusively to decide on the extent and sequence of testing. For some risks, the primary downsides are nonmonetary, or at least difficult to quantify, such as lost business and damage to company image.

If I were working on a project in a financial or actuarial world, and had access to the data, I'd probably lean toward this approach. The accessibility of the technique to the other participants in the risk analysis process is quite valuable. However, avoid this technique on safety- or mission-critical projects. There's no way to account properly for the risk of injuring people or the risk of catastrophic impact to the business.

3.2.15. Quality Risk Analysis Using Hazard Analysis

Another risk analysis technique that you can use is called hazard analysis. Like cost of exposure, it fits with certain fields quite well and doesn't fit many other fields.

A hazard is the thing that creates a risk. For example, a wet bathroom floor creates the risk of a broken limb due to a slip and fall. In hazard analysis, we work to understand the hazards that create risks for our systems. This has implications not only for testing but also for upstream activities that can reduce the hazards and thus reduce the likelihood of the risks.

As you might imagine, this is a very exact, cautious, and systematic technique. Having identified a risk, we then must ask ourselves how that risk comes to be and what we might do about the hazards that create the risk. In situations in which we can't afford to miss anything, this technique makes sense.

However, in complex systems there could be dozens or hundreds or thousands of hazards that interact to create risks. Many of the hazards might be beyond our ability to predict. So, hazard analysis is overwhelmed by excessive complexity and in fact might lead us to think the risks are fewer than they really are. That's bad.

I would consider using this technique on medical or embedded systems projects. However, on unpredictable, rapidly evolving, or highly complex projects, I'd avoid it.

3.2.16. Determining the Aggregate Risk Priority

We are going to cover one more approach for risk analysis in a moment, but I

want to return to this issue of using risk factors to derive an aggregate risk priority using a formula. You'll recall this was the technique shown earlier where we multiplied the likelihood and impact to determine the risk priority number. This is also implicit in the cost of exposure technique, where the cost of exposure for any given risk is the product of the likelihood and the average cost of a loss associated with that risk.

Some people prefer to use addition rather than multiplication. For example, R. Craig uses addition of the likelihood and impact.^[5] This results in a more compressed and less sparse scale of risks. To see that, take a moment to construct two tables. Use likelihood and impact ranging from 1-5 for each, and then populate the tables showing all possible risk priority number calculations for all combinations of likelihood and impact. The tables should each have 25 cells. In the case of addition, the risk priority numbers range from 2-10, while in the case of multiplication, the risk priority numbers range from 1-25.

[5] See his book, *Systematic Software Testing*.

It's also possible to construct sophisticated formulas for the risk priority number, some of which might use subfactors for each major factor. For example, certain test management tools such as the newer versions of Quality Center support this. In these formulas, we can weight some of the factors so that they account for more points in the total risk priority score than others.

In addition to calculating a risk priority number for sequencing of tests, we also need to use risk factors to allocate test effort. We can derive the extent of testing using these factors in a couple ways. We could try to use another formula. For example, we could take the risk priority number and multiply it by some given number of hours for design and implementation and some other number of hours for test execution. Alternatively, we could use a qualitatively method where we try to match the extent of testing with the risk priority number, allowing some variation according to tester judgment.

If you do choose to use formulas, make sure you tune them based on historical data. Or, if you are time-boxed in your testing, you can use formulas based on risk priority numbers to distribute the test effort proportionally based on risks.

Some people prefer to use a table rather than a formula to derive the aggregate risk priority from the factors. [Table 3-2](#) shows an example of such a table.

Table 3-2. Using a table for risk priority

	Impact
Low	Low
Low	Medium
Medium	Low
Medium	Medium
High	Low
High	Medium
High	High

Very high	High	Medium	Low	Very low		
Likelihood	Very high	Very high	Very high	High	High	Medium
High	Very high	High	High	Medium	Medium	
Medium	High	High	Medium	Low	Low	
Low	High	Medium	Low	Low	Very low	
Very low	Medium	Medium	Low	Very low	Very low	

First you assess the likelihood and impact as before. You then use the table to select the aggregate risk priority for the risk item based on likelihood and impact scores. Notice that the table looks quite different than the two you constructed earlier. Now, experiment with different mappings of risk priority numbers to risk priority ratings—ranging from very high to very low—to see whether the addition or multiplication method more closely corresponds to this table.

As with the formulas discussed a moment ago, you should tune the table based on historical data. Also, you should incorporate flexibility into this approach by allowing deviation from the aggregate risk priority value in the table based on stakeholder judgment for each individual risk.

In [Table 3-3](#), you see that not only can we derive the aggregate risk rating from a table, we can do something similar for the extent of testing. Based on the risk priority rating, we can now use a table like [Table 3-3](#) to allocate testing effort. You might want to take a moment to study this table.

Table 3-3. Using a table for extent of testing

Risk Priority	Extent of Testing	Comments
Very low	None	Only report bugs observed for this risk item if discovered during other tests.
Low	Opportunistic	Leverage other tests to explore the risk item with minimal effort and only if the opportunity presents itself.
Medium	Cursory	Run a small number of tests that sample the most interesting conditions for the risk item.
High	Broad	Run a medium number of tests that exercise many different interesting conditions for the risk item.
Very high	Extensive	Run a large number of tests that exercise many combinations and variations of interesting conditions.

3.2.17. Stakeholder Involvement

On a few occasions in this section so far, I've mentioned the importance of stakeholder involvement. In the last sections, we've looked at various techniques for risk identification and analysis. However, the involvement of the right participants is just as important, and probably more important, than the choice of technique. The ideal technique without adequate stakeholder involvement will usually provide little or no valuable input, while a less-than-ideal technique, actively supported and participated in by all stakeholder groups, will almost always produce useful information and guidance for testing.

What is most critical is that we have a cross-functional team representing all of the stakeholders who have an interest in testing and quality. This means that we involve at least two general stakeholder groups. One is made up of those who understand the needs and interests of the customers and/or users. The other includes those who have insight into the technical details of the system. We can involve business stakeholders, project funders, and others as well. Through the proper participant mix, a good risk-based testing process gathers information, builds consensus around what not to test, what to test, the order in which to test, and the way to allocate test effort.

I cannot overstate the value of this stakeholder involvement. Lack of stakeholder involvement leads to at least two major dysfunctions in the risk identification and analysis. First, there is no consensus on priority or effort allocation. This means that people will second-guess your testing after the fact. Second, you will find—either during test execution or worse yet after delivery—that there are many gaps in the identified risks, or errors in the assessment of the level of risk due to the limited perspectives involved in the process.

While we should always try to include a complete set of stakeholders, often not all stakeholders can participate or would be willing to do so. In such cases, some stakeholders may act as surrogates for other stakeholders. For example, in mass-market software development, the marketing team might ask a small sample of potential customers to help identify potential defects that would affect their use of the software most heavily. In this case, the sample of potential customers serves as a surrogate for the entire eventual customer base. As another example, business analysts on IT projects can sometimes represent the users rather than involving users in potentially distressing risk analysis sessions where we have conversations about what could go wrong and how bad it would be.

3.2.18. Risk-Based Testing and Failure Mode and Effect Analysis

Exercise 2

Read the HELLOCARMS System Requirements Document, a hypothetical project which nonetheless is derived from a real project that RBCS helped to test.

If working in a classroom, break into groups of three to five. If you are working alone, you'll need to do this exercise yourself. Perform a quality risk analysis on this project. To help manage the time required for this exercise, identify and assess risks for *functional* quality characteristics only. Use the template on the next page.

To keep the time spent reasonable, I suggest 30 minutes identifying quality risks, then 15 minutes assessing the level of each risk.

If you are working in a classroom, once each group has finished its analysis, discuss the results.

Table 3-4. Functional quality risk analysis template

No.	Quality Risk	Likelihood	Impact	Risk Pri.#	Extent of Testing	Traceability
1.1.000	Functionality: Suitability					
1.1.001	[Functional risks related to suitability go in this section.]					
1.2.000	Functionality: Accuracy					
1.2.001	[Functional risks related to accuracy go in this section.]					
1.3.000	Functionality: Interoperability					
1.3.001	[Functional risks related to interoperability go in this section.]					
1.4.000	Functionality: Security					
1.4.001	[Functional risks related to security go in this section.]					
1.5.000	Functionality: Compliance					

1.5.001	[Functional risks related to compliance go in this section.]				
---------	--	--	--	--	--

3.2.19. Risk-Based Testing and Failure Mode and Effect Analysis

Exercise 2 Debrief

You can see my solution to the exercise in [Table 3-5](#). Immediately after that table are two lists of by-products. One is the list of project risks discovered during the analysis. The other is the list of requirements document defects discovered during the analysis.

As a first pass for this quality risk analysis, I went through each functional requirement and identified one or more risk items for it. I assumed that the priority of the requirement was a good surrogate for the impact of the risk, so used that. Even using all of these shortcuts, it took me about an hour to get through this.

Table 3-5. Functional quality risk analysis for HELLOCARMS

No.	Quality Risk	Likelihood	Impact	Risk Pri.#	Extent of Testing	Trace
1.1.000	Functionality: Suitability					
1.1.001	Reject all applications for home equity loans.	5	1	5	Extensive	010-010-010
1.1.002	Reject all applications for home equity lines of credit.	5	1	5	Extensive	010-010-010
1.1.003	Reject all applications for reverse mortgages.	5	1	5	Extensive	010-010-010
1.1.004	Fail to properly process some home equity applications.	3	1	3	Extensive	010-010-190
1.1.005	Fail to properly process some home equity line of credit applications.	3	2	6	Broad	010-010-200
1.1.006	Fail to properly process	3	3	9	Broad	010-

	some home equity reverse mortgage applications.					010- 210
1.1.007	Fail to properly process some combined products (e.g., home equity and credit cards).	3	4	12	Cursory	010- 010- 220
1.1.008	Fail to properly process some original mortgage applications.	3	5	15	Cursory	010- 010- 230
1.1.009	Fail to properly process some preapproved applications.	3	4	12	Cursory	010- 010- 240
1.1.010	Scripts not available for all fields and screens.	4	2	8	Broad	010- 010- 020
1.1.011	Customer source data not collected.	3	2	6	Broad	010- 010- 030
1.1.012	Customer source data categories not well defined.	2	2	4	Extensive	010- 010- 030
1.1.013	Accepts invalid data at input fields.	1	1	1	Extensive	010- 010- 040
1.1.014	Existing trade lines not displayed and/or processed properly.	3	1	3	Extensive	010- 010- 050 010- 010- 100
1.1.015	Trade line payoff details not passed to LoDoPS.	2	1	2	Extensive	010- 010- 050 010- 010- 100
1.1.016	Loan to be paid off	4	3	12	Cursory	010-

	included in debt-to-income calculations.					010-110
1.1.017	Cannot resume incomplete/interrupted applications.	3	2	6	Broad	010-010-060
1.1.018	Applicant not asked about existing relationship.	5	2	10	Broad	010-010-070
1.1.019	Applicant existing relationship not passed to GLADS.	3	2	6	Board	010-010-070
1.1.020	Loan status information lost after initiation.	3	2	6	Broad	010-010-080
1.1.021	Cannot abort or abandon an application cleanly (i.e., must close browser).	4	3	12	Cursory	010-010-090
1.1.022	Cannot retrieve existing application by customer ID.	4	4	16	Opportunity	010-010-120
1.1.023	Loans over \$500, 000 not transferred for approval.	4	1	4	Extensive	010-010-130
1.1.024	Loans over \$500, 000 automatically denied.	3	1	3	Extensive	010-010-130
1.1.025	Property valuation over \$1, 000,000 not transferred for approval.	4	2	8	Broad	010-010-140
1.1.026	Property valuation over \$1, 000,000 automatically denied.	3	2	6	Extensive	010-010-140
1.1.027	Inbound telemarketing operations fail in supported region.	1	2	2	Extensive	010-010-150
1.1.028	Outbound	1	2	2	Extensive	010-

	telemarketing operations fail in supported region.					010-150
1.1.029	Branding for brokers and other business partners not supported.	4	2	8	Broad	010-010-160
1.1.030	Untrained users (e.g., end customers) cannot enter applications via Internet.	4	3	12	Cursory	010-010-170
1.1.031	Product operations for retail bank branches not supported.	4	4	16	Opportunity	010-010-180
1.1.032	Flexible pricing schemes not supported.	4	5	20	Opportunity	010-010-250
1.2.000	<i>Functionality: Accuracy</i>					
1.2.001	Customer presented with products for which they are ineligible.	3	1	3	Extensive	010-020-010
1.2.002	Customer not presented with products for which they are eligible.	4	1	4	Extensive	010-020-010
1.2.003	Application decisioning inconsistent with Globobank credit policies.	3	1	3	Extensive	010-020-010
1.2.004	Risk-based pricing miscalculated based on credit score, loan to value, and debt to income.	4	1	4	Extensive	010-020-020
1.2.005	New loan payment not included in credit scoring.	3	2	6	Broad	010-020-030
1.2.006	Pricing add-ons not calculated correctly.	3	3	9	Broad	010-020-040
1.2.007	Government retirement	3	1	3	Extensive	010-

	income not handled properly.					020-050
1.2.008	Duration of additional income not captured.	4	3	12	Cursory	010-020-060
1.3.000	<i>Functionality: Interoperability</i>					
1.3.001	Can't pull information from GloboRainBQW into HELLOCARMS.	2	2	4	Extensive	010-030-010
1.3.002	HELOCARMS and Scoring Mainframe reject joint applications.	3	1	3	Extensive	010-030-020
1.3.003	HELOCARMS and Scoring Mainframe cannot handle/resolve duplication of information on joint applications.	1	2	2	Extensive	010-030-030
1.3.004	HELOCARMS trade line communication to LoDoPS fails.	3	1	3	Extensive	010-030-040 010-030-070
1.3.005	Loan status information from LoDoPS to HELLOCARMS lost or corrupted.	5	2	10	Broad	010-030-050 010-030-140
1.3.006	HELOCARMS can't continue if the Scoring Mainframe indicates an undischarged bankruptcy or foreclosure.	4	3	12	Cursory	010-030-060
1.3.007	HELOCARMS communication of	5	1	5	Extensive	010-030-

	government retirement income to LoDoPS fails.					080
1.3.008	HELLOCARMS application information not passed to Scoring Mainframe properly.	3	1	3	Extensive	010-030-090
1.3.009	HELLOCARMS does not receive information from Scoring Mainframe properly.	3	1	3	Extensive	010-030-100
1.3.010	Decisioning requests not queued for Scoring Mainframe as needed.	4	2	8	Broad	010-030-110
1.3.011	Tentatively approved, customer-accepted loans not passed to LoDoPS.	5	2	10	Broad	010-030-120
1.3.012	Declined applications not passed to LoDoPS.	5	2	10	Broad	010-030-130
1.3.013	Changes made in loan information in LoDoPS not propagated back to HELLOCARMS.	5	2	10	Broad	010-030-140
1.3.014	Computer-telephony integration not supported.	3	5	15	Cursory	010-030-150
1.3.015	Applicant existing relationship not passed to GLADS.	4	3	12	Broad	010-010-070
1.4.000	<i>Functionality: Security</i>					
1.4.001	Agreed-upon security requirements not supported.	1	2	2	Extensive	010-040-010
1.4.002	"Created By" and "Last Changed By" audit trail information lost.	3	1	3	Extensive	010-040-020
1.4.003	Outsourced	3	2	6	Broad	010-

	telemarketers see actual credit scores and other privileged information.					040-030
1.4.004	Internet applications insecure against intentional attacks.	1	2	2	Extensive	010-040-040
1.4.005	Internet applications insecure against unintentional attacks.	3	2	6	Broad	010-040-040
1.4.006	Anonymous browsing on Internet not permitted.	2	4	8	Broad	010-040-050
1.4.007	Fraud detection too lenient.	3	1	3	Extensive	010-040-060
1.4.008	Fraud detection too strict.	3	1	3	Extensive	010-040-060
1.5.000	<i>Functionality: Compliance</i>					
1.5.001	[Functional risks related to compliance go in this section.]					

3.2.19.1. Project Risk By-Products

In the course of preparing the quality risk analysis document, I observed one project risk inherent in the requirements. That is, given the lack of clarity around security requirements (see 010-040-010), there's a strong chance that the necessary infrastructure won't be in place as needed to support this project schedule. How many project risks did you notice?

3.2.19.2. Requirements Defect By-Products

In the course of preparing the quality risk analysis document, I observed the following defects in the requirements.

1. For 010-010-150, what are the supported States, Provinces, and Countries?
2. For 010-010-150, the phrase "all support States" should be "all supported States".
3. For 010-020-020, the phrase "dept-to-income" should be "debt-to-income".
4. For some requirements, the ID number is duplicated; e.g., 010-030-140.
5. Not all requirements are prioritized; e.g., 010-030-150.
6. For 010-040-010, what are the agreed-upon security approaches discussed here?
7. There doesn't appear to be any direct mention of regulatory compliance requirements, though certainly many would apply for a bank.

How many requirements problems did you notice?

3.2.20. Failure Mode and Effect Analysis

The last, and most formal, technique we'll consider for risk-based testing is Failure Mode and Effect Analysis. This technique was developed originally as a design-for-quality technique. However, you can extend it for risk-based software and systems testing. As with an informal technique, we identify quality risk items, in this case called failure modes. We tend to be more fine-grained about this than we would in an informal approach. This is in part because, after identifying the failure modes, we then identify the effects those failure modes would have on users, customers, society, the business, and other project stakeholders.

This technique has as its strength the properties of precision and meticulousness. When it's used properly, you're less likely to miss an important quality risk with this technique than with the other techniques. Hazard analysis is similarly precise, but it tends to be overwhelmed by complexity due to the need to analyze the upstream hazards that cause risks. For Failure Mode and Effect Analysis (often called FMEA, or "fuh-me-uh"), the downstream analysis of effects is easier, making the technique more general.

ISTQB Glossary

Failure Mode and Effect Analysis (FMEA): A systematic approach to risk identification and analysis in which you identify possible modes of failure and attempt to prevent their occurrence.

Failure Mode, Effect and Criticality Analysis (FMECA): An extension of FMEA, as in addition to the basic FMEA, it includes a criticality analysis, which is used to chart the probability of failure modes against the severity of their consequences. The result highlights failure modes with relatively high probability and severity of consequences, allowing remedial effort to be directed where it will produce the greatest value.

However, this precision and meticulousness has its weaknesses. It tends to produce lengthy outputs. It is document heavy. The large volume of documentation produced requires a lot of work not only during the initial analysis, but also during maintenance of the analysis during the project and on subsequent projects. It is also hard to learn, requiring much practice to master. If you must learn to use FMEA, it's best to start with an informal technique for quality risk analysis on another project first or to first do an informal quality risk analysis and then upgrade that to FMEA after it is complete.

I have used FMEA on a number of projects, and would definitely consider it for high-risk or conservative projects. However, for chaotic, fast-changing, or prototyping projects, I would avoid it.

Failure mode and effect analysis was originally developed to help prevent defects during design and implementation work. I came across the idea initially in D.H. Stamatis's book *Failure Mode and Effect Analysis* and decided to apply it to software and hardware/software systems based on some work I was doing with clients in the mid-1990s. I later included a discussion of it in my first book, *Managing the Testing Process*, published in 1999, which as far as I know makes the first software-testing-focused discussion of the technique. I discussed it further in *Critical Testing Processes* as well. So, I can't claim to have invented the technique by any means, but I can claim to have been a leading popularizer of the technique amongst software testers.

Failure Mode and Effect Analysis exists in several variants. One is Failure Mode and Effects and Criticality Analysis (FMECA, or "fuh-me-kuh"), where the criticality of each effect is assessed along with other factors affecting the level of risk for the

effect in question.

Two other variants—at least in naming—exist when the technique is applied to software. These are software failure mode and effect analysis and software failure mode, effects and criticality analysis. In practice, I usually hear people use the terms FMEA and FMECA in the context of both software and hardware/software systems. In this book, we'll focus on FMEA. The changes involved in the criticality analysis are minor and we can ignore them here.

3.2.21. Quality Risk Analysis Using Failure Mode and Effect Analysis

The FMEA approach is iterative. In other words, reevaluation of residual risk—an effect-by-effect basis—is repeated throughout the process. Because this technique began as a design and implementation technique, ideally the technique is used early in the project.

As with other forms of risk analysis, we would expect test analysts and test managers to contribute to the process and the creation of the FMEA document. Because the documents can be intricate, it's important that testers who want to contribute understand their purpose and application. As with any other risk analysis, test analysts and test managers, like all participants, should be able to apply their knowledge, skills, experience, and unique outlook to help perform risk analysis itself, following a FMEA approach.

As I mentioned before, FMEA and its variants are not ideal for all projects. However, it should be applied when appropriate, as it is precise and thorough. Specifically, FMEA makes sense under the following circumstances:

- The software, system, or system of systems is potentially critical and the risk of failure must be brought to a minimum. For example, avionics software, industrial control software, and nuclear control software would deserve this type of scrutiny.
- The system is subject to mandatory risk-reduction or regulatory requirements—for example, medical systems or those subject to ISO 6150.
- The risk of project delay is unacceptable, so management has decided to invest extra effort to remove defects during early stages of the project. This involves using the design and implementation aspects of FMEA more so than the testing aspects.
- The system is both complex and safety critical, so close analysis is needed.

define special test considerations, operational constraints, and design decisions. For example, a battlefield command, communication, and control system that tied together disparate systems participating in the ever-changing scenario of a modern battle would benefit from the technique.

As I mentioned earlier, if necessary, you can use an informal quality risk analysis technique first, then augment that to include the additional precision and facts considered with FMEA.

Because FMEA arose from the world of design and implementation—not testing and because it is inherently iterative, you should plan to schedule FMEA activities very early in the process, even if only preliminary, high-level information is available. For example, a marketing requirements document or even a project charter can suffice to start. As more information becomes available, and as decisions firm up, you can refine the FMEA based on the additional details.

Additionally, you can perform a FMEA at any level of system or software decomposition. In other words, you can—and I have—perform a FMEA on a system, but you can—and I have—also perform it on a subset of system modules during integration testing or even on a single module or component.

Whether you start at the system level, the integration level, or the component level, the process is the same. First, working function-by-function, quality characteristic-by-quality characteristic, or quality risk category-by-quality risk category, identify the failure modes. A failure mode is exactly what it sounds like: a way in which something can fail. For example, if we are considering an e-commerce system's security, a failure mode could be "Allows inappropriate access to customer credit card data". So far, this probably sounds much like informal quality risk analysis to you, but the next step is the point at which it gets different.

In the next step, we try to identify the possible causes for each failure mode. This is not something included in the informal techniques we discussed before. Why do we do this? Well, remember that FMEA is originally a design and implementation tool. We try to identify causes for failures so we can define them out of the design and avoid introducing them into the implementation. To continue with our e-commerce example, one cause of the inappropriate access failure mode could be "Credit card data not encrypted".

The next step, also unique to FMEA, is that, for each failure mode, we identify the possible effects. Those effects can be on the system itself, on users, on

customers, on other project and product stakeholders, even on society as a whole. (Remember, this technique is often used for safety-critical systems like nuclear control where society is indeed affected by failures.) Again, using our commerce example, one effect of the access failure mode could be "Fraudulent charges to customer credit cards".

Based on these three elements—the failure mode, the cause, and the effect—we can then assess the level of risk. We'll look at how this works in just a moment. We can also assess criticality. In our e-commerce example, we would say that leakage of credit card data is critical.

Now, we can decide what types of mitigation or risk reduction steps we can take for each failure mode. In our informal approaches to quality risk analysis, we limited ourselves to defining an extent of testing to be performed here. However, in FMEA—assuming we involved the right people—we can specify other design and implementation steps too. For the e-commerce example, a mitigation step might be "Encrypt all credit card data". A testing step might be "Penetration-test the encryption".

Notice that this example highlights the iterative elements of this technique. The mitigation step of encryption reduces the likelihood of the failure mode, but it introduces new causes for the failure mode, such as "Weak keys used for encryption".

We not only iterate during the process, we iterate at regular intervals in the lifecycle, as we gain new information and carry out risk mitigation steps, to refine the failure modes, causes, effects, and mitigation actions.

3.2.22. Determining the Risk Priority Number

Let's return to the topic of risk factors and the overall level of risk. In FMEA, people commonly refer to the overall level of risk as the risk priority number, RPN.

When doing FMEA, there are typically three risk factors used to determine the risk priority number:

- Severity. This is an assessment of the impact of the failure mode on the system, based on the failure mode itself and the effects.
- Priority. This is an assessment of the impact of the failure mode on users, customers, the business, stakeholders, the project, the product, and society, based on the effects.

- **Detection.** This is an assessment of the likelihood of the problem existing in the system and escaping detection without any additional mitigation. This takes into consideration the causes of the failure mode and the failure mode itself.

People performing a FMEA often rate these risk factors on a numerical scale. You can use a 1 to 10 scale, though a 1 to 5 scale is also common. You can use either a descending or ascending, so long as each of the factors uses the same type of scale, either all descending or all ascending. In other words, 1 can be the most risky assessment or the least risky, respectively. If you use a 1 to 10 scale, then a descending scale means 10 is the least risky. If you use a 1 to 5 scale, then a descending scale means 5 is the least risky. For ascending scales, the most risky would be 10 or 5, depending on the scale. Personally, I always worry about using anything finer grained than a five-point scale. Unless I can actually tell the difference between a 9 and a 10 or a 2 and a 3, for example, it seems like I'm just lying to others and myself about the level of detail at which I understand risks. Trying to achieve this degree of precision can also lengthen debates between stakeholders in the risk analysis process, often to little if any benefit.

As I mentioned before, you determine the overall or aggregate measure of risk using the risk priority number (or RPN), using the three factors. The simplest way to do this—and one in common use—is to multiply the three factors. However, you can also add the factors. You can also use more complex calculations, including the use of weighting to emphasize one or two factors.

As with risk priority numbers for the informal techniques discussed earlier, the FMEA RPN will help determine the level of effort we invest in risk mitigation. However, note that FMEA risk mitigation isn't always just through testing. In fact, multiple levels of risk mitigation could occur, particularly if the RPN is serious enough.

Where failure modes are addressed through testing, we can use the FMEA RPN to sequence the test cases. Each test case inherits the RPN for the highest-priority risk related to it. We can then sequence the test cases in risk priority order wherever possible.

3.2.23. Benefits, Costs, and Challenges of FMEA

So, what are the benefits of FMEA? In addition to being precise and thorough—and thus less likely to misassess or omit risks—FMEA provides other advantages. It requires detailed analysis of expected system failures that could be caused by software failures or usage errors, resulting in a complete view—if perhaps an

overwhelming view—of the potential problems.

If FMEA is used at the system level—rather than only at a component level—you can have a detailed view of potential problems across the system. In other words, if we consider systemic risks, including emergent reliability, security, a performance risks, we have a deeply informed understanding of system risks. Again, those performing and especially managing the analysis can find this overwhelming, and it certainly requires a significant time investment to understand the entire view and its import.

As I've mentioned, another advantage of FMEA—as opposed to other quality risk analysis techniques discussed—is that we can use our analysis to help guide design and implementation decisions. The analysis can also provide justification for not doing certain things, for avoiding certain design decisions, for not implementing in a particular way or with a particular technology.

As with any quality risk analysis technique, our FMEA analysis can focus our testing on specific, critical areas of the system. However, because it's more precise than other techniques, the focusing effect is correspondingly more precise. This can have test design implications, too, because you might choose to implement more fine-grained tests to take the finer-grained understanding of risk into account.

There are costs and challenges associated with FMEA, of course. For one thing you have to force yourself to think about use cases, scenarios, and other reali that can lead to sequences of failures. Because of the fine-grained nature of the analysis, it's easy to focus on each failure mode in isolation, without consideri everything else that's going on. You can—and should—overcome this challeng of course.

As mentioned a few times, the FMEA tables and other documentation can be huge. This means that participants and those managing the analysis can find the development and maintenance of these documents a large, time-consuming, expensive investment.

As originally conceived, FMEA works function-by-function. When looking at a component or a complex system, it might be difficult to define independent functions. I've managed to get around this myself by doing the analysis not just by function, but by quality characteristic or by quality risk category.

Finally, when trying to anticipate causes, it might be challenging to distinguish true causes from intermediate effects. For example, suppose we are considering a failure mode for an e-commerce system such as "Foreign currency transaction

rejected". We could list a cause as "Credit card validation cannot handle foreign currency". However, the true cause might be that we simply haven't enabled foreign currency processing with our credit card processing vendor, which is a simple implementation detail—provided someone remembers to do it. These challenges are in addition to those discussed earlier for quality risk analysis in general.

3.2.24. Case Study of FMEA

In [figure 3-4](#), you see an example of a quality risk analysis document. It is a case study of an actual project. This document—and the approach we used—followed the Failure Mode and Effect Analysis approach.

As you can see, we started—at the left side of the figure—with a specific function and then identified failure modes and their possible effects. We determined criticality based on the effects, along with the severity and priority. We listed possible causes to enable bug prevention work during requirements, design, and implementation.

Next, we looked at detection methods—those methods we expected to apply anyway for this project. The more likely the failure mode was to escape detection, the worse the detection number. We calculated a risk priority number based on the severity, priority, and detection numbers. Smaller numbers were worse. Severity, priority, and detection each ranged from 1 to 5. So the risk priority number ranged from 1 to 125.

This particular figure shows the highest-level risk items only because it was sorted by risk priority number. For these risk items, we would expect a lot of additional detection and other recommended risk control actions. You can see that we have assigned some additional actions at this point but have not yet assigned the owners.

Figure 3-4. Case study of Failure Mode and Effect Analysis

Failure Mode and Effects Analysis (Quality Risks Analysis) Form--RPN Sorted										
Initial FMEA--RPN Sorted										
System Function or Feature	Potential Failure Mode(s)- Quality Risk(s)	Potential Effects of Failure	Critical?	Severity	Potential Cause(s) of Failure	Priority	Detection Method(s)	Detection Risk Pil No	Recommended Action	Who/ When?
Shreds Swap File	fails to shred	Security Breach	Y	1	Program Error	1	Test; Debug	1	Test; Debug	
Shreds Swap Files	Shreds Excessively	Data Loss	Y	1	Program Error	1	Trace; Code Review	1	Tracing; Code Review	
Compression Compatibility	Damages Data	Data Loss	Y	1	Program Error	1	Test	2	Test	
Compression Compatibility	Hangs System	Data Loss	Y	1	Program Error	1	Test	2	Test	
Compression Compatibility	Shreds Improperly	Data Loss	Y	1	Program Error	1	Test	2	Test	
Internet Files Recognition	Recognizes Incorrectly	Data Loss	Y	1	Program Error	1	Trace; Code Review	2	Test; Rules Validation	
Network Compatibility	Shreds Network	Data Loss	Y	1	Program Error	1	Test	2	Test Selected/ Improve Network Coverage	
Removes File Name	Damages FS	Data Loss	Y	1	Program Error	1	Trace; Code Review	2	Test; Debug	
Removes File Name	Fails to Remove	Security Breach	Y	1	Program Error	1	Trace; Code Review	2	Tracing; Code Review	

During testing actions associated with a risk item, we would expect that the number of test cases, the amount of test data, and the degree of test coverage would all increase as the risk increased. Notice that we can allow any test procedures that cover a risk item to inherit the level of risk from the risk item. That documents the priority of the test procedure, based on the level of risk.

3.2.25. Risk-Based Testing and Failure Mode and Effect Analysis Exercise 3

In the previous exercise, you did an informal quality risk analysis for the HELLOCARMS project. Now, revise that informal quality risk analysis to use a failure mode and effect analysis (FMEA) approach.

To do this, there are three main steps:

1. For each risk item (which we'll call a failure mode here), add one or more potential causes and potential effects.
2. For each risk item, based on the causes and effects, assess the severity, priority, and detection using a five-point scale where 1 is the riskiest rating and 5 is the least risky rate for each.
3. Calculate the risk priority as the product of the three factors, severity, priority, and detection.

Once done, compare the FMEA document to the informal document. What is the

increase in size and detail, for example?

You may omit any revisions to the "extent of testing" and "tracing" columns. The former would require an extensive discussion to revise, and the latter should not require revision.

If working in a classroom, break into groups of three to five. If you are working alone, you'll need to do this exercise yourself. Use the template on the next page. To keep the time spent reasonable, I suggest 45 minutes for this transformation.

If you are working in a classroom, once each group has finished its transformation, discuss the results.

Table 3-6. Functional quality risk analysis template

No.	Failure Mode	Potential Cause	Potential Effect	Sev	Pri	Det	R
1.1.000	Functionality: Suitability						
1.1.001	[Functional risks related to suitability go in this section.]						
1.2.000	Functionality: Accuracy						
1.2.001	[Functional risks related to accuracy go in this section.]						
1.3.000	Functionality: Interoperability						
1.3.001	[Functional risks related to interoperability go in this section.]						
1.4.000	Functionality: Security						
1.4.001	[Functional risks related to security go in this section.]						
1.5.000	Functionality: Compliance						
1.5.001	[Functional risks related to compliance go in this section.]						

3.2.26. Risk-Based Testing and Failure Mode and Effect Analysis Exercise 3 Debrief

You can see my solution to the exercise in Table 3-7.

A few notable points:

1. I used the "likelihood" score from the informal analysis as the "detection" score here in most cases because the meaning is about the same.
2. I used the "impact" score from the informal analysis as the "priority" score here in most cases because the meaning is about the same.
3. I ranked the "severity" based on the impact on the system, using the common scale found in bug tracking systems:
 1. Loss of data
 2. Loss of functionality with no workaround
 3. Loss of functionality with a workaround
 4. Partial loss of functionality
 5. Cosmetic issue
4. The informal risk analysis consists of about 900 words, while the FMEA table has about 1,400, around 50 percent more. In fact, the transformation of informal risk analysis probably limited the expansion of the text. Typically we see formal risk analysis techniques result in a much greater volume of information than the informal techniques would.

Table 3-7. Functional quality risk analysis for HELLOCARMS

No.	Failure Mode	Potential Cause	Potential Effect	Sev	Pri	Det	R
1.1.000	<i>Functionality: Suitability</i>						
1.1.001	Reject all applications for home equity loans.	Logic error Scoring failure	Major business loss	2	1	5	10
1.1.002	Reject all applications for home equity lines of credit.	Logic error Scoring failure	Significant business loss	2	1	5	10
1.1.003	Reject all applications for reverse mortgages.	Logic error Scoring failure	Some business loss	2	1	5	10
1.1.004	Fail to properly process some home equity applications.	Logic error Scoring failure	Some business loss	3	1	3	9

1.1.005	Fail to properly process some home equity line of credit applications.	Logic error Scoring failure	Minor business loss	3	2	3	18
1.1.006	Fail to properly process some home equity reverse mortgage applications.	Logic error Scoring failure	Minimal business loss	3	3	3	21
1.1.007	Fail to properly process some combined products (e.g., home equity and credit cards).	Logic error Scoring failure	Minimal business loss	3	4	3	30
1.1.008	Fail to properly process some original mortgage applications.	Logic error Scoring failure	Minimal business loss	3	5	3	45
1.1.009	Fail to properly process some preapproved applications.	Logic error Scoring failure	Minimal business loss	3	4	3	30
1.1.010	Scripts not available for all fields and screens.	User interface error	Some improper loan handling	3	2	4	24
1.1.011	Customer source data not collected.	User interface error	Loss of marketing data	1	2	3	6
1.1.012	Customer source data categories not well defined.	User training problem User interface error	Partial loss of marketing data	1	2	2	4
1.1.013	Accepts invalid data at input fields.	User interface error	Improper loan rejection and acceptance	3	1	1	3
1.1.014	Existing trade lines not displayed and/or processed properly.	User interface error Logic error	Improper rejection of some loans	3	1	3	9
1.1.015	Trade line payoff details not passed to LoDoPS.	Logic error LoDoPS interface failure	Lack of proper stipulations on customers	2	1	2	4

1.1.016	Loan to be paid off included in debt-to-income calculations.	Logic error Scoring failure	Improper rejection of some loans	2	3	4	2
1.1.017	Cannot resume incomplete/interrupted applications.	User interface error Logic error	Wasted agent time Minor business loss	2	2	3	1
1.1.018	Applicant not asked about existing relationship.	User interface error	Loss of marketing data	2	2	5	2
1.1.019	Applicant existing relationship not passed to GLADS.	GLADS interface failure	Loss of marketing data	1	2	3	6
1.1.020	Loan status information lost after initiation.	LoDoPS interface failure LoDoPS failure	Unable to provide status on customer request	3	2	3	1
1.1.021	Cannot abort or abandon an application cleanly (i.e., must close browser).	Logic error User interface error	Waste of agent time	4	3	4	3
1.1.022	Cannot retrieve existing application by customer ID.	Logic error User interface error	Waste of agent time Customer inconvenience	4	4	4	4
1.1.023	Loans over \$500, 000 not transferred for approval.	Logic error User interface error	Improper loan issuance	2	1	4	8
1.1.024	Loans over \$500, 000 automatically denied.	Logic error User interface error	Improper loan rejection	2	1	3	6
1.1.025	Property valuation over \$1, 000,000 not transferred for approval.	Logic error User interface error	Improper loan issuance	2	2	4	10
1.1.026	Property valuation over \$1, 000,000 automatically denied.	Logic error User interface error	Improper loan rejection	2	2	3	1
1.1.027	Inbound telemarketing	Logic error	Some business	2	2	1	4

	operations fail in supported region.	User interface error	loss				
1.1.028	Outbound telemarketing operations fail in supported region.	Logic error User interface error	Some business loss	2	2	1	4
1.1.029	Branding for brokers and other business partners not supported.	User interface error	Some business loss	4	2	4	2
1.1.030	Untrained users (e.g., end customers) cannot enter applications via Internet.	User interface usability problems Security problem	Some business loss	4	3	4	3
1.1.031	Product operations for retail bank branches not supported.	Logic error Security problem Branch interface error	Some business loss	4	4	4	4
1.1.032	Flexible pricing schemes not supported.	Logic error	Some business loss	4	5	4	4
1.2.000	<i>Functionality: Accuracy</i>						
1.2.001	Customer presented with products for which they are ineligible.	Logic error Scoring failure	Improper issuance of loans	2	1	3	6
1.2.002	Customer not presented with products for which they are eligible.	Logic error Scoring failure	Improper rejection of applicants	2	1	4	8
1.2.003	Application decisioning inconsistent with Globobank credit policies.	Logic error Scoring failure	Improper loan rejection and acceptance	2	1	3	6
1.2.004	Risk-based pricing miscalculated based on	Logic error Scoring failure	Logic error Scoring failure	2	1	4	8

	credit score, loan to value, and debt to income.						
1.2.005	New loan payment not included in credit scoring.	User interface error Logic error	Improper issuance of some loans	2	2	3	1
1.2.006	Pricing add-ons not calculated correctly.	Logic error	Cheat customer or Globobank	2	3	3	1
1.2.007	Government retirement income not handled properly.	Logic error Scoring failure	Improper loan rejection and acceptance	2	1	3	6
1.2.008	Duration of additional income not captured.	Logic error Scoring failure	Improper loan rejection and acceptance	4	3	4	2
1.3.000	<i>Functionality: Interoperability</i>						
1.3.001	Can't pull information from GloboRainBQW into HELLOCARMS.	GloboRainBQW interface failure	Loss of marketing data	1	2	2	4
1.3.002	HELOCARMS and Scoring Mainframe reject joint applications.	Logic error	Some business loss	4	1	3	6
1.3.003	HELOCARMS and Scoring Mainframe cannot handle/resolve duplication of information on joint applications.	Logic error	Some business loss	4	2	1	4
1.3.004	HELOCARMS trade line communication to LoDoPS fails.	LoDoPS interface error	Lack of proper stipulations on customers	2	1	3	6
1.3.005	Loan status information from LoDoPS to HELLOCARMS lost or corrupted.	LoDoPS interface error	Unable to provide status on customer request	4	2	5	3

1.3.006	HELLOCARMS can't continue if the Scoring Mainframe indicates an undischarged bankruptcy or foreclosure.	Logic error	Improper loan rejection	2	3	4	2
1.3.007	HELLOCARMS communication of government retirement income to LoDoPS fails.	LoDoPS interface failure	Improper loan rejection	2	1	5	10
1.3.008	HELLOCARMS application information not passed to Scoring Mainframe properly.	Scoring interface failure	Improper loan rejection and acceptance	1	1	3	3
1.3.009	HELLOCARMS does not receive information from Scoring Mainframe properly.	LoDoPS interface failure	Improper loan rejection and acceptance	1	1	3	3
1.3.010	Decisioning requests not queued for Scoring Mainframe as needed.	Decisioning interface failure	Improper loan rejection	1	2	4	8
1.3.011	Tentatively approved, customer-accepted loans not passed to LoDoPS.	LoDoPS interface failure	Improper loan rejection	1	2	5	10
1.3.012	Declined applications not passed to LoDoPS.	LoDoPS interface failure	Regulatory non compliance Lawsuit Reputation damage	1	2	5	10
1.3.013	Changes made in loan information in LoDoPS not propagated back to HELLOCARMS.	LoDoPS interface failure	Improper status provided to customer	4	2	5	30
1.3.014	Computer-telephony integration not supported.	Call center phone interface	Waste of agent time Customer	2	5	3	30

1.3.015	Applicant existing relationship not passed to GLADS.	failure GLADS interface failure	inconvenience Loss of marketing data	1	3	4	10
1.4.000	<i>Functionality: Security</i>						
1.4.001	Agreed-upon security requirements not supported.	Firewall failure Encryption problem	Regulatory non compliance Lawsuit	2	2	1	4
1.4.002	"Created By" and "Last Changed By" audit trail information lost.	Logic error	Regulatory non compliance Lawsuit Reputation damage	2	1	3	6
1.4.003	Outsourced telemarketers see actual credit scores and other privileged information.	Encryption problem User interface error Privilege logic error	Regulatory noncompliance Lawsuit Reputation damage	2	2	3	10
1.4.004	Internet applications insecure against intentional attacks.	Encryption problem User interface error Privilege logic error	Regulatory noncompliance Lawsuit Reputation damage	2	2	1	4
1.4.005	Internet applications insecure against unintentional attacks.	Encryption problem User interface error Privilege logic error	Regulatory noncompliance Lawsuit Reputation damage	2	2	3	10
1.4.006	Anonymous browsing on Internet not permitted.	User interface error Privilege logic error	Some loss of business	4	4	2	10
1.4.007	Fraud detection too lenient.	Logic error Scoring interface error	Improper loan issuance	2	1	3	6
1.4.008	Fraud detection too strict.	Logic error Scoring interface error	Improper loan rejection	2	1	3	6

1.5.000	<i>Functionality:</i> <i>Compliance</i>					
1.5.001	[Functional risks related to compliance go in this section.]					

3.2.27. Risk Based Testing and the Testing Process

We've talked so far about quality risk analysis techniques. As with any technique, we have to align and integrate the selected quality risk analysis technique with the larger testing process and indeed the larger software or system development process. [Table 3-8](#) shows a general process that you can use to organize the quality risk identification, assessment, and management process for quality risk-based testing.^[5]

[5] This process was first published in my book *Critical Testing Processes*.

Table 3-8. Quality risk analysis process

Step#	Step
1	Identify the key testing and quality stakeholders. Obtain stakeholder commitment to participate in a quality risk analysis.
2	Survey the key stakeholders about the techniques and methods for quality risk analysis. If appropriate, propose a technique. Obtain consensus on the technique and the method selected.
3	Gather ideas from the key stakeholders about the quality risks, the failure modes associated with those risks, the quality impact of such failures, and the priority of the risks. Identify the recommended action to mitigate each risk.
4	Report any incipient bugs identified in other project documents during the analysis, such as bad or missing requirements, design problems, and so forth.
5	Document the quality risks as appropriate for the technique used. Circulate the document to the stakeholders for approval. Iterate steps 4, and 5 as necessary to finalize the quality risks, their priorities, and the recommended actions.
6	Check the quality risk analysis document(s) into the project library or configuration management system. Place the document under change control.

Let's go through the process step-by-step and in detail. Identify the stakeholders who will participate. This is essential to obtain the most benefit from quality risk based testing. You want a cross-functional team that represents the interests of all stakeholders in testing and quality. The better the representation of all interests, the less likely it is that you will miss key risk items or improperly estimate the levels of risk associated with each risk item. These stakeholders are typically in two groups. The first group consists of those who understand the needs and interests of the customers and users—or are the customers and users. They see potential business-related problems and can assess the impact. The second group consists of those who understand the technical details. They see what is likely to go wrong and how likely it is to go wrong.

Select a technique. The previous sections should have given you some ideas on how to do that.

Identify the quality risk items using the technique chosen. Assess the level of risk associated with each item. The identification and assessment can occur as a single meeting, using brainstorming or similar techniques, or as a series of interviews, either with small groups or one-on-one. Try to achieve consensus on the rating for each risk item. If you can't, escalate to the appropriate level of management. Now select appropriate mitigation techniques. Remember that this doesn't just have to be testing at one or more levels. It can also include reviews of requirements, design, and code; defensive programming techniques; static analysis to ensure secure and high-quality code; and so forth.

Deliver the by-products. Risk identification and analysis often locates problem requirements, design, code, or other project documents, models, and deliverables. These by-products can be actual defects in these documents, project risks, and implementation assumptions and suggestions. You should see these by-products to the right person for handling.

Review, revise, and finalize the quality risk document that was produced.

This document is now a valuable project work product. You should save it to the project repository, placing it under some form of change control. The document should change only with the knowledge of—ideally, the consent of—the other stakeholders who participated.

That said, it will change. You should plan to revise the risk assessment at regular intervals. For example, review and update the document at major project milestones such as the completion of the requirements, design, and

implementation phases and at test level entry and exit reviews. Also, review and update when significant chunks of new information become available, such as the completion of the first test cycle in a test level. You should plan to add new risk items and reassess the level of risk for the existing items.

Throughout this process, be careful to preserve the collaborative nature of the endeavor. In addition to the information-gathering nature of the process, the consensus-building aspects are critical. Both business-focused and technically focused participants can and should help prioritize the risks and select mitigation strategies. This way, everyone has some responsibility for and ownership of the testing effort that will be undertaken.

3.2.28. Risk-Based Testing throughout the Lifecycle

A basic principle of testing discussed in the Foundation syllabus is the principle of early testing and quality assurance. This principle stresses the preventive potential of testing. Preventive testing is part of analytical risk-based testing, implicit in the informal quality risk analysis techniques and explicit in FMEA.

Preventive testing means that we mitigate risk before test execution starts. This can entail early preparation of testware, pretesting test environments, pretesting early versions of the product well before a test level starts, insisting on tough entry criteria to testing, ensuring requirements for and designing for testability, participating in reviews including retrospectives for earlier project activities, participating in problem and change management, and monitoring the project progress and quality.

In preventive testing, we integrate quality risk control actions into the entire lifecycle. Test managers should look for opportunities to control risk using various techniques, such as those listed here:

- An appropriate test design technique
- Reviews and inspection
- Reviews of test design
- An appropriate level of independence for the various levels of testing
- The use of the most experienced person on test tasks
- The strategies chosen for confirmation testing (retesting) and regression testing

Preventive test strategies acknowledge that we can and should mitigate quality risks using a broad range of activities, many of them not what we traditionally think of as "testing". For example, if the requirements are not well written, perhaps we should institute reviews to improve their quality rather than relying on tests that will be run after the badly written requirements become a bad design and ultimately bad, buggy code?

Dynamic testing is not effective against all kinds of quality risks. For example, while we can easily find maintainability issues related to poor coding practices in a code review—which is a static test—dynamic testing will only reveal the consequences of unmaintainable code over time, as excessive regressions start to occur.

In some cases, it's possible to estimate the risk reduction effectiveness of test in general and of specific test techniques for given risk items. For example, user case-based functional tests are unlikely to do much to reduce performance or reliability risks.

So, there's not much point in using dynamic testing to reduce risk where there is a low level of test effectiveness. Quality risk analysis, done earlier in the project, makes project stakeholders aware of quality risk mitigation opportunities in addition to—and in many cases in advance of—levels of dynamic testing like system test and system integration test.

Once we do get to dynamic test execution, we use test execution to mitigate quality risks. Where testing finds defects, testers reduce risk by providing the awareness of defects and opportunities to deal with them before release. Where testing does not find defects, testing reduces risk by ensuring that under certain conditions the system operates correctly.

I mentioned earlier that we use level of risk to prioritize tests in a risk-based strategy. This can work in a variety of ways, with two extremes referred to as depth-first and breadth-first. In a *depth-first* approach, all of the highest-risk tests are run before any lower risk tests, and tests are run in strict risk order. In a *breadth-first* approach, we select a sample of tests across all the identified risks using the level of risk to weight the selection while at the same time ensuring coverage of every risk at least once.

As we run tests, we should measure and report our results in terms of residual risk. The higher the test coverage in an area, the lower the residual risk. The fewer bugs we've found in an area, the lower the residual risk. Of course, in doing risk-based testing, if we only test based on our risk analysis, this can lead to

blind spots, so we need to use testing outside the predefined test procedures to see if we have missed anything. This is where experienced-based testing techniques, discussed extensively in the companion volume for Advanced Test Analysts, provide superior value.

If, during test execution, we need to reduce the time or effort spent on testing, we can use risk as a guide. If the residual risk is acceptable, we can curtail our tests. Whatever tests we haven't run are less important than those we have run. If we do curtail further testing, that serves to transfer the remaining risk onto the users, customers, help desk and technical support personnel, or operation staff.

Suppose we do have time to continue test execution? In this case, we can adjust our risk analysis—and thus our testing—for further test cycles based on what we've learned from our current testing. First, we revise our risk analysis. Then we reprioritize existing tests and possibly add new tests. What should we look to decide whether to adjust our risk analysis? Here are some main factors to look for:

- Totally new or very much changed product risks
- Unstable or defect-prone areas discovered during the testing
- Risks, especially regression risk, associated with fixed defects
- Discovery of unexpected bug clusters
- Discovery of business-critical areas that were missed

If you have time for new additional test cycles, consider revising your quality analysis first. You should also update the quality risk analysis at each project milestone.

3.2.29. Risk-Based Testing in the Fundamental Test Process

Now we'll talk about how risk analysis and risk-based testing fits into the fundamental test process. Let's start with the overall direction or objectives of testing, which sets the context of test planning.

As mentioned earlier, risk management should occur throughout the entire lifecycle. An organization can use a test policy document or test strategy document to describe processes for managing product and project risks through testing. The document can also show how risk management is integrated into

and affects the various test levels. In a subsequent section on test documentation, we'll talk about test policy documents and test strategy documents.

Test planning is the first activity of the ISTQB fundamental test process and primarily a test management activity. When a risk-based test strategy is used risk identification and risk analysis activities should be the foundation of test planning and should occur at the beginning of the project. This is true whatever software development lifecycle model is followed, not just for the Agile or iterative models.

ISTQB Glossary

master test plan: A test plan that typically addresses multiple test levels.

test plan: A document describing the scope, approach, resources, and schedule of intended test activities. It identifies, among other test items, the features to be tested, the testing tasks, who will do each task, the degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice, and any risks requiring contingency planning. It is a record of the test planning process. [After IEEE 829]

Given our risk analysis, we can then address various risk mitigation activities in our test plans. This activity can occur in the master test plan for project-wide risks and at the level test plans for level-specific risks. Certain risks are best addressed in particular levels, and certain risks are serious enough to require attention at multiple levels.

As discussed earlier, the level of risk determines the amount of test effort and the sequence of tests. The plan should describe how that allocation of effort and sequencing happens. In addition to describing how, through testing, we intend to mitigate product risks, the test plan should describe how we intend to respond and manage the project risks that could affect our planned testing.

In our test plan, the exit criteria and other sections can address the question of when risk mitigation is sufficient. What level of residual risk will project stakeholders accept? What steps should we take for particular project risks? The test plan should address these and other risk-related questions. Again, in subsequent sections of this chapter, we'll take a closer look at this.

Since we're on the topic of fitting risk management into the test planning process right now, let me give some tips on doing so. These tips apply especially to the risk identification and analysis tasks, but they can also apply more broadly to test planning.

To reinforce something I've mentioned before, this time in the context of the process itself, it's critical to use a cross-functional team for both risk identification and risk analysis. This recommendation applies not only for quality risks. It can also help when identifying test-related project risks and the measurement risks.

for dealing with them. Otherwise, you can be surprised—and not pleasantly so—during later test activities when external dependencies are not met or participants outside the test team can't or won't do what you need done for testing.

ISTQB Glossary

risk-based testing: An approach to testing in which you reduce the level of product risks and inform stakeholders on their status starting in the initial stages of a project. It involves the identification of product risks and their use in guiding the test process.

For quality risks, I recommend a two-pass approach. First, identify the quality risk items, and then assess the level of risk. There are two reasons for this approach. First, the mental activities of brainstorming and analysis are different and participants can find switching between them taxing. Second, since the levels of risk are relative, it's difficult to establish the appropriate level for each risk item until you've seen the entire list of risk items.

For test-related project risks, the situation is similar. First, we want to identify the risks we need to deal with. Once that list is in place, we can consider the actions we want to take for each risk.

A major potential problem with risk-based testing is being overwhelmed by the amount of information you need to manage. So be coarse-grained if you can. In hazard analysis and FMEA, of course, the fine-grained level of detail is required by the technique, so you'll simply have to allow the time and energy required to manage it. However, for less-formal techniques, you can minimize the amount of information you have to manage—and the documentation required to manage it—by refining risk items only when necessary to deal with different levels of risk.

For example, suppose that we're testing an e-commerce site. As a quality risk item, we might identify "Slow system response to customer input". However, if we're browsing for items, the impact of a slow response might be different than when we're checking out. We might need to split that risk item into two risk items, one for browsing, and one for checkout.

Another way to manage the potentially overwhelming amount of information is to use the most lightweight, informal technique possible given the constraints and requirements of the project. If safety or criticality issues require formal techniques like FMEA, then so be it. However, using an overly formal technique where it's not warranted is likely to sap support for the risk-based strategy as a whole and perhaps squander limited testing effort.

Let me quickly reinforce three other points mentioned earlier. First, some people

still do risk-based testing as a bug hunt, focusing on finding the largest possible number of bugs. This approach arises from considering only the technical source of risks, those that lead to a high likelihood for bugs. Not all tests should find bugs; some should build confidence. So, be sure to consider business sources of risk and test areas where the impact of bugs would be high whether bugs are likely or not.

Second, risk-based testing should be part of a preventive approach to testing. This means that we should start risk identification and analysis as early as possible in the project.

Third, remember that risk items and risk levels will evolve as the project progresses. So when thinking about how to integrate risk-based testing into the process, be sure to follow up and realign risk analysis, your testing, and the project at key project milestones.

3.2.30. Challenges of Risk-Based Testing

When you adopt risk-based testing, there are a number of challenges you'll need to surmount. Many of these are most acute during the test planning activities, but they can—and do—persist throughout the test process. Let me review some of the key challenges and offer some ideas on surmounting them.

The first of these challenges tends to arise during risk analysis. As a test manager leading a quality risk analysis effort, I often have found it difficult to build agreement and consensus on the level of risk associated with one or more risk item. Some will say likelihood is high, some will say it is low. Some will say impact is high, some will say it is low. This can create a deadlock in finalizing quality risk analysis and reduce the benefits of risk-based testing for the rest of the test process because the overall priority—and thus the effort, sequencing, and potential triage priority—remains in doubt.

There are various ways to try to resolve these disagreements. If you or perhaps one of the disagreeing parties has some political influence in the organization, you can try to influence the decision. Personally, I prefer not to favor—or be seen as favoring—one assessment over another. Only in rare circumstances would I actually feel strongly enough about the risk assessment to use my (typically scarce) political capital for this. If I have one priority assessment for each risk item, agreed upon by all stakeholders, then I tend not to care much about what the priority assessment is.

What I might try to do is broker some sort of bartering arrangement between disagreeing parties. In some cases, the disagreement will be not really about the

level of risk but rather about the relative effort and sequencing of the tests. You can perhaps ask if the two parties would accept changes in other risk levels to keep the risk levels realistic across all the risks while achieving an acceptable outcome for all.

In some cases, the root cause of the disagreement is a lack of awareness of the reasons for the assessment. In other words, you can deal with the challenge through education. You can make this happen by including the appropriate stakeholders in the risk assessment process. For example, developers, system administrators, and technical support staff will have superior insight into technical issues like the likelihood of certain types of bugs. Sales, marketing, business analysts will have superior insight into business issues like the impact of certain types of bugs.

The ultimate fallback position is, of course, escalation of the disagreement to some senior project stakeholder for resolution. Since this is likely to happen for at least some risks, you are wise to arrange for some person to have this role deadlock-breaker at the outset. This role might, in some projects or organizations, require the wisdom of Solomon and the patience of Job, so try to find the right person. Because what you are trying to do is build consensus, remember that this role requires the consent of all stakeholders; i.e., they have to agree to accept the decision of this person.^[1]

^[1] For readers with a non-Abrahamic religious background, according to the Christian, Jewish, and Muslim religious texts, Solomon was a king of Israel some three millennia ago. He was known for his ability to judge wisely and shrewdly, in a way that obtained the proper outcome. Job was a successful and devout businessman who, in a contest between God and Satan, is afflicted with a great deal of misery before ultimately being restored to health and success.

The next challenge also begins during risk assessment but can recur at any point when we reassess the level of risk. Sometimes, achieving consensus on the level of risk for most of the risk items is easy—everyone agrees that everything is a high priority! The problem here is that, as one of my clients memorably said, "When everything is a priority, there is no prioritization." Of course, this corrodes the value of risk-based testing because, with limited effort and limited time for testing, we now have lost the ability to use relative risk levels to make trade-offs.

Resolution of this problem can also involve escalation, but you can try a couple of techniques to resolve this problem first. For one thing, recognize that all

projects exist in an evolving continuum where trade-offs are made between quality, schedule, scope, and budget considerations. These trade-offs are either made deliberately, analytically, and thoughtfully, in a planned way, in advance or they are made hurriedly, reactively, and chaotically. So, if during the risk assessment you see a trend toward the "everything is a high priority" result, ask people to consider, for any risk item they are rating, what they would sacrifice to ensure thorough testing of the risk item.

Would they be willing to spend \$10, 000, \$100, 000, or \$1, 000,000 more on the project to ensure thorough testing of the risk item? Would they be willing to delay the release by a day, a week, a month, or a year to ensure thorough testing of the risk item? Would they be willing to eliminate a feature or characteristic of the product entirely if they were not sure that all the related risk items were thoroughly tested? If the answers are, "Not a dollar more, not a day later, and the feature stays in regardless of the amount of testing", then, guess what? That means the risk item is *not* a high-priority quality risk.

Another technique to deal with this problem, if it is recurring, is to adopt a risk analysis approach that makes the costs of testing explicit. For example, the cost of exposure technique discussed earlier does this.

Another challenge arises throughout the use of risk-based testing. This is the challenge of getting people to make rational decisions about risk. Unfortunate human beings are not generally rational about risk; they are often irrational.

Let me illustrate with an anecdote. I have a friend who is, like me, a traveling business consultant. Unlike me—but like many other people—he has a fundamental fear of flying. His fear is associated most strongly with an incident in La Guardia airport where a two-engine jet he was traveling in suffered what is called in pilot's parlance a "bird strike". In this particular case, a seagull went directly into one of the jets, stalling it during takeoff. This resulted in what's referred to as an "aborted takeoff", which is a mild-sounding phrase for what is one of the most dangerous things that can happen during a flight.

Now, a few years ago, my friend bought a sports car, a beautiful black Porsche. One day, while I was visiting his house near New York, he took me for a ride in it. The next thing I knew we were on the Long Island Expressway, in heavy traffic, traveling at speeds of up to 120 miles per hour. So, I was thinking, "Interesting. This is the guy who is afraid to fly."

Keep in mind that death in an automobile accident is about 100 times more likely than death in a plane accident. In fact, many if not most people in

industrialized countries will be involved in a car accident where someone is injured—I have been personally and I know people who have been seriously injured in car accidents—yet fear of *driving* is relatively rare.

For a concise, excellent explanation of why this happens, you can read Erik Simmons's paper "Risk Perception", available in the RBCS Library at www.rbcus.com. To summarize, Erik cites some factors that increase or decrease the perceived level of risk.

A feeling of dread associated with the risk increases the perceived level of risk while a feeling—accurate or not—of controllability of the risk reduces the perceived level of risk.

Risks that are imposed have an increased perceived level of risk, while those that are voluntary have a reduced perceived level of risk.

When the consequences are immediate, that increases the perceived level of risk, while delayed consequences reduces the perceived level of risk.

A low degree of knowledge about the consequences increases the perceived level of risk, while a high degree of knowledge about the consequences decreases the perceived level of risk.

Chronic consequences decrease the perceived level of risk, while acute consequences increase the perceived level of risk.

Naturally enough, severe consequences increase the perceived level of risk, while nonsevere consequences decrease the perceived level of risk.

If there is some connection, similarity, or historical relationship between the risk under consideration and some other negative event or events, that increases the perceived level of risk, particularly if the related events had severe consequences and especially if the consequences were more severe than the most likely consequences of the risk under consideration. In other words, since the human mind often understands new concepts through analogy to existing, understood concepts, any analogy, however weak, to another bad event leads to an immediate desire to avoid the risk. If there are actual examples of the risk question becoming negative events in the past, particularly if the consequences were very bad, that will increase the perceived level of risk.

Finally, familiarity or experience with the risk in previous projects or on this project decreases the perceived level of risk.

Notice that I have repeated the phrase *perceived level of risk*. This is not the same as the *actual level of risk*. That's the problem. That's the irrational element. But that's also the path to addressing the challenge. If you can get these irrational elements into the open, make them part of the discussion, you have a chance of getting people to be more rational about risk.^[1]

[1] I discuss this issue of rationality and risk in *Critical Testing Processes*. On this topic specifically and on the broader topic of risks in general, I can also recommend *The Logic of Failure, Against the Gods, Freakonomics, and True Odds*.

Further, while I've referred to these as irrational behaviors, notice that, in a biological sense, these are smart, adaptive instincts. Imagine yourself 100,000 years ago, standing on a plain in Africa, seeing a large, spotted, furry, vaguely catlike animal coming your way. You've never seen this animal before, but you have seen one that was similar in size, catlike appearance, and fur, and this animal was engaged in eating your brother the last time you saw it. Whether that animal is a tiger—very likely to eat you—or a cheetah—not very likely to eat you—is immaterial at that moment. Getting away from the animal is a smart thing to do.

Another challenge, which exists throughout the lifecycle, is the time investment required for risk-based testing. Risk-based testing requires up-front and ongoing time investments to work. So make sure that you (as the test manager) and the rest of the project management team include specific time and tasks related to doing risk identification, risk assessment, and periodic reanalysis of risk.

Finally, some risk analysis techniques have a challenge in adapting to change. If you are in a rapidly changing project—e.g., one without any structured lifecycle or one following an agile methodology—you'll probably want to try to use informal techniques rather than formal ones.

3.2.31. Risk-Based Testing and Failure Mode and Effect Analysis Exercise 4

Using the quality risk analysis for the HELLOCARMS, outline a set of functional test suites to run for HELLOCARMS system integration testing and system testing. Specifically, the system integration test level should have as an overall objective looking for defects in and building confidence around the ability of the HELLOCARMS application to work with the other applications in the datacenter. The system test level should have as an overall objective looking for defects in and building confidence around the ability of the HELLOCARMS application to

provide the necessary capabilities.

Finally, state general guidelines for sequencing the test cases within each test suite during test execution.

If you are working in a classroom, once each group has finished its work on the test suites and guidelines, discuss the results.

3.2.32. Risk-Based Testing and Failure Mode and Effect Analysis

Exercise 4 Debrief

Based on the quality risk analysis I performed, I created the following lists of system integration test suites and system test suites. Remember that we're only looking at functional tests here because the quality risk analysis focused on functional testing.

System Integration Test Suites

- HELLOCARMS/Scoring Mainframe Interfaces
- HELLOCARMS/ LoDoPS Interfaces
- HELLOCARMS/ GLADS Interfaces
- HELLOCARMS/ GloboRainBQW Interfaces

System Test Suites

- Basic Functionality
- Undesirable Applicants
- End-to-End Transactions
- Error Handling and Recovery
- Security
- Compliance

Test Case Sequencing Guidelines

Because this application exhibits a broad range of functions and must handle many different situations, I decided on a breadth-first set of test sequencing guidelines, shown below the test suite lists. In a breadth-first approach, we

select a sample of tests across all the identified risks using the level of risk to weight the selection while at the same time ensuring coverage of every risk at least once.

So prior to starting test execution, we use traceability information from our quality risk analysis to make the test procedures inherit the risk priority number of the high-risk quality risk item to which the test procedure traces. We then break the complete set of test cases into three subsets:

- **High priority:** Consists of a subset of test procedures that account for one-third of the total planned scripted test effort. Fifty percent of the test procedures have a risk priority number from 1 to 5. Thirty-five percent of the test procedures have a risk priority number from 6 to 10. Fifteen percent of the test procedures have a risk priority number from 11 to 15.
- **Medium priority:** Consists of a subset of test procedures that account for one-third of the total planned scripted test effort. Thirty-five percent of the test procedures have a risk priority number from 1 to 5. Fifty percent of the test procedures have a risk priority number from 6 to 10. Fifteen percent of the test procedures have a risk priority number from 11 to 15.
- **Low priority:** Consists of a subset of test procedures that account for one-third of the total planned scripted test effort. Fifteen percent of the test procedures have a risk priority number from 1 to 5. Fifteen percent of the test procedures have a risk priority number from 6 to 10. Seventy percent of the test procedures have a risk priority number from 11 to 15.

We then have a high-priority test period, a medium-priority test period, and a low-priority test period. During each of the three test execution periods, we run the test cases in the subset of tests in strict risk priority order.

This sequencing approach would need to flex, based on the test release schedules and size of the overall test set. For example, if the overall test set is scheduled to last six weeks and test releases are scheduled to be received every week, then we have three two-week periods, each of which involves accepting two test releases. If the overall test set is scheduled to last eight weeks, then having four subsets rather than three would make more sense.

You'll notice that this sequencing approach assumes a roughly equal balance between test procedures with risk priority numbers from 1 to 5, from 6 to 10, and from 11 to 15. If that is not the case, the plan would need to be adjusted. However, the basic intent of the plan is achievable regardless of the actual risk priority number mixture of the test procedures.

Above and beyond the planned scripted test effort, there is additional time reserved for unscripted testing of two types:

- **Confirmation testing:** Testing of any defect fixes we receive in planned or unplanned test releases that enter the test environment.
- **Reactive testing:** The use of exploratory testing, Whittaker's attacks, bug taxonomies, Hendrickson's bug hunting techniques, and other nonsystematic test techniques. (These techniques were covered in the Foundation syllabus and will be covered in more detail later in this course.) Some reactive testing will address risk items ranked as "opportunity" or below, along with risk items not identified during the risk analysis, to serve as a system of checks and balances on the quality risk analysis.

Approximately 60 to 70 percent of each test period will be spent on scripted testing and about 30 to 40 percent on confirmation testing and reactive testing.

3.2.33. Another Case Study of FMEA

Figure 3-5 shows an example of a Failure Mode and Effect Analysis chart, prepared by my associates and me on a project for a client. This is a small excerpt of a much larger worksheet some 250 rows in length. You can see that on the leftmost column, we start by identifying the specific function in question. As I mentioned before, one problem with this approach to Failure Mode and Effect Analysis is that you might miss major risk categories not related to specific functions. We dealt with that by also including sections that address emergent systemic risks like performance, reliability, and so forth.

Figure 3-5. Another case study of Failure Mode and Effect Analysis

CSA Fmea Chart												
System Function or Feature	Potential Failure Mode(s) / Quality Risk(s)	Potential Effect(s) of Failure	Caused?	Severity	Potential Cause(s) of Failure	Priority	Detection Method(s)	Detected	Block Prints	Recommended Action	Value Valuer?	References
Application Enter/Exit	CQA-Application fails to start CQA-application fails to start Application / processes on Client still running after application terminates Application / processes on Server still running after application terminates	Loss of data/service Loss of data/service Potential loss of data/service Potential loss of data/service	Y Y Y Y	1 1 2 2	Bug in CQA application Bug in CQA application Bug in CQA application Bug in CQA application	1 1 2 2	CSA string test CSA string test System Test System Test	5 5 2 2	5 String Test 5 String Test System Test System Test	Test/String Test Test/String Test Business Rules, Pg. 5 Business Rules, Pg. 5	Business Rules, Pg. 5	
Client Application exit/ing mechanism fails		Customer dissatisfaction, service loss, data corruption	Y	1	CSA app bugs	1	String Test	3	2 String Test	CSA/String Test	Business Rules, Pg. 5	
CQA-pop-up Call Record fails then hangs		Loss of data/service	Y	2	CSA app bugs	2	Integration Test	3	12 String Test	CSA/String Test	Business Rules, Pg. 1-4.5& 1-13	
CQA-pop-up call record fails then forced zero-off		Loss of data/service	Y	2	CSA app bugs	2	Integration Test	3	12 String Test	CSA/String Test	Business Rules, Pg. 1-4.5& 1-13	
CQA fails to display reason for call item Productivity		Loss of data/service	Y	3	CSA app bugs	3	Integration Test	3	12 String Test	CSA/String Test	Business Rules, Pg. 1-6	

In the next column, we identify the specific quality risk items. In the column

next to it, we identify the effects (or consequences, if you prefer) should the risk become an actual event.

You'll note the column for criticality, which makes this a FMECA variant of FMEA. Next to that column is the severity, which again is the indication of how bad the effect would be on the system. We rate the severity—like the other risk factor in this example—from 1 to 5, with 1 being the worst.

The next column shows the potential cause of the failure mode. We did not have a lot of technical participation in this particular risk analysis. That limited our insights in this area. If I had to do this particular project over, I would try to ensure better stakeholder participation.

Next, we find the priority, the impact of the effect on the users, customers, business, and other stakeholders. Again, we rated this on a 1 to 5 scale.

In the next column comes the detection method. Again, being a test-focused and tester-prepared risk analysis, we identify various test levels that might detect bugs.

Next comes the detection column, which indicates the likelihood of the defect being present and escaping detection without any explicit effort to detect it. We rated this also on a 1 to 5 scale.

Now we come to the risk priority number, or RPN. The RPN is calculated, in this example, by multiplying the severity, priority, and detection numbers.

The final three columns show the recommended action, the owner and the phase for the action, and references to project work products that relate to this risk item.

3.2.34. Risk-Based Testing and Test Control

So, let's move on in the process into test control. Once test planning is complete, risk management should continue throughout the process. This includes revisiting identification and analysis, along with taking various steps to reduce risk. We might identify new risks during test design, implementation, or execution. New information from other project activities might lead us to reassess the level of existing risks. We should evaluate the effectiveness of testing and other risk mitigation activities during test execution.

Some of these activities occur at project milestones. If we identified and assessed the risks based on the requirements specification, we should reevaluate the risks

once the design specification is finalized. If we find an unexpectedly large number of bugs during test execution for a component, we might decide that the likelihood of defects was higher than thought during the assessment. We could then adjust the likelihood and overall level of risk upward, leading to an increase in the amount of testing to be performed against this component.

Earlier in this section, I mentioned that we might sequence our tests either depth-first or breadth-first, based on risk. However, it's important to understand that, during test execution, no matter which approach we use, we might exhaust the time allocated for test execution without completing the tests. This is where risk-based testing allows us to do test triage based on risks.

We can report our results to management based on the remaining level of risk. Management can then decide to prolong testing, or they can decide, in effect, to transfer the remaining risks downstream, to users, customers, help desk and technical support, and operational staff. We'll revisit an example of how to report in terms of residual risk a little later.

Rarely (but it does happen occasionally), we find that we complete our tests with time left for additional testing. Of course, management might decide to complete the project early at that point. If management decides to continue, we can add new test cycles based on a reevaluation of risk. Our reevaluation might discover new or changed product risks, defect-ridden areas of the system (based on our test results), and new risks arising from defect fixes. We might also tune our testing to the mistakes the developers have made through an analysis of typical problems found. We could also look at areas with little test coverage and expand testing to address those.

If we do reevaluate the risks and refine our testing for subsequent cycles, we should update the quality risk documentation to reflect why we are doing what we are doing. We might need to get stakeholder approval as well.

In addition to having to manage product risks through testing during the test execution period, we also have to control test-related project risks that might affect our ability to execute tests. [Table 3-9](#) shows an example of that, from a real laptop development project.

Table 3-9. Example of controlling project risks

Risk	Impact	Trigger
CD-ROM failure in C-Test boards	Testing end slips out to 11/28 for First Customer Ship critical path items.	11/1-

delays testing due to rework.		
PC Cards initialize, but don't work under Windows.	If this problem is must-fix, it must be resolved in time to finish this test by 11/22. Given that this test takes five days, we need resolution by 11/15.	11/1
The "golden candidate" BIOS, Q3A11, will not enter testing until 11/19.	This is a risk in two ways: We might find showstoppers in the last three or four days of testing before 11/22, and the golden BIOS might ship with very little compatibility testing done against it. Vendor did not deliver a new BIOS on 11/19 as previously scheduled. This increases both risks mentioned above.	11/1

This example lists three project risks that could affect our ability to run tests. The first has to do with a potential problem with the CD-ROM drives on the laptop. If that happens, it would delay the end of testing. You see the trigger date, which is the point at which we will know if the risk has become an event in some cases, if the trigger date is reached and the risk has become an event, it "triggers" (hence the name) a contingency plan.

The next risk is actually both a product risk and a project risk at the same time. Notice that the trigger date is actually a deadline.

The final risk in [Table 3-9](#) has to do with a vendor's failure to deliver the BIOS on schedule. (The BIOS is the built-in software that helps a computer boot.) This project risk has *created* a product risk, because insufficient testing will occur.

3.2.35. Risk-Based Test Results Evaluation and Reporting

With the risk identified and assessed during test planning, and the proper test coverage achieved during test design and implementation, we can track our test execution in terms of risk reduction. In other words, we can evaluate risk-based test results in terms of risks reduced—and not reduced. Based on this evaluation, we can report risk-based test results in terms of risks reduced—and not reduced.

As a practical matter, this requires traceability from test cases and defects back to risk items, as mentioned earlier in this section. Using this traceability, our test

metrics and reporting can address the following areas:

- Risks for which one or more tests have been executed
- Risks for which one or more of the tests fail
- Risks for which one or more must-fix bugs exist

Based on these results, project stakeholders can reach smart, risk-based conclusions about important issues like readiness for release. Let's look at two examples of risk-based evaluation and reporting.

Table 3-10 shows a tabular report. In this report, we have the major risk area listed on the left. Some of the major risk areas identified during the risk analysis are gathered together in the "other" area at the bottom, presumably because they didn't identify many bugs. The list is sorted by number of defects found.

The two columns in the middle give the number of defects found and the percentage associated with each area. The three columns to the right show the planned number of tests, the executed number of tests, and the percentage of planned tests executed, again by risk area.

As you can see, the area of performance is still quite risky. We have found a large number of defects, and only a little more than one-third of the tests have been run. We should expect, as the remaining performance tests are run, to find lots more bugs. The area of interfaces is not a high-risk area, though. Notice that most of the tests have been run. Comparatively very few problems have been found.

Table 3-10. Example: Testing and risk

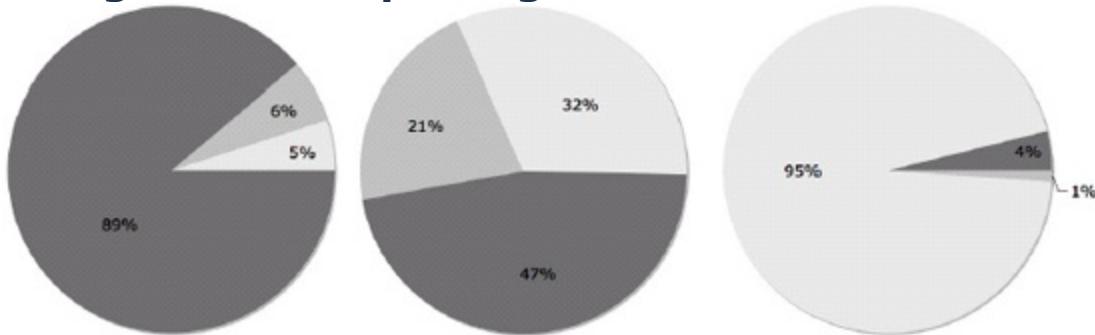
Quality Risk Areas	Defects		Tests		%
	#	%	Planned	Executed	
Performance	304	27	3, 843	1, 512	39
Security	234	21	1, 032	432	42
Functionality	224	20	4, 744	2, 043	43
Usability	160	14	498	318	63
Interfaces	93	8	193	153	78
Compatibility	71	6	1, 787	939	53
Other	21	2	0	0	0
Total	1, 107	100	12, 857	5, 703	44

For some people, the amount of details in [Table 3-10](#), whether shown tabularly or graphically, is still too much. One of our clients, when I told them about risk-based results reporting, said, "Look, that makes sense to me, but my boss, the CIO, won't pay any attention if there's more than one simple chart."

So, [figure 3-6](#) is a single, simple chart that captures the essence of this idea of risk-based test results reporting. The region in medium gray represents risks for which all tests were run and passed and no must-fix bugs were found. The region in light gray represents risks for which at least one tested has failed and at least one must-fix bug is known. The region in dark gray represents other risks, which have no known must-fix bugs but still have tests pending to run.

The timeline at the bottom shows that, when we start testing, the pie chart is dark gray. Over time, as test execution continues, we should see the light gray region eat into the dark gray region and the medium gray region eat into the light gray region. As we get toward the end of test execution, most of the dark gray and light gray should be gone.

Figure 3-6. Reporting residual levels of risk



At any time, for the medium gray region, we can produce a detailed list of risks that are tested, are known to pass the tests, and are not known to have any must-fix bugs. For the light gray region, we can produce a detailed list of risks against which known failed tests and must-fix bugs exist. We can list the tests and the bugs. For the dark gray region, we can produce a detailed list of risks and the tests still remaining to be run against them. This gives a clear picture of the risk reduction achieved and the residual level of risk.

So, let's wind down this section on risk-based testing with an overview of how risk-based testing fits into the entire fundamental test process.

During test planning, we will perform a quality risk analysis, including risk

identification and assessment. We will derive our test estimate and test plan from the risks.

As the rest of the testing process proceeds, we will adjust the quality risk analysis as part of test control.

During test analysis and design, we allocate the test development and executive effort based on the level of risk. We maintain traceability to quality risks to ensure alignment of our testing with risks, to properly sequence the tests, and support risk-based results reporting.

During test implementation and execution, we then sequence the test procedures based on the level of risk. We use exploratory testing to detect unanticipated high-risk areas, to offset the chance of missing a key risk.

As we reach the end of the test execution period, as we evaluate test exit criteria and report our test results, we will measure test coverage against risk and report test results (and thus release readiness) in terms of residual risk.

Finally, as part of test closure, we need to review the behavior of the system in the field or production. We should check to see if the quality risk analysis was accurate based on where we did and didn't see failures and based on where failures were particularly important or not so important. Lessons learned from this exercise should help us improve accuracy of subsequent quality risk assessments.

3.2.36. Risk-Based Testing and Failure Mode and Effect Analysis Exercise 5

Based on your informal risk analysis and list of activities, design risk-based test results reporting charts and/or tables that will communicate risk.

Sketch appearances of these reports toward the end of the project under good scenarios.

Sketch appearances of these reports toward the end of the project under bad scenarios.

If you are working in a classroom, break into groups of three to five. If you are working alone, you'll need to do this exercise yourself. If you are working in a classroom, once each group has finished, discuss the results.

I suggest allotting 60 minutes for this exercise, including the discussion.

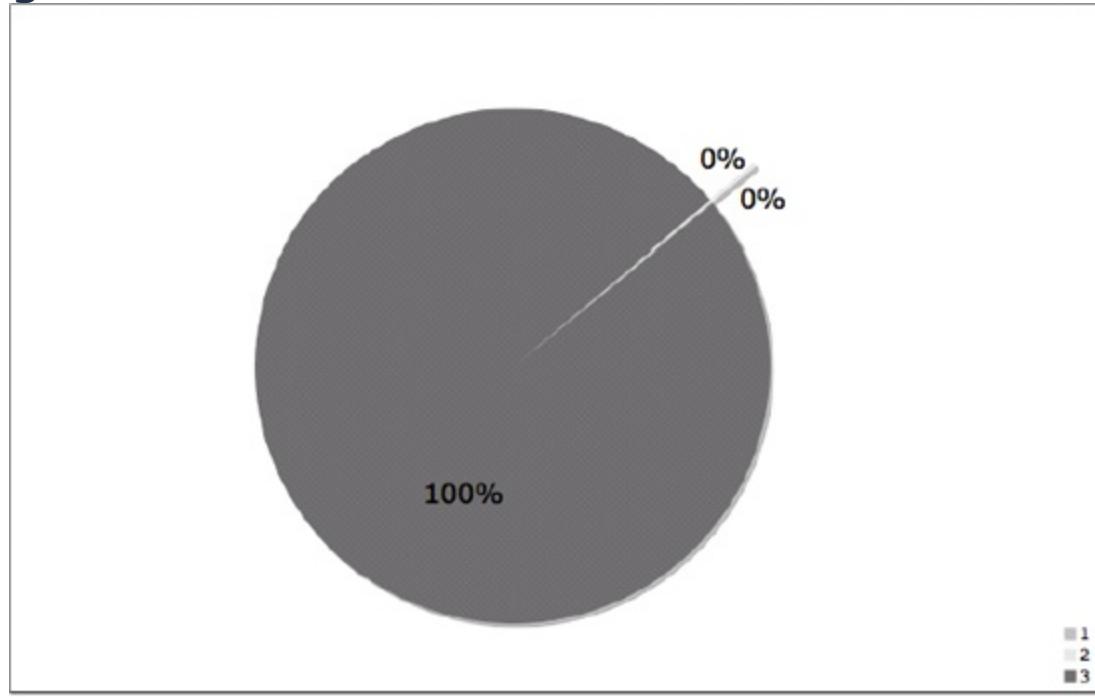
3.2.37. Risk-Based Testing and Failure Mode and Effect Analysis

Exercise 5 Debrief

I prefer the pie chart approach to showing remaining risk, at least to executive level audiences. The finer-grained tabular or histogram approach shows risk by quality category, which is useful for line managers and individual contributors.

For convenience, let me describe the pie charts in [figure 3-6](#) again. The region in medium gray represents those risk items that were identified in the quality risk analysis and for which all tests have been run and have passed and no must-fix bugs remain active. The region in light gray represents those risk items that were identified in the quality risk analysis and for which at least one test is still shown as failing and/or at least one must-fix bug remains active. The region in dark gray represents all other risk items that were identified in the quality risk analysis; for these risk items, there must not be any known failing test cases or must-fix bugs, but there are still tests pending to run.

Figure 3-7. Residual risk chart before test execution



[figure 3-7](#) shows the appearance of the pie chart prior to the start of test execution. As you can see, the entire chart is dark gray now, because we have not run any tests yet, nor have we found any must-fix bugs.

[figure 3-8](#) shows the appearance of the pie chart midway through test execution. Some risk items remain uncovered, almost two-thirds have been addressed through one or more tests. (Remember, risk items with a high level of risk are

likely to be covered by multiple tests, run in risk order, which is why the remaining uncovered risks, represented by the dark gray area, are not 50 percent of the risks.) A sizeable number of bugs and tests are pending resolution at this point, which is why almost half the area is light gray. However, if tests are run in risk order, and bugs are fixed in risk order, the very highest-level risks items should fall into the 20 percent medium gray area.

Now, as we test toward the end of the project, there are four typical scenarios: one good, three bad. Let's look at all four.

Figure 3-8. Residual risk chart midway through test execution

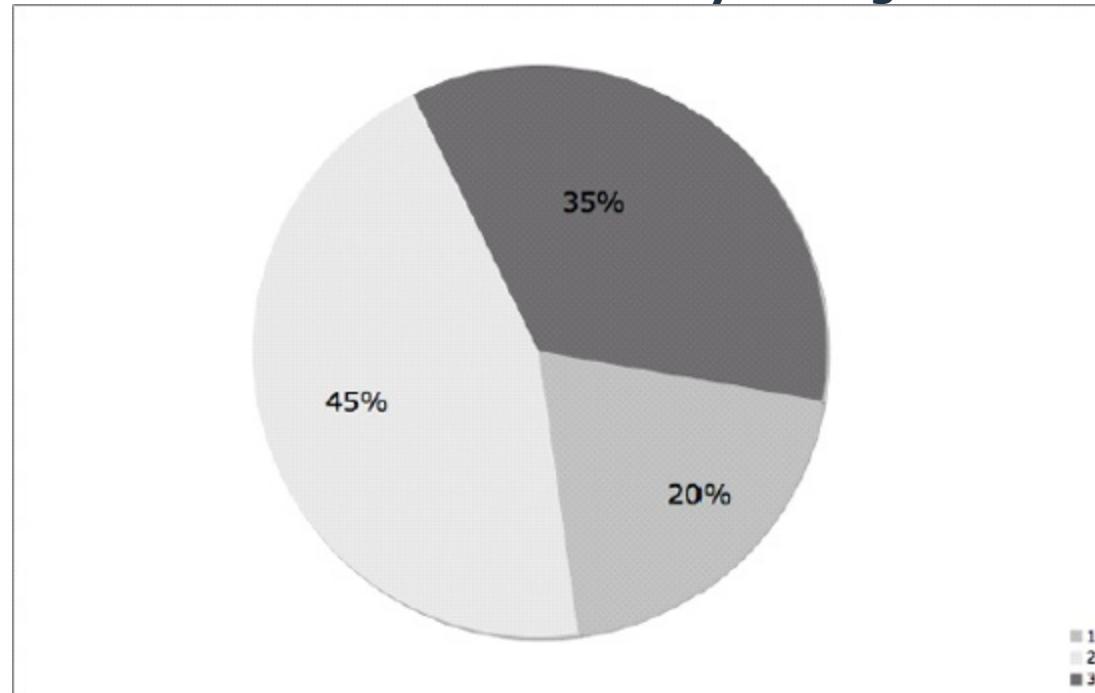


Figure 3-9. Residual risk chart approaching successful completion

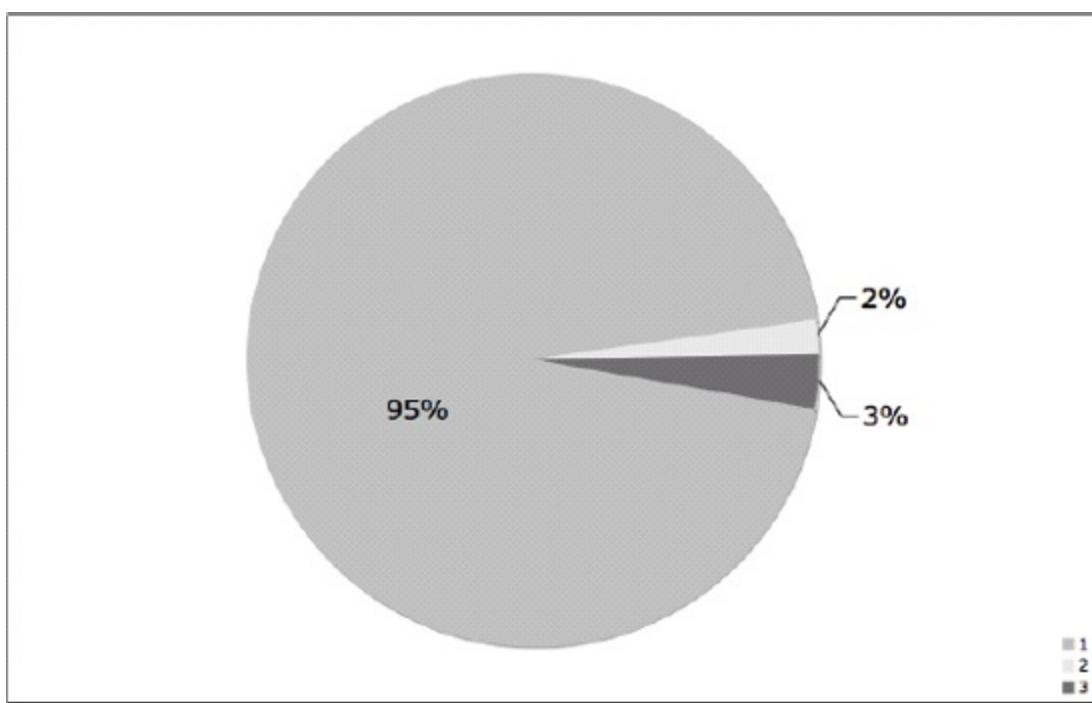
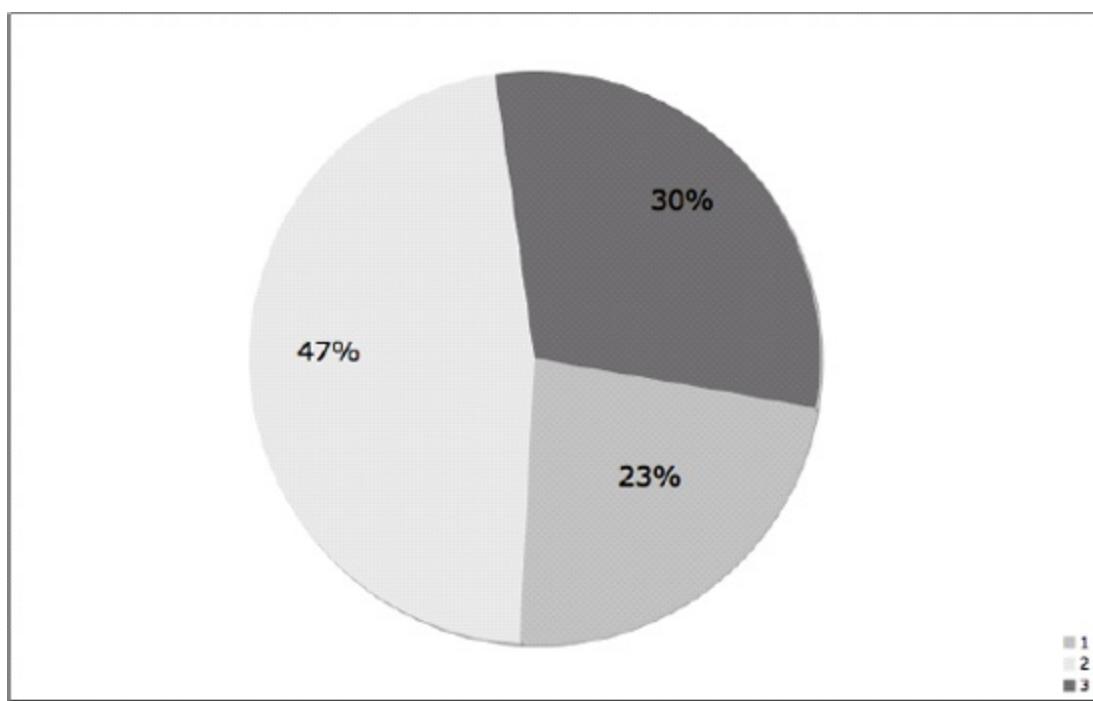


figure 3-9 shows the appearance of the pie chart as we approach a successful project completion. We have covered almost all risks items with tests, and all of those covered are passing and without known bugs. Just 2 percent of the risk items have known must-fix bugs or test failures, and only 3 percent are not yet tested. Remember, as mentioned in the course, at any point in time we can produce a breakdown of the risk items, failed tests, and known bugs for the light gray area and a breakdown of the risk items and unexecuted tests for the dark gray area. Because we should have run the tests and fixed the bugs in risk order, only the lowest-risk risk items, tests, and bugs should remain in the dark gray and light gray breakdowns. We can release at any time subject to management acceptance of the items identified in the light gray and dark gray breakdowns.

Figure 3-10. Residual risk chart at end of project with stalled testing



[figure 3-10](#) shows the appearance of the pie chart as we approach the planned end of the project, where the testing has stalled. The medium gray area is not much larger than that shown earlier at the middle of the project. The dark and light gray areas remain about the same as well. The light gray and dark gray breakdowns would be truly daunting lists of risk items, failing tests, and known bugs. Many behaviors can cause this, such as a lack of change management leading to endless feature creep, unstable design or code leading to constant introduction of new bugs, a loss of test staff or test environments, and failure of the development team to fix bugs.

[figure 3-11](#) shows the appearance of the pie chart as we approach the planned end of the project, where a massive failing test case and bug back-log exists. The medium gray area remains small. While the dark gray area is negligible, more than three-quarters of the risk items have known must-fix bugs or failing test cases. The light gray breakdown would list a large number of risk items, failing tests, and known bugs, many quite serious. Many behaviors can cause this, such as unstable design or code leading to constant introduction of new bugs and failure of the development team to fix bugs.

Figure 3-11. Residual risk chart at end of project with massive bug backlog

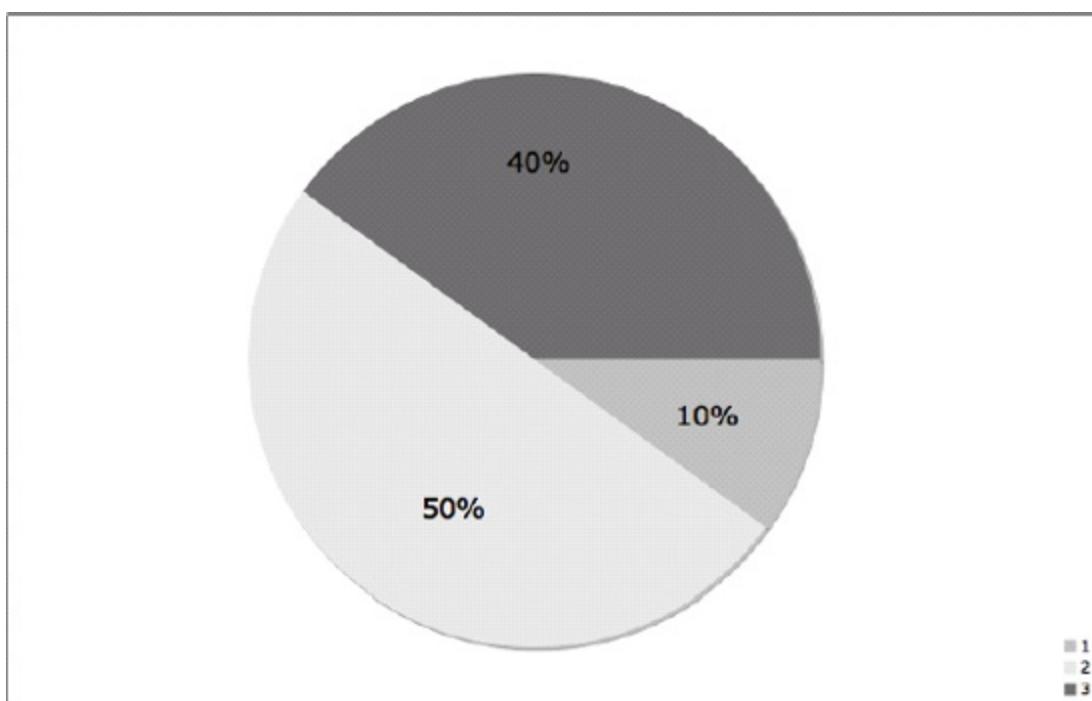


Figure 3-12. Residual risk chart at end of project with test backsliding

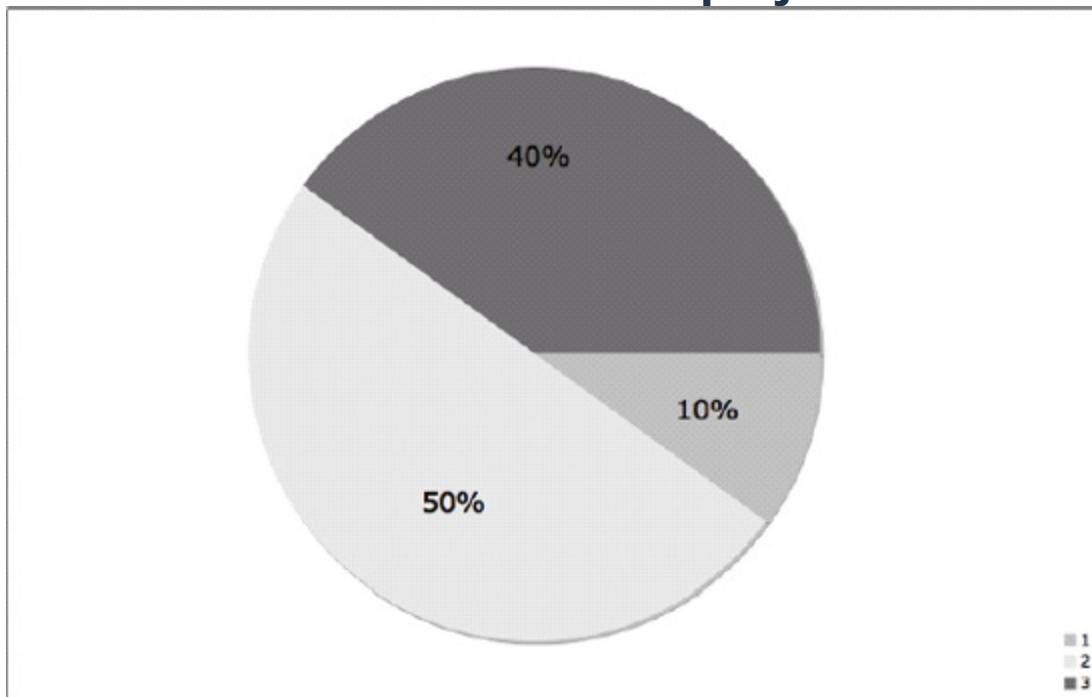


Figure 3-12 shows the appearance of the pie chart as we approach the planned end of the project, where we have seen test backsliding from the midpoint, as seen by a relative increase in dark and light gray areas with respect to the medium gray area. The dark and light gray areas are very large. The light gray and dark gray breakdowns would yield extensive lists of risk items, failing test and known bugs, and they would be lists that got longer, not shorter, since the midpoint of the project. Many behaviors can cause this, such as a lack of chan

management leading to endless feature creep, unstable design or code leading constant introduction of new bugs, failure of the development team to fix bugs and late discovery of many missing risk items in the initial quality risk analysis resulting in a significant increase in test scope.

3.3. Test Management Documentation and Test Plan Documentation Templates

Learning objectives

(K4) Outline test management documents such as the Test Plan, Test Design Specification, and Test Procedure Specification in compliance with IEEE 829.

(K2) Describe at least four important elements of a test strategy or approach and which IEEE 829 compliant documents contain elements of test strategy.

(K2) Illustrate how and why deviations from the test strategy are managed in the other test management documents.

(K2) Summarize the IEEE 829 structure of a master test plan.

(K2) Paraphrase and interpret the topics suggested by the IEEE 829 standard structure of a test plan with respect to tailoring to an organization, risk of a product, and risk, size and formality of a project.

While properly managing testing certainly involves more than creating documents, it is typically necessary to put some structure around the test process, and this typically involves documents. Creating test management documentation is a common duty—though perhaps some would call it a chore—but test professionals like us can find it hard to communicate about the documents that we create. It seems that everyone has a somewhat different name for these documents, and many times the same name refers to different documents in different organizations.

In the ISTQB Advanced syllabus, we try to structure this situation by defining four key management document types that we believe occur when organizations follow testing best practices.

Test policy: The document that describes the organization's philosophy, objectives, and key metrics for testing—and possibly quality assurance as well.

Test strategy (or test handbook): The document that describes the organization's general method of testing. This document would ideally be independent of any specific project but rather would lay out the approach to testing across most projects. This would include product and project risk management; the organization of testing steps, levels, or phases; and the

main activities that occur during testing, such as those described in the ISTQB fundamental test process. You might find this document name a bit confusing because we also talk about test strategies in the sense of a set of organizing principles for testing, such as risk-based analytical testing strategies and dynamic testing strategies. Notice that this document elaborates those organizing principles into an actionable—if general—methodology.

Master test plan (or project test plan, or test approach): The document that describes the application of the test strategy to a particular project. This would include the specific test levels that will occur and the relationship among those levels, including how gaps and overlap between the levels are to be managed and avoided. It would also address activities, like bug reporting, that are common across levels, unless those activities are already described in sufficient detail in the test strategy. Notice that, because different teams might own the different test levels, this can be a cross-functional document. Test managers are wise to work cross-functionally when creating such documents.

Level test plan (or phase test plan): The document that describes the specific activities for each test level. It expands on the master test plan for the specific level addressed, such as the system test plan, the integration test plan, etc.

Are you a bad test manager if you don't have all four documents, addressing exactly the topics described? No. You can combine these documents into one document or spread the information across multiple documents. In some cases, you might not document this information at all, though that's a risky approach for all but the smallest, shortest, and most transient projects.

3.3.1. Test Policy Documents

The test policy describes the "why" of testing for an organization, independent of means, projects, and test levels. A test policy describes the organization's testing philosophy. In some cases, it's even broader, describing the quality assurance philosophy, from the beginning to the end of the lifecycle. It should reflect the organizational values and goals as they relate to testing and perhaps also quality assurance.

Company management, not the test team, establishes the test policy. However, the test team or some other team within the information

technology, research and development, or product development department might draft the policy and submit it to management for approval, promulgation, and establishment. Like any policy that establishes direction, the test policy should be written.

The test policy can be a component in a broader quality policy. Alternatively, an organization can write and promulgate distinct quality and testing policies, but in that case, those documents should be complementary and well aligned.

A test policy document is not a long document. The best ones are short, about one to two pages. They are high-level documents. They briefly discuss the following topics:

- The objectives of testing, such as building confidence in the system, detecting defects, generating information, and managing quality risks.
- The organization's basic, generic, fundamental test process, following the ISTQB process or some other process.
- Various metrics for measuring the effectiveness and efficiency of testing, for all of the objectives defined earlier. For example, in terms of defect detection, the test policy might mention defect detection percentage as an effectiveness metric and the relative cost of defects detected in testing as opposed to after release as an efficiency metric.
- Quality targets, if the document is addressing quality assurance.
- Approaches for metrics-driven, experience-based improvements in the test process, such as checking the metrics for effectiveness and efficiency during a project retrospective and then applying Critical Testing Processes, Test Process Improvement, or the Testing Maturity Model to improve.
- New development and maintenance for large projects and small.

The audience for this document is not only the test team, nor only the technical people, but also the entire organization, from management on down. In the case of a product development organization, the organization might choose to make the testing policy accessible to customers and users.

As an RBCS consultant, I can say that the test policy document is more often missing than present. That creates real problems for test

organizations because there is a lack of agreement on what the test team is supposed to accomplish and no measurement toward goals

3.3.2. Test Strategy Documents

The test strategy describes, at a high level, independent of any specific project, the "how" of testing for an organization. The test strategy is the organization's methods of testing. The test strategy document can cover the options that the organization can use to manage product and project risks during testing. For example, what are the typical product risks that we try to address with testing, and how do we do so? How do those product risks relate to the high-level test objectives? What are the typical project risks that can affect testing, and how do we manage those risks?

A test strategy document can also provide more detail on the various testing levels or phases that your organization typically uses on a project. For example, it might discuss the general entry and exit criteria for the various levels. It could describe how unit testing, integration testing, and system testing are managed. In describing each test level, it might lay out the high-level test activities to be performed.

Since the test strategy provides the "how" details for the test policy's "why" statements, it must be aligned with the test policy. Further, it should provide generic test requirements for each test level and for the testing as a whole. For example, in a risk-based approach, it would specify the generic quality risk areas to be examined during quality risk identification and assessment.

A test strategy is not project dependent. It spans projects. That is, the document describes an approach to testing that applies to many if not all test efforts undertaken during a project. It also spans responsible groups, laying out the approach to take for all levels of testing across the different groups. For example, it can discuss unit testing, integration testing, and system testing and which group is responsible for each level.

A test strategy can provide various details on the test process. It might discuss the following areas:

- The integration procedures, such as top-down, bottom-up, functional, or risk-based
- The test specification techniques, including the test design concepts discussed in the companion volume for advanced test analysts as well

as the level of documentation detail required

- The independence of testing required, which may vary depending on level
- Any mandatory and optional standards to be followed in testing
- The various test environments used for different types of testing
- The application of test automation to various types of testing
- The need for and means for reusability of software work products and test work products, such as test cases, test data, and test plans
- The approach for retesting and regression testing, which might intersect with the test automation topic
- The means for performing and gathering information for test control and for reporting
- The test measurements and metrics to be gathered and frequency and means of communication
- How incident management is to be performed, including reporting of incidents, prioritization of incidents, and management of incidents
- How to carry out configuration management for the testware
- The relationship between groups performing testing and other groups on the project, and the deliverables that flow between those groups

These types of topics can be quite useful when projects are similar because having similar projects reduces the amount of project-specific documentation needed. A test team can simply refer to the test strategy for each project rather than redocument these issues in each test plan.

That said, it's certainly the case that different organizations and even different projects within the same organization might need different test strategies. A test strategy document is not about creating a "one size fits all" approach to testing but rather about creating an approach that can be tailored.

A test strategy can address both short-term and long-term needs. Because those needs could change, the test strategy should address the reasons

they might change and how those changes affect testing. The test strategies can change depending on the project and application. A security-critical or safetycritical application requires a more rigorous, intensive, and perhaps standardscompliant strategy.

Test strategy documents are more often missing than present. This means that people don't address some of the important topics mentioned here. Alternatively, those topics are addressed twice, three times, four times, or more, perhaps once for each project, with inconsistent decisions made each time. There's nothing wrong with rationally and deliberately tailoring the approach to each project, but having to figure out fundamental aspects of testing over and over again because no one can bother to document an overarching test strategy is inefficient, and, on many projects, reduces the effectiveness of the testing.

Let's look at a brief example of a test strategy document. [Figure 3-13](#) shows some key excerpts from the integration test strategy for a large system of systems project. On this project, we were testing a wide area network of interactive voice response (IVR) servers tied to a call center via the same wide area network. The integration of the call center servers and services with the IVR servers and services posed major risks.

We wanted to begin integration testing as quickly as possible to reduce these risks. Therefore, we can classify this as a preventive strategy.

Each build was called a "backbone". There were to be six backbones, starting with BB0 and progressing to BB5. To the extent possible, the systems and services added in each backbone would be done in decreasing risk order. In other words, we wanted to start with the highest risk systems and services.

Each backbone included the systems and services from the previous backbone. So, this is an incremental integration strategy, as discussed in the Foundation syllabus.

Figure 3-13. Example of test strategy in a test plan

Objective

Start integration testing immediately to accelerate integration defect discovery / removal and thus integration risk reduction.

Major Test Analysis, Design and Implementation Tasks

- Analyze the system design and risks to define the various backbones, which will consist of a subset of the servers and services, starting with the highest-risk backbone (BBO) and progressing to the lowest-risk (BB5).
- Define and implement revision-naming for all backbone systems.
- Define and implement bug tracking and resolution processes.
- Define and implement backbone test environments and support.
- Define test cycle duration.
- Design and implement the integration tests for each backbone, including automated test harness and test data.

Notice that the test strategy is based on an analysis of the system design and the risks associated with it; thus, it is an analytical test strategy.

As you can imagine, the test environments were complex and evolving. Therefore, skilled support teams were needed.

3.3.3. Types of Test Strategies

While there are many different test strategies, some written, some just oral tradition or institutional knowledge within a company, we can identify types of test strategies. Understanding these types can help you think more clearly about how you should select, blend, and devise your organization's test strategy.

You should recall from the Foundation syllabus that we can classify test strategies as either preventive or reactive. Preventive test strategies are those that focus on early involvement of the test team and use test planning, analysis, and design activities to identify problems in the project plan, the product requirements and design, and other project work products. This allows the organization to resolve these problems before test execution, thus preventing the late discovery of a large number of defects, which is a leading cause of project failure.

Reactive test strategies are those that focus on responding to the system as it is actually presented to the test team. These strategies allow the organization to avoid costs associated with development of test work

products that might not be useful if the actual system differs from what was initially planned.

As you can see, for projects that are predictable, carefully planned, and executed according to plan, the preventive strategies have the advantage. Why perform all the test planning, analysis, and design in parallel with test execution—or perhaps not at all—if you can do this work earlier in the project?

For projects that are chaotic, poorly planned, or in constant fundamental change, the reactive strategies have the advantage. Why create a bunch of test work products that will have to be maintained or completely re-created when the system delivered turns out to be a total surprise?

Now, this classification of a strategy as preventive or reactive is a simplification. In actuality, a given test strategy will fall somewhere along a spectrum and will often incorporate both preventive and reactive elements.

In my book *Pragmatic Software Testing*, I surveyed the types of test strategies that we find in actual use. That survey is the basis of the list of test strategies in both the Foundation and Advanced syllabi, by the way.^[1] Let's look at the various types of test strategies that exist.

^[1] Specifically, I had a discussion in a newsgroup with Ross Collard, Cem Kaner, and Kathy Iberle, among others, which lead to my list of types of test strategies. I use this list in my course materials for "Pragmatic Software Testing", which were the source of my book *Pragmatic Software Testing* as well as for the test strategy lists in the Foundation and Advanced syllabi.

In analytical strategies, such as risk-based testing, the test team analyzes a test basis to identify the test conditions. The results of the analysis form the basis of the test effort. Analytical test strategies tend to be thorough, excellent for quality risk management, and good at finding bugs. Analytical test strategies are primarily preventive and require significant early-project time and effort.

In model-based strategies, such as those relying on operational profiles, you design, implement, and execute your tests based on models of system behavior. If the models capture the system's essential elements, your tests will be good. So model-based strategies rely on the ability of the tester to develop good models. These strategies fail when the models, or the testers building the models, fail to capture the essential or risky aspects of the

system.

In methodical strategies, such as those that follow checklists of quality characteristics, you follow some standard catalog of test objectives. Methodical strategies are lightweight and effective for stable systems and for systems that are similar to ones you've tested previously. However, significant changes will make these strategies ineffective, at least unless some adjustment to the catalog of test objectives occurs.

In process- or standard-compliant strategies, such as those based on IEEE 829, you follow some process or standard promulgated by a standards committee or group of wise people somewhere. These strategies can save you the time and energy of devising your own methodologies. However, if the overall mission of testing that you have is different from the ones that the creators of the standard know, or if the test problems you are struggling with are not those that the creators of the standard have resolved, then you'll find that this borrowed approach fits about as well as a borrowed tuxedo.

In dynamic or heuristic strategies, such as those based on bug taxonomies or software attacks, you use general rules about how software fails; lists of important software areas, features, and behaviors; or lists of common software data structures and operations to make educated guesses to focus the testing. These test strategies will minimize structure, maximize flexibility, and, typically, focus on finding bugs (rather than building confidence or reducing risk). The lack of structure and documentation means that they lack detailed coverage information and do not systematically reduce quality risks or incorporate any preventive elements. These weaknesses aside, using a purely dynamic strategy is better than not testing at all. Better yet, you can blend dynamic strategies with analytical strategies, providing a good way to check for gaps in the analytically designed tests.

In consultative strategies, such as those where users or programmers determine what is to be tested, the test manager trusts that some other group of people knows best what should be tested. The test manager asks them for the test conditions to cover and covers those conditions. The other group also acts as the primary test oracle, in that we'll ask them to determine the expected results. This strategy can be appropriate when the test team is brought in primarily to serve as the hands and eyes of another group, such as with some types of outsourced testing, but generally, it is a poor substitute for professional testing.

Finally, in regression testing strategies, such as those that rely on extensive functional test automation, we focus on test repetition and smart test selection to try to minimize the risk of breaking something that already works. For stable systems that are changing only slowly, these strategies can make sense. However, there is always the risk that new features will be poorly tested and, should the rate of change accelerate, the test coverage for recently added features will be left behind. In addition, even given a good approach for test automation, a rapidly increasing automated regression test set can overwhelm the abilities of the test team to run and analyze all the test results.

As with the classification of strategies as preventive or reactive, remember that they exist on a continuum. Also, remember that strategies are not philosophies or religions—you don't have to choose just one and stay with it. Strategies may be combined. It is smart to spend some time as a test manager carefully selecting and tailoring your test strategies. You can adopt general strategies for your test team and modify or completely rework them for a particular project too.

3.3.4. Test Plan Templates

Both the Foundation syllabus and the Advanced syllabus discuss the IEEE 829 standard for test documentation. Some templates in this standard are useful for test plans and other related documents. So let's review those templates, but before we go through the specific templates, let's discuss the smart use of templates.

There are two major risks associated with the use of templates for testing (and for just about anything else). The first is that people might use a template as an excuse to turn their brains off. In other words, they focus on filling in blanks in templates rather than the test planning, test design, or test implementation tasks they are trying to accomplish. The smart use of templates is as a spur to thinking, not an impediment to thinking.

The second risk is the "one size fits all" problem, where organizations do not properly adapt the templates for their particular needs or where project teams do not further adapt the templates for their projects. This problem can lead to the templates leaving out key topics as well as including unimportant topics. Test planning should be a planning exercise—not a documentation exercise.

When used properly, the content and structure of the test plans and other

test documentation will vary. Different organizations will do things differently. Internal documentation standards will influence the templates and documents. Risk, formality, and regulations that apply to a project can drive the types of documentation and the level of documentation.

It helps to have common document structures across organizations because that helps readability and communication. Common organizational templates can help. Smart people then can use those templates to spur thinking about testing issues.

A test plan is a subproject plan for all or some of the testing parts of a project. In a moment, we'll look at the IEEE 829 test plan template. You can adapt the IEEE 829 template for each detail test plan as well as for the master test plan. If you don't have a requirement to conform to IEEE 829, you can create your own template or outline. However, the IEEE 829 template can provide some useful inspiration as you do that.

You should not and, practically, cannot write a test plan in a vacuum, separate from the other realities of the project. Test planning and the resulting test plan will be influenced by various considerations. These consideration include the test policy of the test organization; the scope of testing; the objectives of testing and the project; the project and quality risks; constraints related to feature, schedule, and budget; the critical areas of the system under test; the testability of the system under test; and the resources available. At the same time, properly used, the plan itself is a tool to influence these same considerations.

An IEEE 829 test plan includes the following sections:

- Test plan identifier
- Introduction
- Test items (those things that will be delivered for testing)
- Features to be tested
- Features not to be tested
- Approach (including strategies, organizational issues, and the extent of testing)
- Item pass/fail criteria (which I find a rather odd section because I don't

think of test *items* as passing or failing but rather of features or attributes as passing or failing)

- Test criteria (for example, entry criteria, exit criteria, suspension and resumption criteria)
- Test deliverables (such as reports, charts, and so forth)
- Test tasks (which I typically combine with the "Schedule" section and list the key testing milestones)
- Environmental needs
- Responsibilities
- Staffing and training needs
- Schedule
- Risks and contingencies (can include quality [or product] and project risks, though I usually discuss those in separate sections)
- Approvals (if you need sign-off)

For ease of reference, the main headers of the sections in the template are shown in [figure 3-14](#).

Figure 3-14. IEEE 829 test plan template

Test plan identifier

Introduction

Test items

Features to be tested

Features not to be tested

Approach

Item pass / fail criteria

Test criteria

Test deliverables

Test tasks

Environmental needs

Responsibilities

Staffing and training needs

Schedule

Risks and contingencies

Approvals

3.3.5. The Other IEEE 829 Templates and How They Relate to Test Planning Documents

While typically created during test design, implementation, and execution, four other templates relate strongly to the test plan. Each covers topics initially discussed in the test plan and might elaborate on those topics. Certainly, we would want to ensure consistency.

The first of these templates is the IEEE 829 test design specification. The test design specification describes a collection of test cases and the test conditions they cover at a high level. This collection of test cases is sometimes called a test suite. The IEEE 829 test design specification template includes the following sections:

- Test design specification identifier
- Features to be tested (in this test suite, which should be a subset of the features to be tested from the test plan)
- Approach refinements (where we elaborate on the approach described in the test plan, discussing the specific techniques, tools, and so forth needed for this test suite)
- Test identification (which traces to test cases in suite)
- Feature pass/fail criteria (e.g., test oracle, test basis, legacy systems, etc.)

For ease of reference, the main headers of the sections in the template are shown in [figure 3-15](#).

Figure 3-15. IEEE 829 test design specification

Test design specification identifier

Features to be tested

Approach refinements

Test specification

Feature pass / fail criteria

The sequencing of test suites and cases within suites is often driven by risk and business priority, which are often discussed in the test plan. Project constraints, resources, and progress affect the sequencing, and are also covered in the test plan.

Next, we come to the IEEE 829 test case specification. A test case specification describes the details of a test case. This template includes the following sections:

- Test case specification identifier
- Test items (what is to be delivered and tested, which should be a subset of the test items in the test plan)
- Input specifications (user inputs, files, and so forth)
- Output specifications (the expected results, including screens, files, timing, and such)
- Environmental needs (hardware, software, people, and props, again a subset of the test plan's environmental needs)
- Special procedural requirements (such as operator intervention or permissions)
- Intercase dependencies (if needed to set up preconditions)

While this template defines a standard for contents, the question of what is a test case is certainly an open one. In practice, test cases vary significantly in effort, duration, and number of test conditions covered. The test plan should clarify these issues.

For ease of reference, the main headers of the sections in the template are shown in figure 3-16.

Figure 3-16. IEEE 829 test case specification

Test case specification identifier

Test items

Input specifications

Output specifications

Environmental needs

Special procedural requirements

Intercase dependencies

Next we have the IEEE 829 test procedure specification. A test procedure specification describes how to run one or more test cases. This template includes the following sections:

- Test procedure specification identifier
- Purpose (such as which tests are run)
- Special requirements (skills, permissions, environment, etc.)
- Procedure steps (logging, setup, start, proceed [steps themselves],

measurement of results, shutdown/suspension, restart [if needed], stop, wrap-up/teardown, contingencies)

For ease of reference, the main headers of the sections in the template are shown in [figure 3-17](#).

While the IEEE 829 standard distinguishes between test procedures and test cases, in practice test procedures are often embedded in test cases. In other words, the same document describes the actions to take, the inputs and other data to use, and the expected results to be verified. However, for automated tests, separating the scripts from the inputs and the expected results is important, as we'll discuss later in this course.

A test procedure is sometimes referred to as a test script. A test script can be manual or automated.

Figure 3-17. IEEE 829 test procedure specification

Test procedure specification identifier

Purpose

Special requirements

Procedure steps

Another template in the IEEE 829 standard for documentation that relates to the test plan is the test item transmittal report. Unique among the IEEE 829 templates, someone other than a tester almost always should create this document. Generally, the release engineering team or its equivalent will create this document and submit it to the tester with the test object.

Naturally enough, the test item transmittal report describes the items being delivered for testing. The IEEE 829 template for test item transmittal reports includes the following sections:

- Test item transmittal report identifier
- The transmitted items (including item names and revision numbers,

which should relate back to the test items described in the test plan)

- The location of the test object or item, including where it is, where it came from, what media are being delivered, how they are labeled, and any other information needed to unambiguously identify the test object or item to be picked up and deposited in the test environment
- The status of the test object or item, including the bugs fixed, the changes introduced, and so forth
- Finally, if the test object or item is under change control, the name of the person or persons who approved that object or item for release to testing

For ease of reference, the main headers of the sections in the template are shown in [figure 3-18](#).

Test item transmittal reports are sometimes called release notes. Often, release notes include only some of the information mentioned above. They are often informal documents. The test plan should clarify what information the release engineering team must supply with each delivery of test items and how the items are to be delivered. Otherwise, you stand a strong chance of not getting the information you need and even of not getting the items properly delivered.

Figure 3-18. IEEE 829 test item transmittal report

Test item transmittal report identifier

Transmitted items

Location

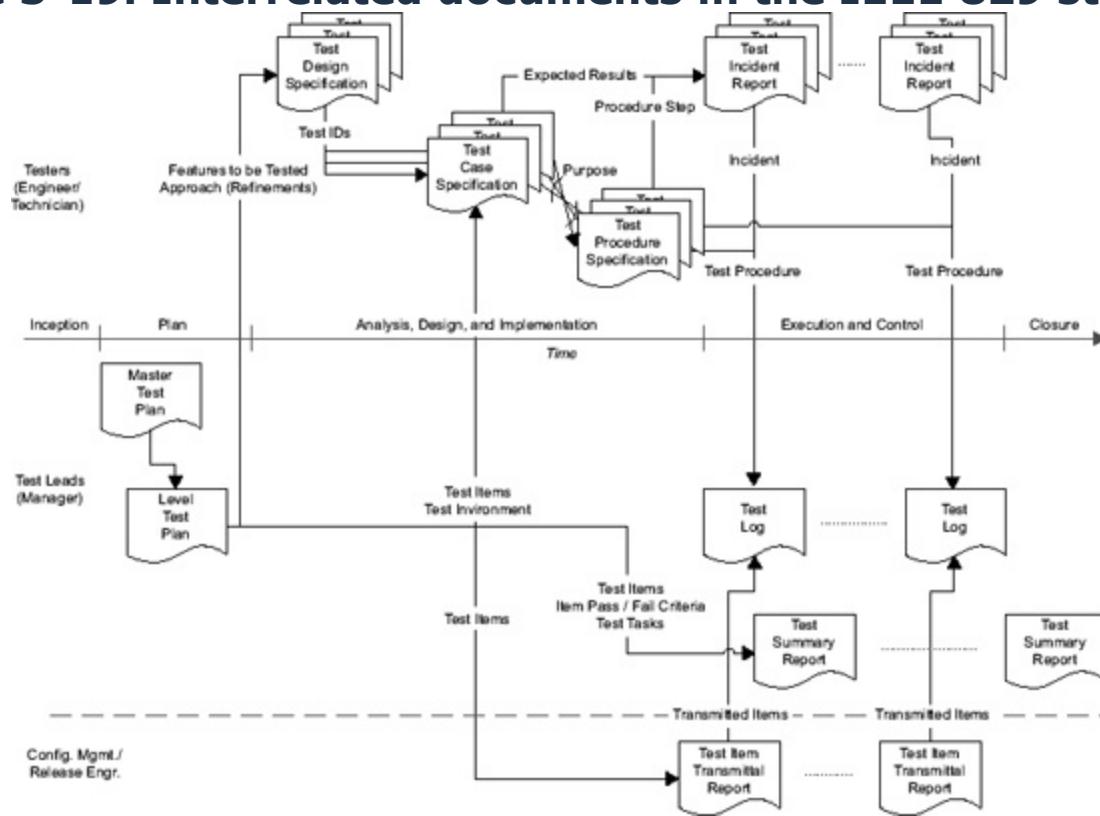
Status

Approvals

Because the IEEE standard contains a number of interrelated documents,

let's look at an overview of it. I suggest that you spend a few moments studying [figure 3-19](#) before you read my description of it in the coming pages.

Figure 3-19. Interrelated documents in the IEEE 829 standard



In the middle of the figure, I've shown the project timeline. It starts with project inception; then planning; then analysis, design, and implementation; then test execution and control; and finally test closure. This timeline includes the steps of the fundamental test process described in [Chapter 2](#). This timeline places each IEEE 829 document in the context of a typical project, with the testing following the ISTQB fundamental test process. On the left side of the figure, you see the role usually associated with producing the work product shown to the right.

ISTQB Glossary

level test plan: A test plan that typically addresses one test level.

During the planning period of the project, the test leads will produce the master test plan and one or more detail or level test plans. In a risk-based test approach, the plans will follow from the risk analysis.

Now, the testers will analyze the requirements, the design, and the risks. They will elaborate on the features to be tested, identified in the test plans. One or more test design specifications will come from this analysis. As the testers design test cases, they will link those back to the test design specifications in each test case specification. Using test case identifiers, as shown earlier in this chapter, will enable that linkage. The test case specifications will include information on the test items and the test environments to be tested. The test case specifications will also give expected results for the test cases.

Based on the test case specifications, the tester will implement test procedure specifications. These include specific test procedure steps. The purpose section of the test procedure specification links the test procedure to one or more test cases. One or more test procedures can involve the use of each test case, and one or more test case can be associated with a given test procedure. As noted previously, this distinction between test case and test procedure, while important in the case of automated tests, is frequently ignored when writing manual test cases.

When test execution begins, we will receive a test object and, hopefully, from the release engineer, a test item transmittal report for each test item. Each test object released to the test team should be described by a test item transmittal report.

As the tester runs tests, they will log test results in the test logs. The test logs note the test procedures being run. In running tests, testers usually report incidents. A number of test incident reports can be filed in a given

test execution period. The test incident reports are linked to the specific test object being tested. The test incident reports link back to the test case specification in terms of expected results. The test incident reports link back to the test procedure specification in terms of test procedure being run. In some implementations of the standard, the linkage would be to the test case. Of course, where the test procedure and the test case are the same document, the linkage is effectively to both.

During test execution, the test manager and the test analysts will not only create the test logs, but also produce one or more test summary reports. The information in the test logs link into the test summary reports. The test summary reports also link back to the test plan based on the test items, the item pass/fail criteria, and the planned and actual test tasks. During test closure, the test manager might produce a final test summary report that summarizes the results of the entire test execution period.

You can expect to find that this IEEE 829 documentation standard will be important on ISTQB Advanced exams. I recommend that you review [figure 3-19](#) until the entire standard is clear to you.

3.3.6. Master Test Plan Documents

With the IEEE 829 test plan template and its related templates clearly in mind, let's discuss master test plans and detail test plans further. The master test plan describes the "how" of testing for a single project, spanning multiple levels of testing. It describes how we intend to apply the test strategy to a specific project. When the testing is decomposed into multiple levels, such as unit test, integration test, and system, the master test plan should describe the relationship between these levels and the division of work across the levels.

When writing a master test plan, you should align the master test plan with the test policy and strategy. However, in some cases, the master test plan will deviate from policies or strategies for specific reasons. You should explain those reasons in the master test plan.

Across the project or operation that the testing serves, we want to ensure that the master test plan complements the project plan or operations guide. It should describe the testing effort as part of the larger project or operation. It should ensure that testing is organized in a way that serves the purposes and goals of the project or operation as well as the broader organizational objectives for testing.

As mentioned in the previous section, the specific content and structure of a master test plan can vary. However, most of the good ones I've seen and written tended to include those topics from the IEEE 829 template, though often under different names.

It's important to clarify the scope of the test work in test plans, so I like to see a section that addresses this topic, in terms of both items to be tested and not to be tested and quality attributes to be tested and not to be tested. The test plan should address the testing schedule and budget, and those should be consistent with the project or operational budget. Again, when a project involves multiple test levels, make sure this definition of scope is clear for each level described.

Because miscommunications and mistakes often occur when trying to coordinate the delivery of test items, the execution of test cycles, and the ultimate completion of the test cycles, the test plan should describe in some detail the test execution cycles and their relationship to the software release plan. Under what circumstances are we ready to start a level of testing, under what circumstances might we suspend and resume a level of testing, and under what circumstances are we ready to declare a level of testing completed? In general, foul-ups are always more likely during hand-offs across the project team, so the test plan should discuss relationships and deliverables that will pass between testing and other people or departments.

The test plan should address directly or by reference the organizational objectives of testing, the business justification for and value of the work to be done. What are the product risks that we are covering with testing? What are the project risks that might affect testing?

In some cases, the master test plan and detail test plan can be a single document. This is common if only one level of testing is a formal level. For example, if we have a formal system test performed by an independent test team, but we have informal component and integration testing performed by programmers and informal acceptance testing performed by customers under the name of a beta program, then the system test plan is probably the only written test plan. So it would need to cover the topics mentioned earlier.

Testing is downstream of most other activities of a project. This means that chaos and disorder in other areas of the project can leak into the testing effort if we're not careful. If during test planning, you identify a situation

where disorder elsewhere in the project will affect your testing, then your master test plan or a detail test plan might need to fill the gaps in other project documents. For example, if the configuration management process is not documented, you can expect to have challenges with test item delivery, test item transmittal reports, revision numbering, and other related release engineering topics. The test plan should specify how test objects will be delivered to the test team.

3.3.7. Detail Test Plan Documents

The detail test plan describes, for a single test level on a single project, the "how" of testing. The detail test plan will describe the particular activities for a test level. Theoretically, there would be a detail test plan for each level of testing, and it would expand on the master test plan for that level. The detail test plan provides level-specific schedule, task, and milestone details. If there are level-specific standards and templates, then the detail test plan should cover that.

Now, though theory calls for a master test plan and then four or more detail test plans, one for each level, when working on an informal project or operation, you might find that a single detail test plan is the only test management document actually written down. If that's the case, it has to address the topics that would theoretically be covered in a master test plan—and potentially even in the strategy and policy documents.

3.3.8. Case Study of a Test Plan and Deviations from It

[figure 3-20](#) shows an example outline of an IEEE 829-like test plan, from the interactive voice response server project I've mentioned a few times. These are the major section headers from the plan.

The plan covers two levels of testing, preintegration testing and integration testing. On this project, preintegration testing corresponded to system integration testing, as defined in the ISTQB syllabi and glossary. Integration testing corresponded to system testing of the system of systems, which I discussed earlier in [chapter 1](#).

You can see the sections that define the various entry, stopping, and exit criteria. It was important to ensure coordination across these two levels of testing.

There were other levels of testing. Formally, there were system test levels for each of the systems included in the system of systems. There were also

informal unit test levels and component integration test levels for these systems, owned by the development teams. Some of those test levels ended up not happening, or at least not being done very thoroughly, which led to the late discovery of many bugs.

Figure 3-20. Example of major sections in a test plan

<ul style="list-style-type: none">• Overview• Scope, Definitions and Setting<ul style="list-style-type: none">▪ Scope▪ Definitions▪ Setting• Entry and Exit Criteria<ul style="list-style-type: none">▪ Pre-Integration Test Entry Criteria▪ Pre-Integration Test Stopping Criteria▪ Pre-Integration Test Exit Criteria▪ Formal Integration Test Entry Criteria▪ Formal Integration Test Stopping Criteria▪ Formal Integration Test Exit Criteria	<ul style="list-style-type: none">• Quality Risks• Schedule of Milestones• Test Environments<ul style="list-style-type: none">▪ System Revisions▪ Backbone Content Tracing▪ EPCOT and Labs• Test Development• Test Execution<ul style="list-style-type: none">▪ Human Resources▪ Test Hardware▪ Test and Issue Tracking and Management▪ Issue Isolation and Classification▪ Test Cycles• Change History• Referenced Documents
--	---

Test environments for system of systems projects tend to be complicated, and this project was no exception. You can see that we had three major topics to address in this area:

- The System Revisions section addresses the fact that we were assembling the test environments—which were the ultimate production environments—during the test execution period, on a just-in-time basis. (In some cases, of course, it ended up being on a not-quite-in-time basis, which I'll discuss in a moment.) So, what versions of the environments will be available, and when? The plan addresses those questions here.
- Backbone Content Tracing refers to which specific software and system revisions would be included in each integration build, referred to here as a backbone. Again, because things were constantly changing, we needed to make sure we knew what was being tested.
- Finally, the EPCOT and Labs section covers the infrastructure and facilities. EPCOT was a clever name we gave to the test environments

as they came together because it was supposed to show us what tomorrow would look like.

Now, there are plans, which are good, and then there's what ends up happening on projects. It's a good idea to keep track of deviations from the plans. The IEEE 829 standard suggests using test logs to track events that lead to deviations from the plans.

On this particular project, I used a simple table in an Access database as my log. [figure 3-21](#) shows one page of a 27-page report that showed the various deviations we had from the plan. I've redacted some information to protect the innocent, as the saying goes—along with some who were not so innocent.

Figure 3-21. Example: Test plan deviations

Project Impact	Name	Noted	Change Description	Impact Description	Schedule Impact	Resource Impact	Other Costs/Saving
8/4/1998	Integration	7/31/1998	We are going to lose Beta to Alpha , so they can wrap up their QVS API development and testing.	If we get the CDR server set up as the cooperating host for stress and capacity, the impact should be minimal. If not, we will take a serious hit. Also, integration may get hurt if baby-alpha gets on the critical path.			
8/4/1998	Integration	8/1/1998	The latest—and supposedly final—plan for Beta is that it "will not" return to Alpha , but will remain here. Team Alpha will come up from Team Beta the week of 8/10 to make the "production ready" and to train the network team in the operation of the unit.	None, really. Having Beta out of commission for a couple days doesn't matter, because we're in the post-BB0/pre-BB1(2) (unplanned) hiatus.	0	0	\$0
8/10/1998	Integration	9/10/1998	TIBCO licenses expire frequently. It was hourly, then daily, now it's weekly.	When this happens, tests die and must be restarted. It's probably chewed up twenty or so test hours, and maybe ten or so person-hours. It also reduces confidence in test results.	3	13	

The first two line items have to do with the impending loss of a hardware resource named Beta. Beta was one of the IVR servers, appropriately enough one of the early models. There were also models called alpha and baby alpha, but we didn't use these for testing because they did not have the full call capacity required.

Ultimately, you can see that Beta did not go away, but it did receive an upgrade. The upgrade fell in a rare break in the testing action, so it did not interfere with our work.

The third line item discusses test problems that arose from TIBCO license issues. TIBCO provided some of the middleware on this project. Logistical and setup issues led to license problems, which negatively affected testing.

These kinds of logs are very useful when it comes time to explain to project managers and other project stakeholders the kinds of events that are

occurring and affecting your ability to test. You can use a Word document, but I prefer an Access database or Excel file. Access or Excel allow you to calculate the impact, in schedule, resource, and other costs, of various changes that have affected your testing.

3.3.9. Another Case Study, of a Master Test Plan

figure 3-22 shows an outline of an IEEE 829-like master test plan, this time from the Internet appliance project I've mentioned a few times. It addresses both hardware and software testing, with separate sections addressing the quality risks in each area. There was a detail test plan for the hardware testing. The details for the software testing were incorporated in this master test plan.

Figure 3-22. Example of a master test plan

- | | |
|---|--|
| <ul style="list-style-type: none">• Overview• Bounds<ul style="list-style-type: none">▪ Scope▪ Definitions▪ Setting• Quality Risks<ul style="list-style-type: none">▪ Hardware▪ Software• Transitions<ul style="list-style-type: none">▪ Integration Test Entry Criteria▪ System Test Entry Criteria▪ System Test Continuation Criteria▪ System Test Exit Criteria | <ul style="list-style-type: none">• Schedule• Test Configurations and Environments• Test Development• Test Execution<ul style="list-style-type: none">▪ Human Resources▪ Test Case and Bug Tracking▪ Bug Isolation▪ Release Management▪ Test Cycles• Risks and Contingencies• Change History• Referenced Documents |
|---|--|

The software testing included two levels, integration test and system test. Integration test was less formal, so while we had entry criteria for it, we did not have continuation criteria or exit criteria. Integration testing would be done when all the integration tests were over. System testing did have formal entry, continuation, and exit criteria.

3.3.10. A Detailed Case Study, of a Test Plan for a Simple PC Application

We've reviewed test plans at a high level in the previous parts, looking at outlines and major topics. For the next few moments, let's take a closer look at selected sections of an IEEE 829-like test plan. This plan addressed

testing of a simple PC application. The application was designed to allow comparative PC performance testing during PC evaluation and purchase. It resided on a dedicated USB drive.

RBCS served as the test team for this project. We were performing what we called a component test on the user interface of the application. It was a relatively short, low-cost, low-risk test project, so formalism and rigor is low, as you'll see in the plan. In this case study as before, the non-RBCS names have been changed, again, to anonymize the client and the participants.

This first excerpt shows the resources section. In [Table 3-11](#), we define the test team, plus the client contact. Given the simplicity of the project, the only outside interaction the test team had was with Quincy.

[figure 3-23](#) shows the test execution process section of the test plan. Before I describe the key ideas, I suggest that you read the figure completely first.

Notice that in the first sentence I established the objective of testing. This project has a single objective, that of finding bugs. Because this is a one-off project, not an ongoing test team, I put this statement, which would otherwise occur in a test policy document, here in this test plan.

I also defined our strategy briefly here, which is a combination of consultative testing (the client supplied the test charters), object-based analytical testing (analyzing the use cases the client provided to identify test conditions), and reactive (using exploratory testing). We defined the receipt of releases, the test cycles, and the mapping of test cases to test cycles in this paragraph as well.

Table 3-11. Case study resources

Title	Roles	Name
Test Manager	Plan, track, and report on test execution. Conduct end-of-day debrief, status update, and results posting to Client Contact and test team. Secure appropriate resources. Review bug reports and test results.	Rex Black
Lead Test Engineer	Execute tests. Guide the other test engineers. Act as Test Manager when needed. Review bug reports and test results.	Nabi Berri
Test Engineers	Execute tests. Report results to the Lead Test Engineer or Test Manager as needed. Review bug reports and test	Amit Harare

	results.	Laurel
Test System Admin	Ensure proper configuration of all the test environments.	Becker Maxim
Client Contact	Deliver test release. Receive test results. Guide resolution of bugs for Phase II. Act as liaison for any blocking test issues not resolvable by the test team.	Quincy Adams

Figure 3-23. Case study test execution process

The objective of the [testing project] is to find bugs in the [application] user interface. The RBCS test team intends to do so by running a set of test charters and exploratory tests against two test releases, one release for each of the two Component Test Phases. The test team shall run every test case exactly one time as part of each of the two Component Test Phases.

To track the testing, the test team shall use ... Test Case Summary worksheets ... At the beginning of each test cycle, the Test Manager... .shall assign a "basket" of test cases to each tester. In Phase II, the Test Manager... .shall attempt to assign a different basket to each tester, to minimize the risk that any invalid assumptions on one tester's part result in a test escape. Once assigned their basket of tests, these testers shall follow the process described in the rest of this section to execute each test case, and repeat the process until they have exhausted their list.

If they empty their basket prior to the end of a cycle, they are to assist other testers by coordinating a reassignment of some of those testers' test cases to themselves.

The next paragraph describes the test assignment and tracking processes along with the process for the tester to run the tests. Notice that workload balancing is covered as well.

Table 3-12 shows an example of the use cases and test charters supplied by the client. Again, I recommend reading the table before proceeding.

Table 3-12. Case study use case and test charter

Name:	Test Current Computer
User Goal:	To calculate and see True Performance Scores for the computer into which [the application USB drive] is currently plugged.
Starting Point:	Test Current Computer can be initiated from any application state.

Path:	User clicks Test in the Primary Function area. The Animation runs in the Content area while the [performance test] executes and the True Performance Scores display is created. As soon as the True Performance Scores display is created, the Animation stops and the current computer's True Performance Scores are displayed in the Content area.
Ending Point:	The current computer's True Performance Scores are displayed in the Content area.
Notes:	Clicking Test always causes the [performance test] to execute and a new True Performance Scores display to be built, even if this scenario has already been executed one or more times for the current computer.
Test Charter:	Verify the Path from Starting Point to Ending Point, including the specified error conditions.

As you can see, there are many opportunities for test design and exploratory testing here. For example, the use case says that this use case can be initiated from any application state. Hmm, how many different states are there? Have we tested them all?

Notice too how concise the test charter is; this gives the tester many different ways to explore the use case.

[Table 3-13](#) shows the schedule of milestones. Again, I suggest you read this table before proceeding.

Table 3-13. Case study schedule of milestones

Date	Milestones
9/30	Delivery of the Test Charters to RBCS.
10/7	Delivery of the System Test Plan for the AMD DIPstick Test Project by RBCS.
10/13	Delivery of the First Test Release (assumed to be feature-complete) to RBCS.
10/13	Delivery of the shell DLL needed to [create test data] to RBCS.
10/14-10/20	Testing Phase I by RBCS.
10/21	Delivery of Phase I Test Results and Bug Reports by RBCS.
10/27	Delivery of the Second (Final) Test Release to RBCS.

10/28- 11/3	Testing Phase II by RBCS.
11/4	Delivery of Phase I Test Results and Bug Reports by RBCS.

Notice that the test basis—the use cases and test charters—are delivered at the start of the project. We analyze those and develop a plan. We then have two one-week test cycles, with a week in between cycles to allow for bug fixes. The entire project is just a little over one month long.

Finally, we see the risks and contingencies for the project in [Table 3-14](#). The risks are quite simple for this simple project. Notice, too, that we don't have a lot of ability to control the risks. Most of the contingencies result in reduced testing, delay of testing, or both.

3.3.11. Test Management Documentation and Test Plan Documentation Templates Exercise

Review the HELLOCARMS System Requirements Document. Using the IEEE 829 template, outline a combined master test plan and detail test plan covering system test and system integration test, based on your earlier risk analysis exercises and your analysis of the HELLOCARMS project. Use a bullet-item level of detail, 2 to 4 bullet items per section.

Table 3-14. Case study risks and contingencies

Risk	Contingency
Test release very buggy during test Phase I or II.	Run whatever tests are not blocked. Allow mid-phase test release during test phase, complete as many tests as possible.
Component remains very buggy at end of Phase II.	There is no provision within the scope of the project for additional testing, so this is an unmitigated risk.
Schedule slip in start of Phase I or Phase II.	Delay start of test phase. Because tester availability is now closely tied to specific schedule days, a delay in a phase start date may result in a disproportionately long delay in the phase end date.

If you are working in a classroom, break into groups of three to five. If you are working alone, you'll need to do this exercise yourself. If you are working in a classroom, once each group has finished, discuss the results.

I suggest allotting 90 minutes for this exercise, including the discussion.

3.3.12. Test Management Documentation and Test Plan Documentation Templates Exercise Debrief

Here's the outline I created for this exercise.

Test plan identifier

Adhere to whatever document naming/numbering standard exists at Globobank

Introduction

Describe project at a high level

Describe testing at a high level

Test items

Globobank system includes the following components:

- SQL and other items for database server
- Java and other items for application server
- Java, HTML, Flash, and other items for Web server

Globobank system includes the following integration components:

- LoDoPS interfacing
- Decisioning mainframe interfacing
- Other interfacing components

All of these items must be packaged in the same way as they will be for delivery to the data center

Features to be tested

Home equity loan, home equity line of credit, and reverse mortgage functionality

Rejection of undesirable applicants

End-to-end transaction handling

Error handling and recovery

Security

Regulatory compliance

HELLOCARMS/Scoring Mainframe Interfaces

HELLOCARMS/LoDoPS Interfaces

HELLOCARMS/GLADS Interfaces

HELLOCARMS/GloboRainBQW Interfaces

Features not to be tested

[At this point, only functional quality characteristics are identified.]

Approach

Primary test strategy is analytical risk-based testing, accounting for 60 percent of the test execution effort, with the effort allocated relative to risk item level

Secondary test strategy is reactive, using software attacks and exploratory testing, accounting for 20 percent of the test execution effort

Confirmation testing to validate bug fixes and check for success with testing previously failed test cases, accounting for 20 percent of the test execution effort

System testing and system integration testing will be handled by the HELLOCARMS independent test team, assuming some basic unit testing and component integration testing by development

Item pass/fail criteria

I would include this in the test criteria section in terms of measured product

quality

Test criteria (e.g., entry, exit, suspension, and resumption)

Need to define entry criteria for system and system integration test levels based on the following:

- Readiness of testware, test environments, test team
- Demonstrated quality of test items (unit and component integration test results, smoke tests, build verification tests, etc.)
- SDLC lifecycle model rules

Need to define exit criteria for system and system integration test levels based on the following:

- Product quality using bug metrics
- Completeness of testing using test case completion metrics
- Test efficiency using test hours metrics
- Thoroughness of testing using test coverage metrics
- Residual level of risk
- Resolution of known defects
- Management acceptance of residual risk

Test deliverables

Define the four or five test results dashboard charts and supporting data delivered to project manager during test execution

Define the detailed information delivered to technical support, development, etc., during test execution

Define the testware, test environments, test processes, etc., that will be delivered for maintenance testing at the end of test execution

Test tasks

Refer to the test estimate work breakdown structure

Environmental needs

Replica (as close as possible) of the data centers and call centers shown in figure 1 of the HELLOCARMS System Requirements Document

Test interfaces to the Scoring Mainframe, LoDoPS, GLADS, and GloboRainBQW, with sufficient control over the data interchange to produce all necessary inputs and outputs

Responsibilities

Define test team members and their roles

Define major nontest participants

Staffing and training needs

Define the team, perhaps by reference to the previous section

Discuss technical, testing, or domain training needed for team

Schedule

Refer to the test estimate work breakdown structure

Risks and contingencies

Refer to the quality risk analysis document for quality risks

Refer to the project risks identified in the quality risk exercise

Approvals

Add a signature section here if needed

3.4. Test Estimation

Learning objectives

(K3) Estimate the testing effort for a small sample system using a metrics-based and an experience-based approach considering the factors that influence cost, effort, and duration.

(K2) Understand and give examples of the factors listed in the syllabus which can lead to inaccuracies in estimates.

Estimation is a management activity, often a part of or a predecessor to the planning activity. However, estimation can also be part of a sales or bidding activity. Estimation creates approximate cost and schedule targets for the tasks that the estimators expect to occur on a particular project or as part of some particular ongoing operation.

A good estimate has a number of characteristics when it is created:

- It is based on the knowledge and wisdom of experienced practitioners.
- It is supported by those who will do the work.
- It is specific and detailed about costs, resources, tasks, and people.
- It is based on the most likely cost, effort, and duration for each task.

Given that an estimate has these characteristics in advance, it has the best chance of having the most important characteristics after the project is completed: that of having accurately forecast the costs, resources, and time required.

Unfortunately, software and system engineering estimation is difficult, and our estimates rarely achieve that important retrospective characteristic. The problems are not only technical, but also (and perhaps most of all) political. Gaining acceptance of realistic, accurate, complete estimates is hard. That said, there are well-established project management best practices for estimation. Applying these best practices can help you create good estimates.

In test estimation, we apply known estimation best practices to the testing work that will occur on a project or as part of an ongoing operation. Your test estimate should include all activities involved in the test process. For

example, you can use the ISTQB fundamental test process to create your work breakdown structure (often shown as a Gantt chart). That is, you can have the following major activities:

- Planning
- Control
- Analysis
- Design
- Implementation
- Execution
- Results reporting and exit evaluation
- Closure

Of these, test execution gets the most management attention. The cost, effort, and duration of test execution are critical because test execution is on the critical path. Unfortunately, test execution estimates are notoriously troublesome and unreliable when the system under test is of low or unknown quality—which is very often the case. The major determinant in how long test execution takes is very often not the time required to run each test case once but rather the time required to find, fix, and confirm as fixed all the critical bugs.

In some cases, people try to estimate test execution by estimating the number, effort, and duration of the test cases. However, unless you have some way of adjusting that for the find/fix/confirm cycles, that way of estimating can result in a test estimate based on the assumption of a low test case failure rate. That's often not a good assumption.

That said, you will make some assumptions during your estimation. Make sure that you document all the important estimation assumptions, and during test control, watch carefully for signs that those assumptions are invalid. If you can capture the assumptions as variables in a spreadsheet or a project management tool like Microsoft Project, you can then modify them as you go through the project, reestimating the schedule.

3.4.1. Factors Affecting Estimation

As you start to think about the assumptions that you make when estimating, you'll often start to become more aware of the factors that can influence cost, effort, and duration of the testing activities. It's important to consider all of these factors.

I typically think of these factors as falling into four categories:

- Process factors
- Material factors
- People factors
- Delaying factors

Let's look at each category. I will review the process, material, and people factors from the point of view of attributes that enable quicker, cheaper testing and more accurate, reliable estimates. The delaying factors tend to be associated with slower, more expensive testing and inaccurate, unreliable estimates.

Some factors arise from the process by which work is done:

- The extent to which testing activities pervade the project, starting on day one of the project
- Clearly defined hand-offs between testing and the rest of the organization
- Well-managed change control processes for project and test plans, product requirements, design, implementation, and testing
- The chosen system development or maintenance lifecycle, including the maturity of testing and project processes within that lifecycle
- Timely and reliable bug fixes
- Realistic and actionable project and testing schedules and budgets
- Timely arrival of high-quality test deliverables
- Proper execution of early test phases (unit, component, and integration)

Some factors, the material factors, arise from the nature of the project, the tools at hand, the resources available, and so forth:

- Existing, assimilated, high-quality test and process automation and tools
- The quality of the test system, by which I mean the test environment, test process, test cases, test tools, and so forth
- An adequate, dedicated, and secure test environment
- Separate, adequate development debugging environment
- The availability of a reliable test oracle (so we can know a bug when we see one)
- Available, high-quality (clear, concise, accurate, etc.) project documentation like requirements, designs, plans, and so forth
- Reusable test systems and documentation from previous, similar projects
- The similarity of the project and the testing to be performed to previous endeavors

Some factors arise from the people on the team:

- Inspired and inspiring managers and technical leaders
- An enlightened management team whose members are committed to appropriate levels of quality and sufficient testing
- Realistic expectations across all participants, including the individual contributors, the managers, and the project stakeholders
- Proper skills, experience, and attitudes in the project team, especially in the managers and key players
- Stability of the project team, especially the absence of turnover
- Established, positive project team relationships, again including the individual contributors, the managers, and the project stakeholders
- Competent, responsive test environment support

- Project-wide appreciation of testing, release engineering, system administration, and other nonglamorous but essential roles (i.e., not an "individual heroics" culture)
- Use of skilled contractors and consultants to fill gaps
- Honesty, commitment, transparency, and open, shared agendas among the individual contributors, the managers, and the project stakeholders

Finally, there are certain delaying factors that, when present, always increase schedule and effort:

- High complexity of the process, project, technology, organization, or test environment
- Lots of stakeholders in the testing, the quality of the system, or the project itself
- Many subteams, especially when those teams are geographically separated
- The need to ramp up, train, and orient a growing test or project team
- The need to assimilate or develop new tools, techniques, or technologies at the testing or project levels
- The presence of custom hardware
- Any requirement for new test systems, especially automated testware, as part of the testing effort
- Any requirement to develop highly detailed, unambiguous test cases, especially to an unfamiliar standard of documentation
- Tricky timing of component arrival, especially for integration testing and test development
- Fragile test data, such as for example, data that is time-sensitive
- The requirements for a high level of quality of the system, as would be the case for mission-critical and safety-critical systems
- A large, complex system under test

- Extensive or total newness, which would mean that no historical data from previous test projects or applicable benchmark data could be used for estimation or for testing

Some or perhaps all of these process, material, people, and delaying factors will occur on your next project. When present, these factors will influence the resources and time required. So, when preparing a test estimate, consider how to incorporate the impact of each of these factors.

Forgetting just one of these factors can turn a realistic estimate into an unrealistic one. Experience is often the ultimate teacher of why these factors matter, but if you are a smart test manager, make sure to ask smart questions about whether and how each factor will affect your project.

Your logging of deviations from plan will help you learn from experience, because in many cases, deviation from plan indicates an estimation assumption that proved invalid. In [figure 3-24](#), we revisit the deviation-from-plan test log I showed in an earlier section. I'm focusing on one specific deviation from the plan, the one related to the TIBCO licenses.

Figure 3-24. Example of estimation factors

Impact	Project Name	Noted	Change Description	Impact Description
8/10/1998	Integration	9/10/98	TIBCO licenses expire frequently; it was hourly, then daily, now it's weekly.	When this happens, tests die and must be restarted. It's probably chewed up twenty or so test hours, and maybe ten or so person-hours. It also reduces confidence in test results

In our plans and estimates, we had assumed a stable test environment. This hypothetical test environment was one where all infrastructure was properly configured and no false positives or test-blocking events occurred. Of course, this assumption was inaccurate. This inaccurate assumption lead to an inaccurate estimate and a flawed plan.

As an interesting side note, about two months after this note was entered in the test log, we discovered that a license key had not been properly installed at the beginning of testing. You might say, "Well, Rex, if you had a better pre-test checklist, you might have caught this." That's true for smaller, simpler, more stable test environments. On this project, we actually had a database that tracked complex dependencies between test cases, hardware, infrastructure, and software in the test environment and testers. Even with such a database, you can only be so fine-grained in what

you track. As a manager, you can—and should—try to mitigate test-related project risks. However, to assume that you will be able to mitigate all test-related project risks is a dangerous, optimistic fantasy that will result in inaccurate, unrealistic estimates.

3.4.2. Estimating Techniques

You can estimate both bottom-up and top-down. Bottom-up estimates are based on each task; i.e., we estimate each task individually and then derive the project estimate from those estimates. Top-down estimates are based on the overall project; i.e., we estimate for the test effort or perhaps for the high-level test activities all at once.

There are various test estimation techniques in use:

- Intuition, guesses, and experience
- Work breakdown structures, such as Gantt charts and tools like Microsoft Project
- Team estimation sessions, either unstructured or structured with some approach like the Delphic oracle, three point, or wideband
- Test Point Analysis, which is usable when function points are available
- Company standards and norms, which can influence and in some cases determine test estimates
- The typical or believed-typical percentages of the overall project effort or staffing level, such as the infamous tester-developer ratio
- Organizational history and metrics, including proven tester-developer ratios within the company
- Industry averages, metrics, and other predictive models

Let's look more closely at some of the more common or interesting techniques.

Perhaps the most common—and I think the most useful—estimation technique is creating a work breakdown structure. A work breakdown structure is a hierarchical decomposition of the testing project into activities, then tasks, then subtasks, until you have reached a fine-grained level of detail sufficient to allow you to make smart, accurate estimates for

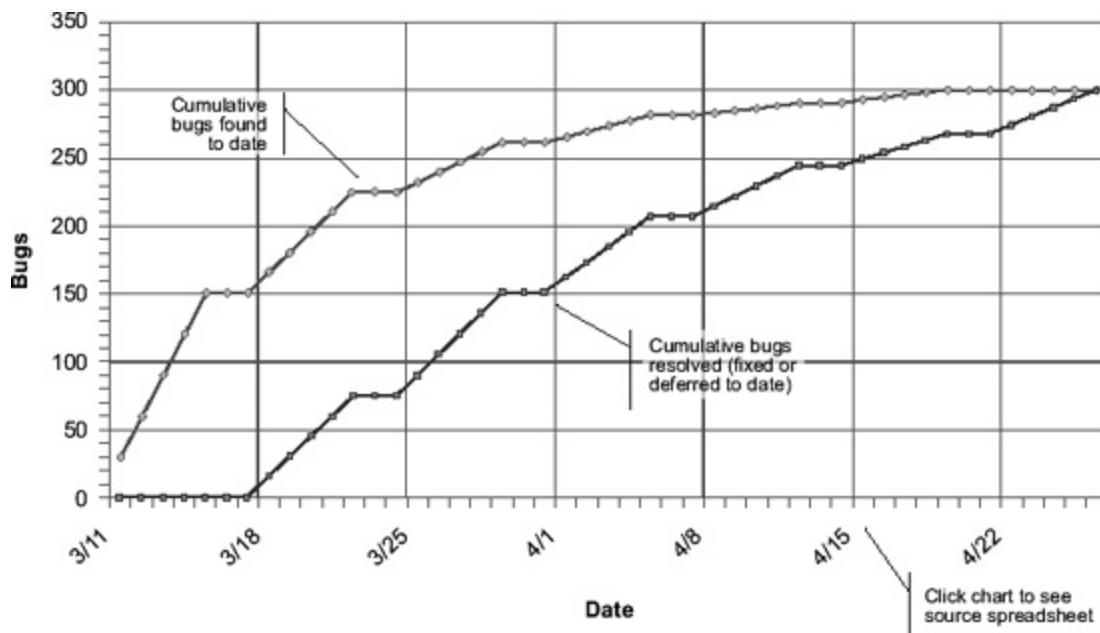
the duration and effort associated with each task.

To start with, you can use the ISTQB fundamental test process for the activities:

- Planning and control. I'd recommend breaking these into two separate activities, because control is ongoing.
- Analysis and design. The ISTQB fundamental test process treats these as single, commingled activities, and you probably can too, for estimation purposes.
- Implementation. You could also include this with analysis and design, as I often do, under the label "test development" or "test preparation". However, I often find it useful to separate test environment configuration tasks—which are under implementation in the ISTQB fundamental test process—from other test development tasks because I'm often dependent on outside groups for my test environments. In addition, if there is extensive test automation—which the ISTQB fundamental test process includes in implementation—I would tend to show those as a separate group of tasks too.
- Execution. This is the tricky one. Here, I often do not use a fine-grained work breakdown structure but will actually use spreadsheets to estimate both the time required to execute the test cases and the time required to find, fix, and confirm as fixed the critical bugs.^[1] I'll show you an example in a moment.

^[1] My book Critical Testing Processes illustrates an example of this approach.

Figure 3-25. Predicting bug find/fix/confirm duration



- Results reporting and evaluating exit criteria. You might choose to show this as a separate activity, or you might include it in control or even as part of test execution, as I often do.
- Closure. Because this happens, for the most part, after the project is over, people tend to be less interested in these tasks, but you should be interested because doing test closure properly saves lots of time on subsequent projects, builds a repository of useful historical metrics, and makes planning for the next project much easier.

Now, break down each activity to discrete tasks and perhaps subtasks. How do you know what tasks to include? Well, based on your risk analysis, you know what you're supposed to test and how much, so ask yourself, "What test suites are required for the critical risks? For each suite, what tasks are required?" For each task, ask yourself if you need to identify subtasks. At the finest level of granularity, tasks should be short, say one to two days to at most about a week.

In [figure 3-25](#), you see an example of the use of historical metrics to estimate the time required to find, fix, and confirm as fixed the critical bugs, which is a critical part of the test execution estimate. In the source spreadsheet that creates this chart, I used some historical data from a couple of past projects along with some simple models for bug find and fix rates. The model is parameterized so that it responds to the number of testers and developers assigned. At their core, models like this one rely on historical data, formulas and constants derived from previous projects. So the more data you have, the more accurate your model can be and the

more you can predict the accuracy of the estimate.

3.4.3. Using Industry Averages

Industry averages are also useful in estimation. In [Table 3-15](#) and [Table 3-16](#), you see figures derived from Capers Jones's studies.^[5] Jones has gathered data from thousands of projects across hundreds of clients, which makes his figures quite useful.

^[5] This material is found in Capers Jones's book *Estimating Software Costs*.

Jones uses function points for some of his figures. Function points are based on counting the size of an application based on its requirements, in a rigorous and defined fashion. You can also convert from lines of code to function points—and vice versa—according to Jones. For reference, one function point is about 125 lines of C code and about 55 lines of C++ code.

Table 3-15. Task, activity, and size estimation rules

Person-months for the project	(FP/150).FP0.4
Effort to develop a test case	1 person hr
Number of test cases needed	F _P 1.2
Number of bugs to expect	F _P 1.25
U.S. average bugs per function point	5.00
Bug origin ratio (requirements: design: code: documentation: regression)	3:4:5:2:1
Defect removal effectiveness (DRE) for a formal design inspection	65%
DRE for a formal code inspection	60%
DRE for a well-designed test suite	30%
U.S. average DRE across project	85%

Let's look at some particularly interesting metrics in [Table 3-15](#). For example, Jones has found that, on average, there are five bugs per function point. That equates to one bug for every 25 lines of C code and one bug for every 11 lines of C++ code.

Where do bugs come from? Jones has found that, typically, 20 percent of

bugs were introduced in requirements, 25 percent were introduced in design, 33 percent were introduced in coding, 15 percent were introduced in documentation, and 7 percent of bugs were introduced in as bad fixes (regressions).

Look at the DRE figures. DRE, or defect removal effectiveness, is the proportion of bugs that were present in the system (at the time that some quality control activity took place) and were removed by that activity. For example, Jones has found that formal design inspections remove on average 65 percent of the defects present in the design at the time of the review.

Table 3-16. Project effort ratios (testing includes bug fixing)

Coding: testing: other (10 FP)	2:1:1
Coding: testing: other (100 FP)	4:3:3
Coding: testing: other (1 KFP)	3:3:4
Coding: testing: other (10 KFP)	2:3:5
Coding: testing: other (100 KFP)	1:2:3
Coding: testing: other (end-user)	6:3:1
Coding: testing: other (MIS)	1:1:2
Coding: testing: other (systems)	2:3:5
Coding: testing: other (commercial)	2:3:3
Coding: testing: other (military)	1:1:3

Table 3-16 includes effort ratios relating coding, testing, and other activities. One thing to keep in mind about the testing figures is that they include the development time required to fix the bugs.

3.4.4. Test Point Analysis

Test point analysis is a test estimation approach that builds on function point analysis to estimate the testing effort for high-level testing, including system test and acceptance test. It also draws on ISO 9126 quality characteristics analysis to determine what is to be tested. It is part of the T-MAP test model, a model derived by the Sogeti consultancy in Europe.^[1]

[1] Test point analysis is discussed in Software Testing by Martin Pol, et

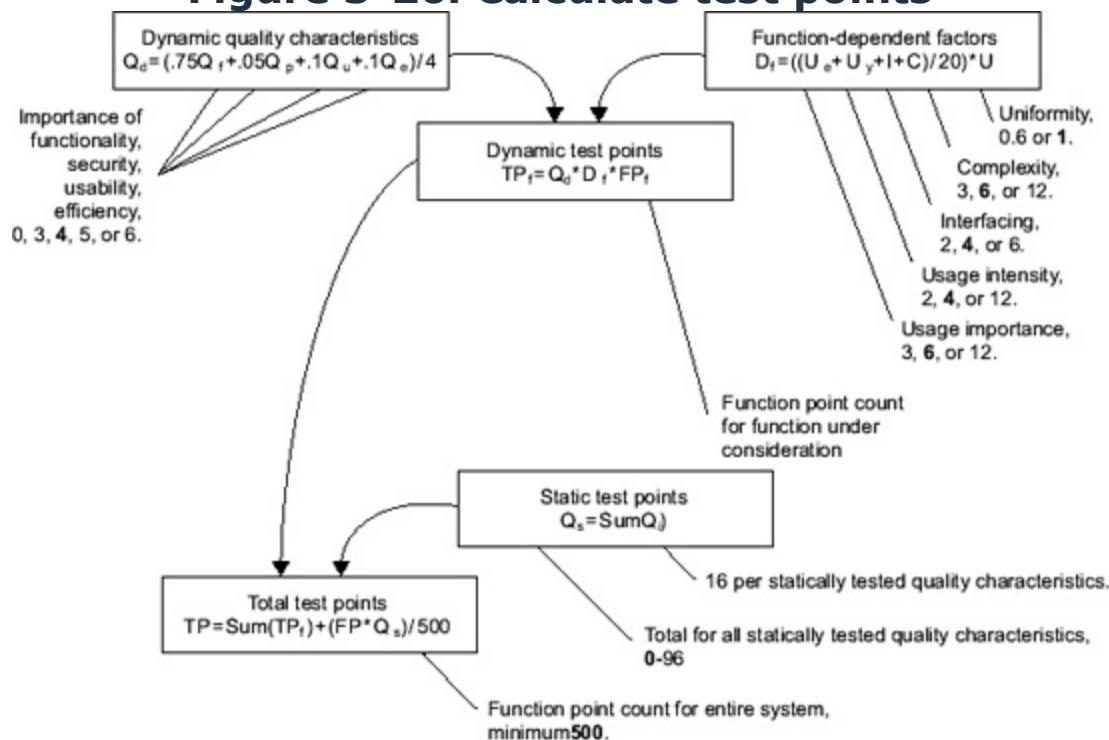
al.

The number of test points for a project is calculated based on three main factors:

- The size of the project, which is calculated using function points adjusted for complexity, interfaces, and uniformity.
- The test strategy, in particular which quality characteristics or risks are to be tested and to what extent. The underlying test strategy is assumed to be an analytical type.
- The productivity level of the participants, which is determined by the skills of the test team and influenced by the project, process, technology, and organization.

The test point analysis model is an interesting one. Even if you can't apply it directly, it might help you think about ways you could derive your own test estimation models using historical data. Let's look at the steps of the TPA process. In [figure 3-26](#) you see the calculation of test points. Let's go around this figure clockwise starting in the upper left.

Figure 3-26. Calculate test points



First, we are going to calculate, for each functional area of the system, a figure that gives the effort weighting for the dynamic quality

characteristics; i.e., those quality characteristics that we will test with dynamic tests. Dynamic tests, remember, are tests in which we execute the system under test, as opposed to static tests, in which the system under test is not executed. Four characteristics are considered: functionality, usability, security, and efficiency. You select a number—0, 3, 4, 5, or 6—depending on the importance of dynamic testing for each quality characteristic. Smaller numbers indicate less importance, large numbers more importance. The default value, shown in bold on the figure, is 4.

Moving to the upper right, next, we calculate, for each functional area of the system, a function-dependent weighting factor based on various attributes that will make it harder or easier to test. For the factors of usage importance, usage intensity, interfacing, and complexity, you select one of three numbers—again with the default shown in bold. Lower numbers mean it's easier or less critical to test, higher numbers mean it's harder or more important to test. For uniformity, you select one of two numbers, with the smaller number meaning harder to test.

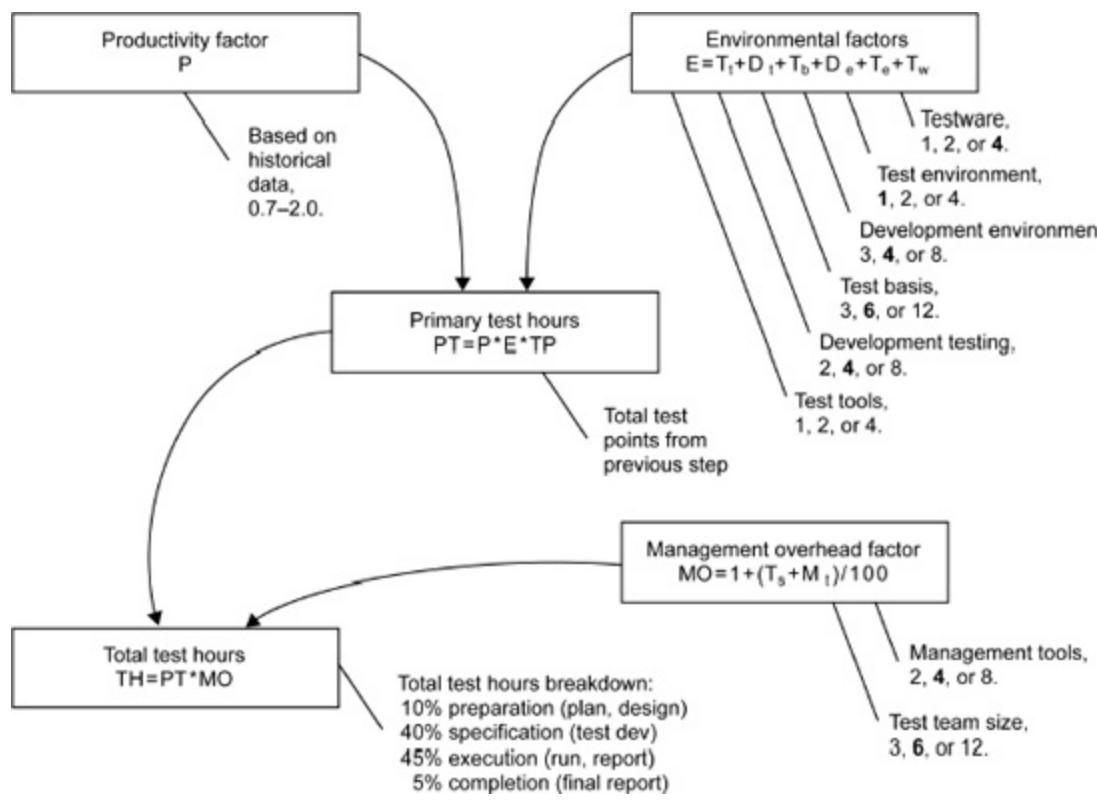
Moving to the upper middle, we next calculate the dynamic test points for this functional area of the system. We multiply the dynamic quality weighting factor by the function-dependent weighting factor by the function point count for this functional area. We repeat this process to calculate the dynamic test point count for all functional areas of the system. This could be a lengthy process if we are dealing with a complex system.

Now, moving to the lower middle, we can calculate the static test points, being the test points we intend to cover by static testing for the entire system. You award 16 static test points for each of the six ISO 9126 quality characteristics that will be addressed by a static test; e.g., 16 points if you intend to review the functional requirements of the system, 16 points if you intend to use static analysis to detect code maintainability problems, etc. Your static test points can range from 0 to 96.

Finally, at the lower left, we calculate the total test points. We sum all the dynamic test points across all functional areas. We then multiply the function point count for the entire system by the static test points and divide that product by 500.

In [figure 3-27](#) we see the use of the test points to calculate the test hours. Again, let's go around this figure clockwise from upper left.

Figure 3-27. Calculate test hours



First, we have a productivity factor, ranging from .7 to 2.0. Smaller numbers mean more productive, larger numbers less so. This factor is calculated from historical data. What that means, effectively, is that to use test point analysis for the first time at an organization, you need to calculate the test points and compare them to the test hours for two or three previous projects to calibrate this productivity factor.

Next, to the upper right, we calculate an environmental weighting factor that will increase or decrease the number of test hours. We rate the situation with test tools, development testing, test basis test environment, development environment, and testware as the underlying attributes. Smaller numbers mean better, more conducive to efficient testing, while larger numbers mean worse, more likely to cause inefficient testing. Again, the default values are shown in bold.

In the middle of the figure, we calculate the primary test hours by multiplying the test points by the productivity factor by the environmental factor.

At the lower right, we calculate management overhead. We consider the test team size and the management tools. Bigger numbers mean more overhead, and thus more total test hours, while smaller numbers mean less overhead.

Finally, at the lower left, we calculate the total test hours by multiplying primary test hours by management overhead. The test hours are then distributed across the test process—which is T-MAP's process, not the ISTQB's—using the breakdown shown at the bottom middle of the figure.

Now, this is clearly a complex model. In addition, you could argue that important factors like outsourcing and offshoring are left out. While imperfect, it does give some ideas about how a complex estimation model can be built, and ultimately tuned for your organization. Given enough historical data from past projects, you can probably create and tune a very accurate test estimation model.

3.4.5. Negotiation and Reducing Test Scope

Negotiation is typically part of test estimation, even if you're not selling your services or bidding on a job. You have to convince someone—the project manager, project sponsors, other project stakeholders, whoever approves schedules and budgets—that the money you are asking for, the time you are asking for, the people you are asking for will be a good investment.

In a later section of this chapter, we'll address the business case for testing, which can help to sell the estimate. However, it's often the case that you must negotiate. Negotiation can result in adjustments to the estimate, generally downward. In these negotiations, you should be trying to work with the project stakeholders to achieve the best possible balance of schedule, budget, quality, and features.

In your attempts to reach the best possible estimate, you can use risk to reduce test scope. One way is to eliminate test coverage areas. You can identify lowest-risk areas and drop or reduce test coverage in those areas. This is a "preserve depth, reduce breadth" approach.

Another way is to reduce the extent of testing while maintaining some amount of coverage. You can identify the least risky of the highest-risk areas and then adopt a balanced extent of testing. This is a "preserve breadth, reduce depth" approach.

There are also other options. You can postpone automation or tool development that you planned to undertake on the project. Of course, the risk of this approach is that you might find that, if you always chose to negotiate away test process improvements like test automation, how will you ever improve testing? Another alternative would be to identify efficient

areas to outsource, such as compatibility or performance testing, where doing so can reduce costs, speed up the schedule, or both.

The negotiation of a budget and schedule can be frustrating. Try to stay focused on the goal of doing the best possible testing within the constraints you'll ultimately have. Be creative to come up with ways that work. Also, be sure that you understand what you are being asked to negotiate. In some cases, there is more money available if the schedule can be accelerated; in other cases, money is the scarcest resource but some schedule delay is acceptable. If both are nonnegotiable, ask management if they actually mean that quality is less important than both schedule and budget? If so, then perhaps they can reduce the scope of the project overall?

Now, if you do end up negotiating away some of the testing, make sure that stakeholders approve reductions in scope. The people who participated in the quality risk analysis should be kept informed of the differences between what was agreed in the analysis and what ultimately will be tested.

This can involve patience. You need to be ready to iterate until this is done, all the while burning as little of your political capital as you can. Negotiate in a way that results in everyone coming away, if not happy, at least comfortable that a realistic, accurate estimate is in place that will result in testing making some significant contributions to the project.

3.4.6. Test Estimation Exercise

Revisit your solution to the exercise in the second "Testing in the Software Lifecycle" in [chapter 1](#). Develop an estimate for the testing work, using the activities you outlined in that solution. You should also consider your quality risk analysis and test plan.

Decompose down to major tasks (3-10 days).

If you are working in a classroom, break into groups of three to five. If you are working alone, you'll need to do this exercise yourself. If you are working in a classroom, once each group has finished, discuss the results.

I suggest allotting 60 minutes for this exercise, including the discussion.

3.4.7. Test Estimation Exercise Debrief

To create this solution, I proceeded in two steps. First, I went through the solution to the [chapter 1](#), section 1 exercise, adding estimated effort in

person-days to each task.

1. Initial Project Planning Period

1.1. Test Planning and Control

1.1.1. Create system and system integration test plan (5d)

1.1.2. Participate in overall project planning (5d)

1.2. Test Analysis and Design

1.2.1. Perform quality risk analysis (5 d)

1.2.2. Outline test suites needed in each iteration (1 d)

2. Iteration One (Very High Priority Features)

2.1. Test Planning and Control

2.1.1. Adjust plan as needed for this iteration (1 d)

2.1.2. Guide testing work during this iteration (ongoing)

2.2. Test Analysis and Design

2.2.1. Adjust quality risk analysis (1 d)

2.2.2. Design test suites and cases for this iteration (30 d)

2.3. Test Implementation and Execution

2.3.1. Implement test suites for this iteration (30 d)

2.3.2. Execute test suites for this iteration (duration 6 weeks)

2.4. Evaluating Exit Criteria and Reporting

2.4.1. Check test results against exit criteria in test plan (ongoing)

2.4.2. Report test results to project management team (ongoing)

3. Iteration Two (High Priority Features)

3.1. Test Planning and Control

3.1.1. Adjust plan as needed for this iteration (1 d)

3.1.2. Guide testing work during this iteration (ongoing)

3.2. Test Analysis and Design

3.2.1. Adjust quality risk analysis (1 d)

3.2.2. Design test suites and cases for this iteration (30 d)

3.3. Test Implementation and Execution

3.3.1. Implement test suites for this iteration (30 d)

3.3.2. Execute test suites for this iteration (duration 6 weeks)

3.4. Evaluating Exit Criteria and Reporting

3.4.1. Check test results against exit criteria in test plan (ongoing)

3.4.2. Report test results to project management team (ongoing)

4. Iteration Three (Medium Priority Features)

4.1. Test Planning and Control

4.1.1. Adjust plan as needed for this iteration (1 d)

4.1.2. Guide testing work during this iteration (ongoing)

4.2. Test Analysis and Design

4.2.1. Adjust quality risk analysis (1 d)

4.2.2. Design test suites and cases for this iteration (30 d)

4.3. Test Implementation and Execution

4.3.1. Implement test suites for this iteration (30 d)

4.3.2. Execute test suites for this iteration (duration 6 weeks)

4.4. Evaluating Exit Criteria and Reporting

4.4.1. Check test results against exit criteria in test plan (ongoing)

4.4.2. Report test results to project management team (ongoing)

5. Iteration Four (Low Priority Features)

5.1. Test Planning and Control

5.1.1. Adjust plan as needed for this iteration (1 d)

5.1.2. Guide testing work during this iteration (ongoing)

5.2. Test Analysis and Design

5.2.1. Adjust quality risk analysis (1 d)

5.2.2. Design test suites and cases for this iteration (30 d)

5.3. Test Implementation and Execution

5.3.1. Implement test suites for this iteration (30 d)

5.3.2. Execute test suites for this iteration (duration 6 weeks)

5.4. Evaluating Exit Criteria and Reporting

5.4.1. Check test results against exit criteria in test plan (ongoing)

5.4.2. Report test results to project management team (ongoing)

6. Iteration Five (Very Low Priority Features)

6.1. Test Planning and Control

6.1.1. Adjust plan as needed for this iteration (1 d)

6.1.2. Guide testing work during this iteration (ongoing)

6.2. Test Analysis and Design

6.2.1. Adjust quality risk analysis (1 d)

6.2.2. Design test suites and cases for this iteration (30 d)

6.3. Test Implementation and Execution

6.3.1. Implement test suites for this iteration (30 d)

6.3.2. Execute test suites for this iteration (duration 6 weeks)

6.4. Evaluating Exit Criteria and Reporting

6.4.1. Check test results against exit criteria in test plan (ongoing)

6.4.2. Report test results to project management team (ongoing)

7. Post Project Period

7.1. Test Planning and Control

7.1.1. Document variances from plan (5 d)

7.1.2. Participate in project retrospective (1 d)

7.2. Test Closure Activities

7.2.1. Finalize testware for archiving or hand-off (20 d)

7.2.2. Document test environment configuration (10 d)

Some comments on these estimates:

- I am assuming a skilled test team that can work with relatively high-level, logical test cases.
- I am assuming approximately equal content size for each iteration, with two person-months of test design and implementation work for each iteration and a six-week test execution period, consisting of three two-week test passes.

Second, I put this schedule into Microsoft Project, setting up the appropriate dependencies between iterations and major activities. You would need to do this work-breakdown-structure using sticky notes on a white board if you

did not have access to such a tool during the course. By the way, I chose January 1 as the arbitrary start date for the project.

Some comments on this work breakdown structure:

- I am assuming one test manager, two test leads, and six test analysts in the test team.
- I don't have a lot of information on the amount of development effort or the number of bugs I will find, so this is a first estimate.
- I would need to align this with the project plan and development plan to ensure that they were consistent.
- If we assume that I will receive weekly test releases during test execution, then I should try to overlap the analysis and design tasks of each subsequent iteration more completely with the previous iteration so that test execution always starts on a Monday.

The Gantt charts produced by Microsoft Project are shown in [figure 3-28](#), [figure 3-29](#), and [figure 3-30](#).

Figure 3-28. Gantt for planning and iterations one and two

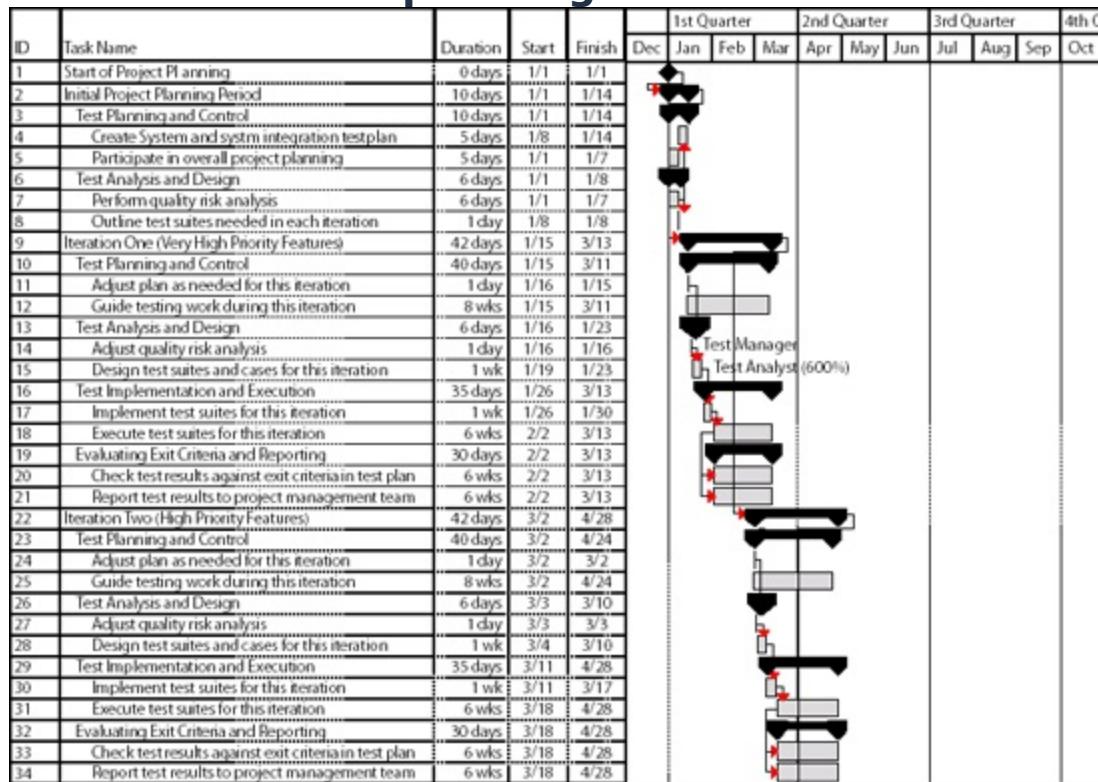


Figure 3-29. Gantt for iterations three and four

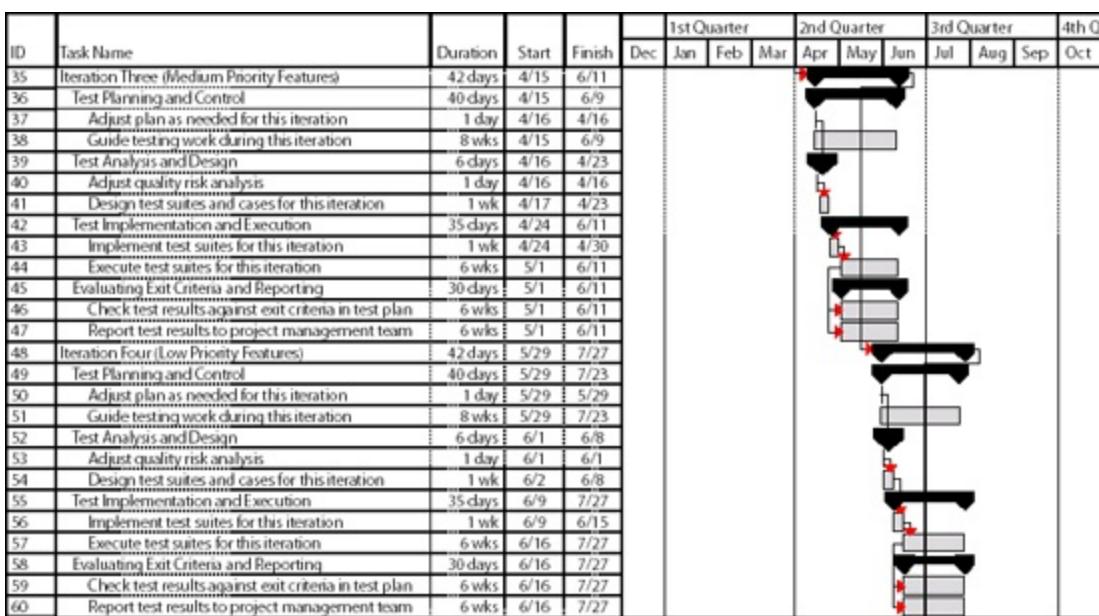
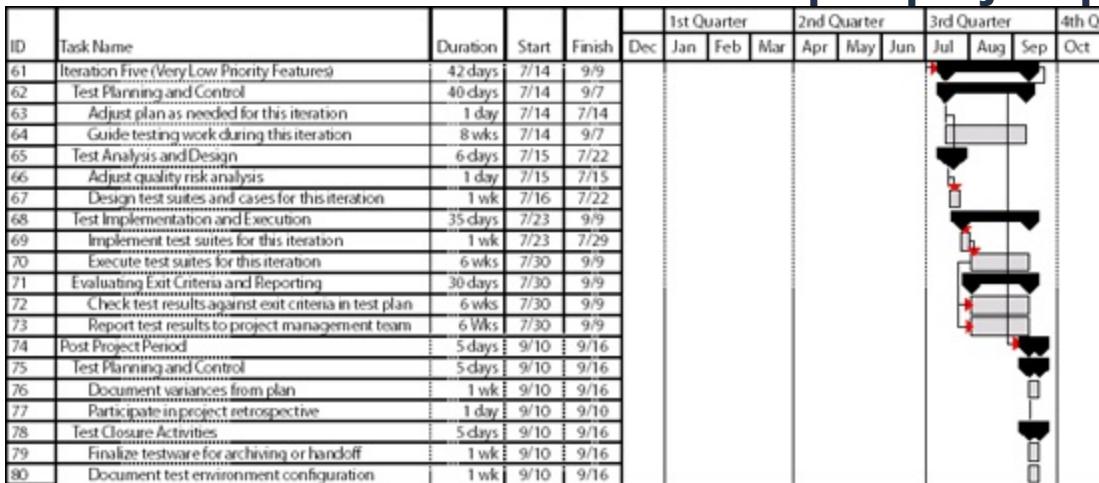


Figure 3-30. Gantt for iterations five and postproject period



3.5. Scheduling Test Planning

Learning objectives

(K2) Explain the benefits of early and iterative test planning

Support your explanation by examples

Dwight Eisenhower, United States general and eventually president, once said, "In preparing for battle I have always found that plans are useless, but planning is indispensable." Now, a project is not a battle—at least, usually no one dies while carrying out a software project—but it shares many attributes with battles. One is the need to try to discover the salient, critical attributes of the endeavor early enough to be able to take useful actions for them and to identify the key risks early enough to mitigate those risks, or at least adopt contingency plans for them should the risks become negative events.

Planning in advance will provide a number of benefits. One is the early discovery and management of risks, often in a way that allows reduction of likelihood and of potential impact. Another is the opportunity to carefully, and in timely fashion, coordinate with others who will participate in the project, ensuring that everyone knows their part when the time comes. Another is the delivery of a high-quality plan, one that is a useful road map for the work to come. These advantages are as true for advanced test planning as for any other project.

Advanced test planning provides some additional benefits too. During test planning, we can often detect project risks and problems beyond the scope of testing, such as bad assumptions that might cause operational problems in production. Early detection allows the project team to manage these risks and problems before they become full-fledged production crises.

During test planning, as with quality risk analysis, we can often detect quality risks and problems before test execution, such as missing information about performance and reliability in the product requirements. We can feed this information back into the quality risk documents and from there into the test design, implementation, and execution process. We can also feed this information to the project team to allow them to take steps to eliminate the problem, if applicable.

During test planning, we can often identify problems in project plans and

other project work products, such as missing skills in the project team in specialized areas like reliability, usability, localization, or performance. We can feed this information to the project team to allow them to take steps to eliminate the problem.

During test planning, we might identify opportunities to adjust the test staff, budget, effort, or duration to better balance quality priorities against feature, schedule, and budget constraints.

During test planning for early levels of testing, we can identify critical components, such as servers or databases or central software elements. This information can allow us to do risk-based integration, planning to test these critical components separately and then together earlier in the project to identify important integration problems as early as possible.

Generally, testing activities are downstream from the rest of the project, with the exception of the help desk or technical support, technical publications, production operations, and project sponsors, which are downstream of testing. Testing does not, should not, and cannot exist in a vacuum. Testing by itself has no value; it only has value when it supports the delivery of something of value to other project stakeholders. Testing by itself is incapable of action because it involves the evaluation of test items built and delivered by others, often in test environments configured and supported by others.

So, a critical element of test planning is the identification of test-to-project touchpoints and hand-offs. In the following list are some examples of such touchpoints and hand-offs:

- Arranging for delivery of early, informal builds during test implementation to test the test cases, test procedures, test environments, and test tools.
- Coordinating the delivery of formal test releases, referred to in IEEE parlance as test item transmittal, in a regular, organized, reliable, efficient, and minimally disruptive fashion.
- Ensuring that test releases have reliable, incrementally increasing, easily ascertained revision numbering so that meaningful test results can be gathered and reported.
- Providing for reliable, timely, competent test environment support when the complexity, size, or criticality of the test environment is such that

the test team should not support it.

All of these are tasks that involve a mutually agreed-upon process, and so you must plan them *cooperatively* with the other project participants.

Even when you actively engage with other project participants to try to obtain the information you need for your test plan, if you wait for all the information to come in, you'll typically end up finishing your test plan the day before test execution starts—if you're lucky. I've had projects where I had open, unresolved questions in the test plan even after the project was over. If you insist on getting the answers to all the questions in advance of writing your test plan, that means no advanced test planning.

Instead, what I try to do is to develop test plans early and issue drafts for discussion to the key testing participants and stakeholders. As additional information arrives, I will update the drafts. If there's information I urgently need—questions I simply must resolve quickly—I will pester and escalate until I get the answers. However, since that can irritate other project stakeholders, thus costing me political capital, I try to use that approach only when necessary.

I have found that this iterative, sometimes Socratic approach to test plan creation, release, and review works very effectively as a planning process. It transforms the test plan into a physical means to promote consensus, communication, and discussion about testing, in particular the specifics of testing for this project or test level, the hows and whos and wheres and whens and whats.^[1]

[1] The Socratic approach, as those who attended law school would know, involves asking thought-provoking questions as a way to help people learn. Carleton University has provided a good explanation, with examples, at serc.carleton.edu/introgeo/socratic/.

3.5.1. A Case Study in Early Test Planning

Let's look at an example of how early and iterative test planning on a project identified a number of important questions early on. On the Internet appliance project, which I've mentioned a few times in this book, I sent two drafts of the test plan, each with questions embedded. [figure 3-31](#) shows the email cover letter sent with the first draft. These drafts served to stimulate discussion and resolution of issues, both in emails and in meetings. In the next few pages, I'll show you some specific examples of the questions I raised in the draft test plan.

Figure 3-31. Early test planning

All—

Attached is the first draft of the Test Plan. It covers hardware (DVT & PVT) and software (Integration and System Test). There are more questions than answers here, at this point, but hopefully we can get most of the issues resolved at tomorrow's review meeting, at least sufficient to proceed.

Ken, as a next step, I'll be developing the budget and the detailed schedule this afternoon and tomorrow. Those will be based on this plan, so, if a first review suggests problems, let's talk ASAP.

Regards,
Rex

figure 3-32 shows a question I raised about the prioritization of hardware-related quality risks. I am asking Ed, Qadeer, and Greg to provide priority of the quality risks shown and perhaps to add some additional quality risks. This question was critical to determining the order in which hardware tests would be run. (Due to space limitation, there are a number of quality risks that are not shown in this figure.)

figure 3-33 shows three test-environment questions. The first has to do with the number of units required for the product verification test, which was the highest level of hardware testing, the last test level prior to declaring the hardware done. I was asking for clarification as to whether we would accept testing of hand-built engineering samples. I was fairly certain that the answer would be, "No, we must use production line units", but I wanted to be sure.

The second question has to do with the location of the test lab and the

people who would be involved in configuring and supporting it.

Figure 3-32. Quality risk priority question



Figure 3-33. Test environment questions

From the Product Verification Test (PW) Entry Criteria section: "The Vendor provides PVT units, in the quantities specified in this plan, for PVT test. (See Test Configurations and Environments.) [Greg/Qadeer/Ed: Do we want the PW units mass-produced on the production line or are engineering samples okay?]"

From the Test Configurations and Environments section: "[Greg/Ken: We need to define the test lab. I haven't spent any time discussing this with people. Who are the right players? Where will it be?]"

From the Test Configurations and Environments section: "[Greg/Qadeer: As first [estimate], per Greg, we will have 50 DVT [Development Verification Test] units and 100 PVT units. We'll need a pretty substantial number for the MTBF [mean time between failure] demonstration. Also, if we run an ALT [accelerated life test], those tests will break the units, so we have to count on some number of samples being "consumed" by the testing.]"

The third question has to do with the number of test units needed. Mean time between failure tests are notorious for tying up hardware resources. Accelerated life tests not only tie up hardware resources, but, because they push the resources until failure, they actually result in a depletion of test resources. I'm asking here for Greg and Qadeer to confirm my thinking on these numbers.

Figure 3-34. Test design questions

[On the server side, we may be able to use some COTS test tools. For example, SilkPerformer ... can test Web sites. Also, [the mail software vendor] should be able to provide us either with test tools or with direct test assistance to execute tests of the mail side. This would leave the update and state interfaces. We should be able to test this by talking to the servers over the LAN, via another x86-based PC running Linux. The test tool can initiate a download from the server by proclaiming its software or data level to be "below" the current revision. It receives the new release from the server but sends the data to /dev/null. The key challenge will be figuring out how to simulate load levels of 500,000 users. We may want to create some other server-side traffic generation tools that load specific subsystems, such as the database or the LAN infrastructure, that makes it "look like" 500,000 people are logged on, but we'd have to do some very careful performance modeling to figure out how to do that. Wayne/Jamie, can we pull in [the mail software vendor], understand their piece of it, and then work together on a high-level design for the load generator(s)? Greg/Ken, I'm going to include some money in my proposed budget for COTS solutions on the Web side, if that's okay?]

figure 3-34 shows some questions I had related to test design. First, I described my current thinking on the architecture and components required for server-side testing, at a high level. Next, I asked two questions. For the two senior development managers/architects, Jamie and Wayne, I asked for some time to discuss with them and the mail software vendor how we should finalize the server-side test architecture and components. For the two decision makers who would approve my test budget, Greg and Ken, I asked for approval to include pricing for COTS test tools in my budget.

3.6. Test Progress Monitoring and Control

Learning objectives

- (K2) Compare the different procedures for controlling test progress.
- (K2) Give at least five conceptually different examples of how test progress findings influence the course of the test process.
- (K4) Use findings related to test progress observed during monitoring and control activities and measures in order to outline an action plan to improve the current test process. Suggest improvements.
- (K4) Analyze test results and determine the test progress, documented in a monitoring report and a final test summary report covering all four reporting dimensions.

While the specifics of test progress monitoring and reporting tend to vary from one organization to another, I have observed five primary dimensions for test progress monitoring and reporting in use:

- Quality risks
- Defects, incidents, or bugs
- Test cases or procedures
- Coverage of the test basis, code, or other areas of interest
- Confidence

The first four—risks, defects, tests, and coverage—are measurable and quantifiable, sometimes in too many ways. What I mean by "too many ways" is that test managers sometimes fixate on one or two of these dimensions, most commonly defects and tests, and produce a dozen or so different views, analyses, and charts. This is less helpful than a holistic approach that considers all five dimensions.

When considering all five dimensions, a test manager should try to measure each dimension and have a way of reporting results. For confidence, we often either measure and report it subjectively or use surrogate metrics, perhaps most typically some combination of requirements coverage, defect find rates, and defect backlog size.

It's important that we align these dimensions of test progress monitoring

with our evaluation of completeness of testing. We should have test progress metrics that align with our defined exit criteria—and vice versa.

Let's look at the four directly measurable dimensions, using some example charts and tables as we go along. While I will use some actual case studies as examples, most of these examples are based on testing of a single, hypothetical e-commerce application.

3.6.1. Product Risk Metrics

If you are following a risk-based analytical test strategy, you can measure testing in terms of risk. You can look at this in a negative sense, in terms of the number of remaining risks, and include detail like the type and level of these risks. You can look at this in a positive sense, in terms of the number of risks mitigated, again potentially including the type and level of these risks. With a tool like Excel, you can cobble this together manually, based on coverage data for tests and bugs. However, some test management tools are starting to include this ability directly.

Figure 3-35. Product risk metrics

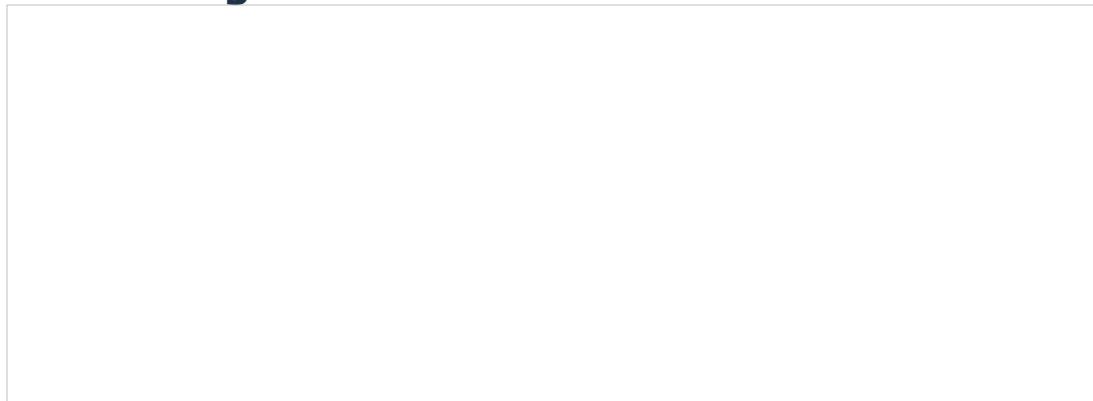


figure 3-35 shows an example you saw earlier, in the section on risk-based testing. Let's review how it works. The region in medium gray represents risks for which all tests were run and passed and no must-fix bugs were found. The region in light gray represents risks for which at least one test has failed and at least one must-fix bug is known. The region in dark gray represents other risks, which have no known must-fix bugs but still have tests pending.

As you can see from the timeline at the bottom, when we start testing, the pie chart is all dark gray. Over time, as test execution continues, we should see the light gray region eat into the dark gray region and the medium gray region eat into the light gray region. As we get toward the end of test

execution, most of the dark and light gray should be gone.

At any time, for the medium gray region, we can produce a detailed list of risks that are tested, are known to pass the tests, and are not known to have any must-fix bugs. For the light gray region, we can produce a detailed list of risks against which known failed tests and must-fix bugs exist. For each risk on this list, we can list the tests and the bugs. For the dark gray region, we can produce a detailed list of risks and the tests still remaining to be run against them. This gives a clear picture of the risk reduction achieved and the residual level of risk.

This type of metric is not commonly encountered, but it's exactly the kind of strategic, critical information that senior managers, executives, and those outside of testing want to receive from testing.

3.6.2. Defect Metrics

Now, when we talk about defect metrics, these are perhaps the most commonly used—and often abused—metrics. One common defect metric looks at the cumulative number of defects reported versus the cumulative number resolved. As the project approaches conclusion, the cumulative number of defects reported should become level (indicating a low or zero discovery rate) while the cumulative number resolved should converge with and ultimately intercept the cumulative number reported.

In the safety-critical and mission-critical world, people tend to be less interested in defects and more interested in failures. So here we often see metrics related to mean time between failure or other failure arrival rates.

It is also common to analyze or break down defects by categories, especially when trying to understand why defects happen. We can base these categories on test items, components, or major subsystems, which can help us spot bug clusters. We can base these categories on root causes or sources of bugs, which help us reduce bug introduction rates. We can base these categories on test releases, which we hope would reveal a downward rate over time but which might contain some surprising spikes. We can base these categories on the phase in which defects were introduced, detected, and removed, which can reveal opportunities to reduce the cost and schedule effects of defects. We can also base these categories on owner.

Another common defect metric is the closure period, which is a chart that shows trends in the lag time from defect reporting to resolution.

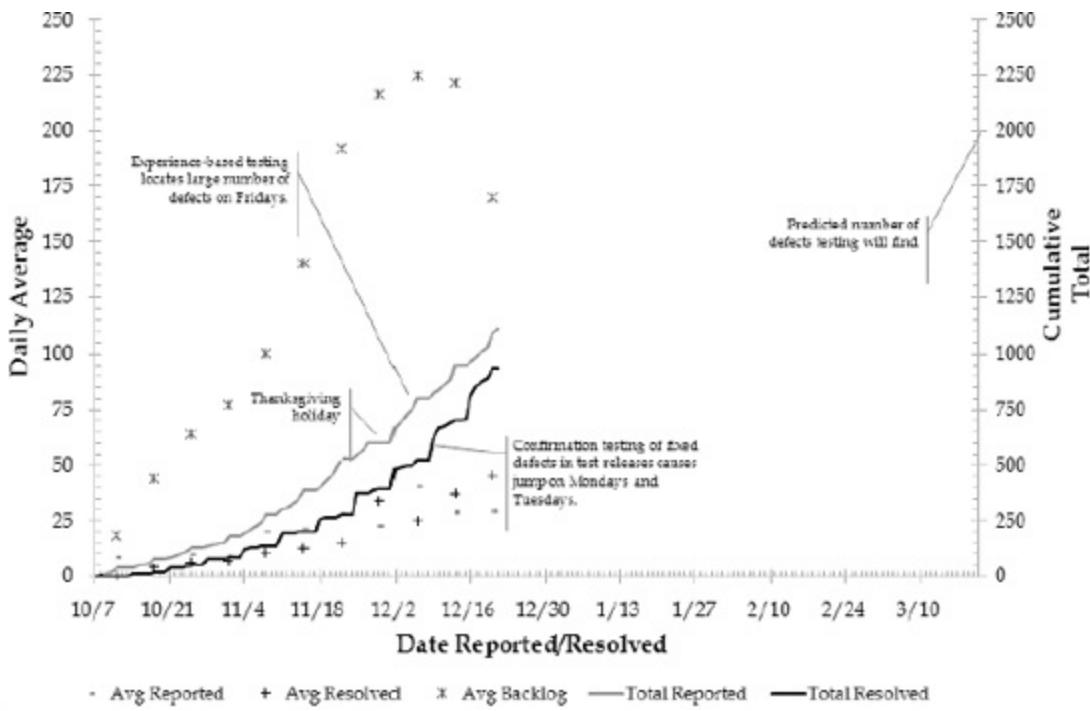
I find these various defect metrics, and ways of charting the trends and analyses, very useful in understanding project, product, and process behaviors and problems. However, sometimes people misuse these metrics as unidimensional metrics. What I mean is that, in some cases, people assume they need only one metric, often the cumulative defects reported and resolved. In other cases, people have a dozen metrics, but all of them all are based on defects.

Another common misuse is to use defect metrics to evaluate people. For example, rewarding testers based on bug find rates leads to lots of trivial bug reports and very cynical developers. Punishing developers based on bug closure period penalizes the best developers (who tend to be assigned the hardest bugs) and encourages sloppy, poorly designed, "bandage-style" bug fixes.

Let's look at a few examples of defect-based charts. [Figure 3-36](#) shows a data-rich variant of the cumulative reported/cumulative resolved chart. This chart has five main data sets, two of which are plotted on the right-side axis and three of which are plotted on the left-side axis.

The first data set, shown by the upper solid line, is the cumulative number of defects reported. It is graphed against the right-side axis. The second data set, shown by the lower solid line, is the cumulative number of defects resolved, also graphed against the right-side axis. The cumulative number of defects resolved includes both fixed and deferred defects, since the point of this chart is to show whether the defect status is trending toward readiness for delivery.

Figure 3-36. Cumulative defects reported/resolved



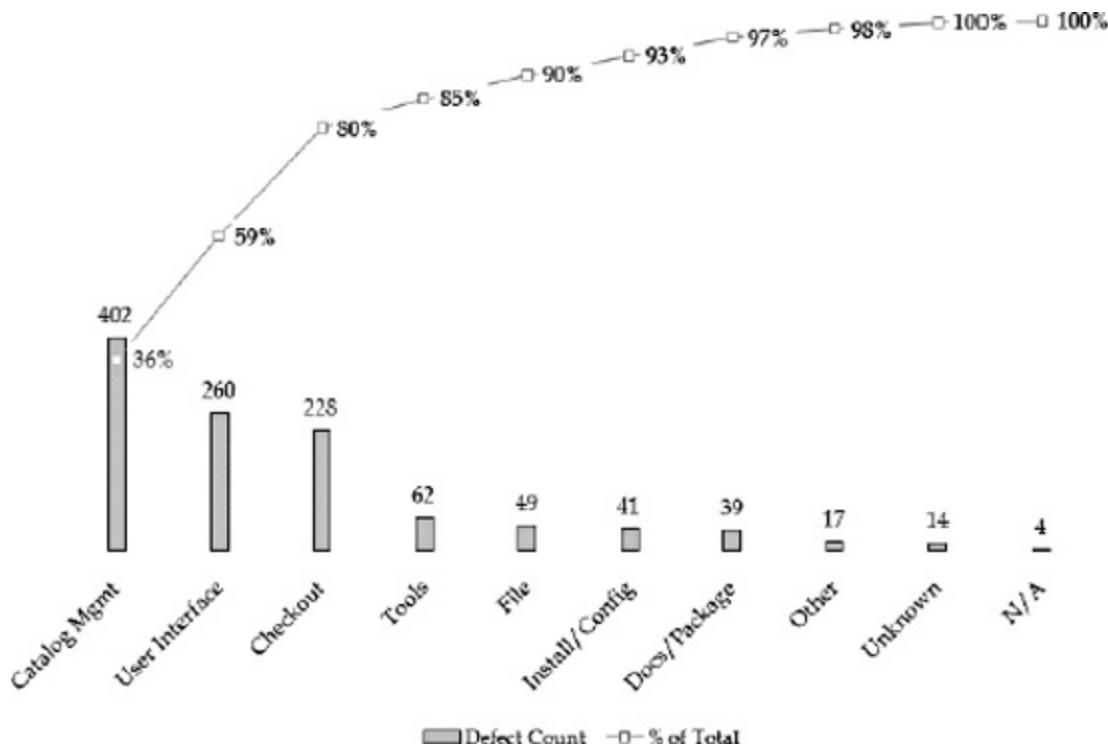
The third data set, shown by the gray minus marks, is the average number of defects reported in a given calendar week. It is graphed against the left-side axis. The fourth data set, shown by the gray plus marks, is the average number of defects resolved in a given calendar week, also graphed against the left-side axis.

The fifth data set, shown by the yellow asterisk marks, is the average backlog in a given calendar week. The backlog for any day is the total number of defects found minus the number resolved.

Because trend charts show trends in time, not causality, this chart includes notations to explain various shapes of curves, such as the flattening of the cumulate defect find line during the Thanksgiving holiday. Without such notations, viewers can mistake echoes of the process as the information we are trying to understand, which in this case is our current situation with respect to product quality problems.

[figure 3-37](#) shows a Pareto analysis of defects by the subsystem they were observed in. We can see that most of the defects—fully 80 percent of those reported—were observed in the catalog management, user interface, or checkout subsystems of this hypothetical e-commerce application. The remaining subsystems accounted for very few of the defects.

Figure 3-37. Defect breakdown by subsystem

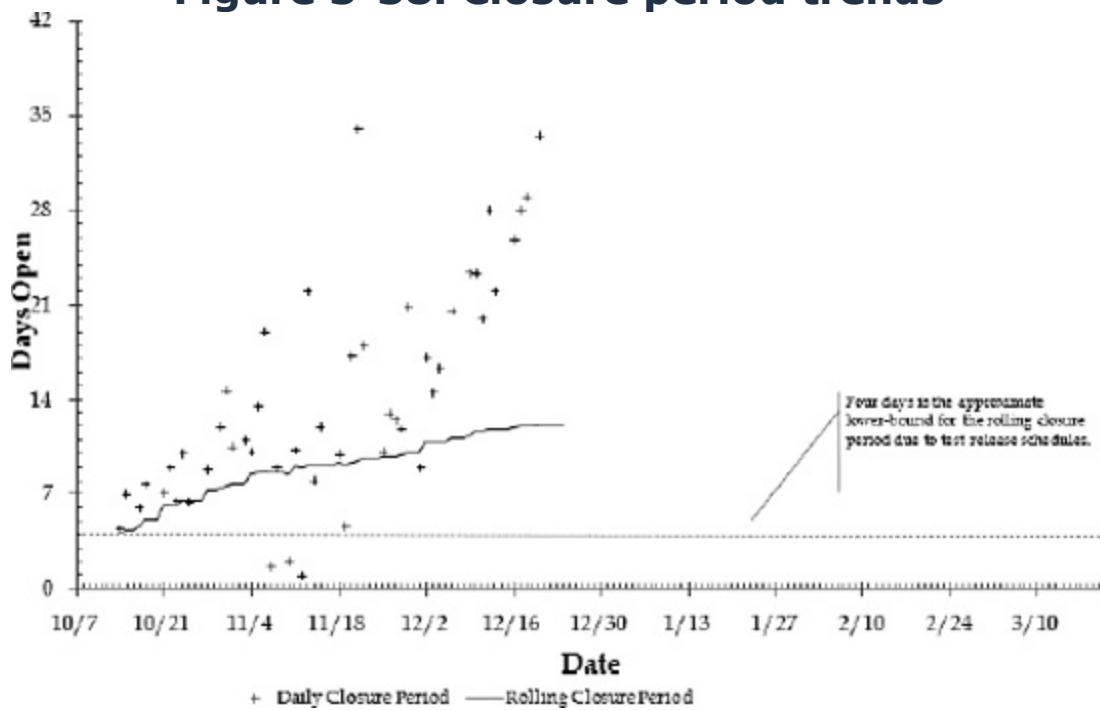


A Pareto analysis, by the way, is a histogram like this one where we look at the frequency of association between some underlying cause—in this case, the subsystem where a defect manifests as a failure—and the number of outcomes associated with that cause—in this case, the number of defects reported against the subsystem. The histogram is sorted so that the most frequent cause is shown to the left, and then causes are shown in decreasing frequency order to the right. It also includes a cumulative percentage across the top that shows the percentage of outcomes associated with the cause below and all the causes to the left of the cause below. A Pareto chart is a very useful analytical technique because it separates what J.M. Juran, its creator, referred to as the "vital few" causes from the "trivial many" causes. We should focus our attention on the vital few, not the trivial many, and this chart helps us do that.

[figure 3-38](#) shows the closure period trends. Closure period is shown in two ways on this figure:

- The rolling closure period value for any given date is the average number of calendar days from the date reported to the date resolved for all defects resolved *as of and before* that given date.
- The daily closure period value for any given date is the average number of calendar days from the date reported to the date resolved for all defects resolved *on* that given date.

Figure 3-38. Closure period trends



So, we would want the rolling closure period to stabilize at some number that is acceptable in terms of the turnaround time. Note that, as shown in the figure, the test release process puts a lower bound on how fast this turnaround time can be.

Once the rolling closure period stabilizes at an acceptable value, we would want the daily closure period to bounce around randomly right above and below the rolling closure period. A daily closure period that is consistently above the rolling closure period indicates something happening in the project that is causing it to take longer and longer to resolve defects.

3.6.3. Test Case (or Procedure) Metrics

Test case or procedure metrics are another very commonly used and often abused form of test metrics. (Because I tend to use the term test case more often than test procedure or test script, I'll use that term here, though different people tend to use different wording for these types of metrics.) One common test case metric looks at test case counts in terms of the total planned versus specified versus run. The tests that are run are sometimes further decomposed into passed, failed, blocked, and skipped.

Another common test metric looks at regression and confirmation test status. We can also evaluate confirmation test status using defect metrics, by looking at the number of times defect reports are reopened. Another

common metric looks at the test hours planned versus the hours actually achieved.

We can use these metrics to understand the project and the test and development processes. However, people tend to misuse metrics. One misuse is overloading, which is relying on a single test metric—perhaps augmented with a single defect metric—to tell the entire story of the test results, which completely omits the key questions of risk and coverage.

Another misuse is to pretend or assume that cumulative test case counts, percentage passed and failed, tell us the whole story in terms of product quality. The problem is that test cases are an abstraction. A situation where 40 percent of the tests fail might not necessarily be a product quality disaster, just as a situation where 95 percent of the tests pass could well be a product quality disaster. It depends on the specific tests that failed.

As with defect metrics, another classic misuse of this metric is to evaluate people. If we say that some developer is bad because a large number of tests run against their code failed, that can produce a strong defensive response by developers against all test results in the future.

Let's look at some examples of test case metrics. At the lowest level, we need some way to track the test cases. While many people use tools to do that, it's also true that many, many test managers still manage test efforts using the lowly spreadsheet as the test tracking method. Because all the capabilities that would be present in a complete test tracking spreadsheet can—and should—be present in a competent test management tool, I'm going to discuss the spreadsheet and leave it to you to map this to any tool you might be using.

Here in [figure 3-39](#) we see a portion of the test case summary worksheet for our hypothetical project. It shows the test cases in one of a number of test suites. (Remember, a test suite is a logical collection of test cases.)

Each column provides important information for each test case as follows:

- Owner: The person who is responsible for running the test case. We are using initials rather than names in this example.

Figure 3-39. Test case summary

- Test ID: A number where the integer digits are the test suite number and three decimal digits are the test case number. It uniquely identifies the test case, which is useful in your defect tracking system as well as any coverage tracking tool you might use.
- Test Suite/Case: This column contains first a test suite name, then the names of the test cases, row by row, in that test suite. In the full worksheet—which isn't shown here—you'd see a whole series of test suites, each with its corresponding test cases.
- Status: If we have run the test, the status is pass, warn, or fail. Pass means that we observed no failures. Warn means that we did observe a failure—and we filed a defect report—but the failure does not seriously impede the feature or function being tested. Fail means that we observed a failure—and filed a defect report—and the failure seriously impedes the feature or function being tested. If we have not and cannot run the test, the status is blocked. If we have not and will not run the test by choice, the status is skipped. If we have not yet run the test but we still plan to run it, the status column is blank.
- Test Environment: Provides identifying information for the hardware, software, and infrastructure used to run the test. Presumably, somewhere in another tracking worksheet or in the test plan it's possible to decode this identifying information to determine what exactly was in the test environment.
- Def ID (or Defect Identifier): Provides the ID number from the defect

tracking system for any defects reported against this test.

- Def RPN (or Defect Risk Priority Number): This is the defect risk rating, probably from the defect tracking system, analogous to the risk priority number in the quality risk analysis. A common way to calculate this is to multiply priority by severity, where priority is the business rating of the defect and severity the system or technical rating. In this example, we are using a 1 to 5 descending scale—i.e., larger numbers less bad—for both priority and severity.
- Run By: The initials of the person who ran the test.
- Plan Date: The plan date to run the test, defined during test implementation when the test manager determined the test sequence.
- Act Date (or Actual Date): The date on which the tester actually ran the test.
- Test Effort: The number of person hours required to run the entire written test from beginning to end.
- Plan Effort (or Planned Effort): The number of person-hours planned for this test. That might be more than the test effort, providing time for additional exploration in this area, or it might be less, meaning the tester is supposed to triage the conditions covered during testing.
- Actual Effort: The number of person-hours ultimately expended on the test. This might not match the planned effort, particularly if the test failed and the tester needed to expend significant effort to isolate and report the problems observed.
- Test Duration: The number of clock-hours required from start to end of test. For automated tests, this would be very different than the effort.

At this point, you have probably recognized that this is a detailed form of test-by-test test log, which complements the event-by-event test logging discussed earlier.

Based on the detailed information in the test case summary—the test-by-test test log—we can derive the test suite summary worksheet shown in [figure 3-40](#). Down the left side, you see two columns: the test suite name and the number of test cases it contains. Again, in some test log somewhere, we have detailed information for each test case. On the

middle-left side, you see four columns under the general heading of "Planned Tests Fulfilled". These are the tests for which no more work remains, at least during this pass of testing.

Figure 3-40. Test suite summary

The weighted failure numbers for each test suite, found in the column about in the middle of the table, gives a metric of how many bugs the test suite found. We weight each bug using the defect risk priority number from the previous worksheet. So, this weighted failure number measures technical risk, the likelihood of finding problems, associated with each test suite based on historical finding data.

On the middle-right side, you see four columns under the general heading of "Planned Tests Unfulfilled". These are the tests for which more work remains during this pass of testing. (IP, by the way, stands for in progress.)

Finally, on the right side, you see four columns under the general heading of "Earned Value". Earned value is a simple project management concept. It says that, in a project, we accomplish tasks by expending resources. So, if the percentage of tasks accomplished is about equal to the percentage of resources expended, we're on track. If the percentage of tasks accomplished is greater than the percentage of resources expended, we're on our way to being underbudget. If the percentage of tasks accomplished is less than the percentage of resources expended, we're on our way to being overbudget.

Figure 3-41. Test case completion

Similarly, from a schedule point of view, the percentage of tasks accomplished should be approximately equal to the percentage of project time elapsed. As with effort, if the tasks percentage is over the schedule percentage, we're happy. If the tasks percentage is below the schedule percentage, we're sad and worried.

In test execution, we can consider the test case or test procedure to be our basic task. The resources—for manual testing, anyway—are usually the person-hours required to run the tests. That's the way this chart shows earned value, test suite by test suite.

With the data captured in the test case summary and condensed in the test suite summary, we can then build two graphical representations of our test project progress. The first, shown as [figure 3-41](#), presents test case completion.

This chart has four main data sets, all plotted against a single axis on the left side. The first data set is the number of tests we plan to complete in a given pass of testing, shown as the "lightning bolts". The flat section in the middle of the lightning bolt occurs due to weekends, when we have scheduled no tests. Each lightning bolt starts over near zero—resetting the test completion data set—because we are running a sequence of similar test passes, using test repetition to manage regression risks.

Figure 3-42. Test hours

The second data set is the actual number of tests completed for the pass at the end of each day, shown as the boxes. The third data set is the number of those completed tests that have passed as of the end of each day, shown as the gray plus marks. The fourth data set is the number of those completed tests that have failed as of the end of each day, shown as the dark minus marks. Again, see that I've used notations to indicate interesting aspects of the data sets, especially why particular test passes are shorter than others.

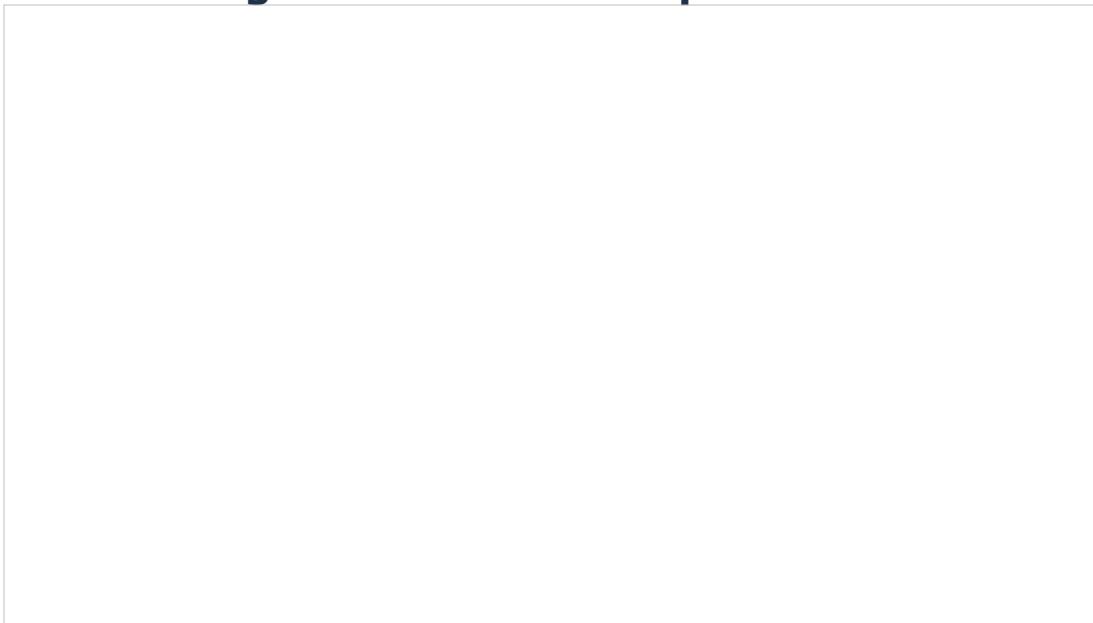
[figure 3-42](#) is the next chart we can build from the test logging data, which presents test hours progress. This chart has two main data sets, both plotted against a single axis on the left side. The first data set is the planned test execution hours per day, indicated by the dashed line. Note that the hours change first upward and then downward during the project, indicating a differing planned intensity of testing. The second data set is the actual test execution hours achieved per date, indicated by the boxes.

Two light gray lines above and below the dashed lines indicate where we think normal weekday progress is. In other words, the boxes can bounce around randomly above and below the dashed lines, provided they stay within the gray lines. For those boxes outside the gray lines, we've provided an explanation of why.

[figure 3-43](#) shows the confirmation test failure rate, but from analysis of defect metrics. In our defect tracking system, we have a counter that keeps track of how many times we opened a particular report. If the defect is

found, fixed, confirmed as fixed, and never seen again, that count is one.

Figure 3-43. Defect open count



However, if the defect is found, allegedly fixed, fails the confirmation test and is reopened, then actually fixed, confirmed as fixed, and never seen again, that count is two. Additional looping on the reopen action would further increment the counter.

In this example, we reopened only 4 percent of the defect reports once or twice. We reopened only 1 percent twice. None are reopened more than twice. So this project is in good shape in that area.

3.6.4. Test Progress Monitoring and Control Exercise 1

[figure 3-44](#), [figure 3-45](#), [figure 3-46](#), and [figure 3-47](#) on the next few pages show defect and test charts at the end of iteration one of HELLOCARMS testing. Based on what these charts are telling you about the HELLOCARMS testing, outline a list of three to five improvements for the subsequent iterations. How would the charts look at the end of iteration two if the improvements work?

Figure 3-44. Cumulative defects reported and resolved at the end of iteration one



Figure 3-45. Closure period at the end of iteration one

If you are working in a classroom, break into groups of three to five. If you are working alone, you'll need to do this exercise yourself. If you are working in a classroom, once each group has finished, discuss the results.

I suggest allotting 45 minutes for this exercise, including the discussion.

Figure 3-46. Test completion at the end of iteration one

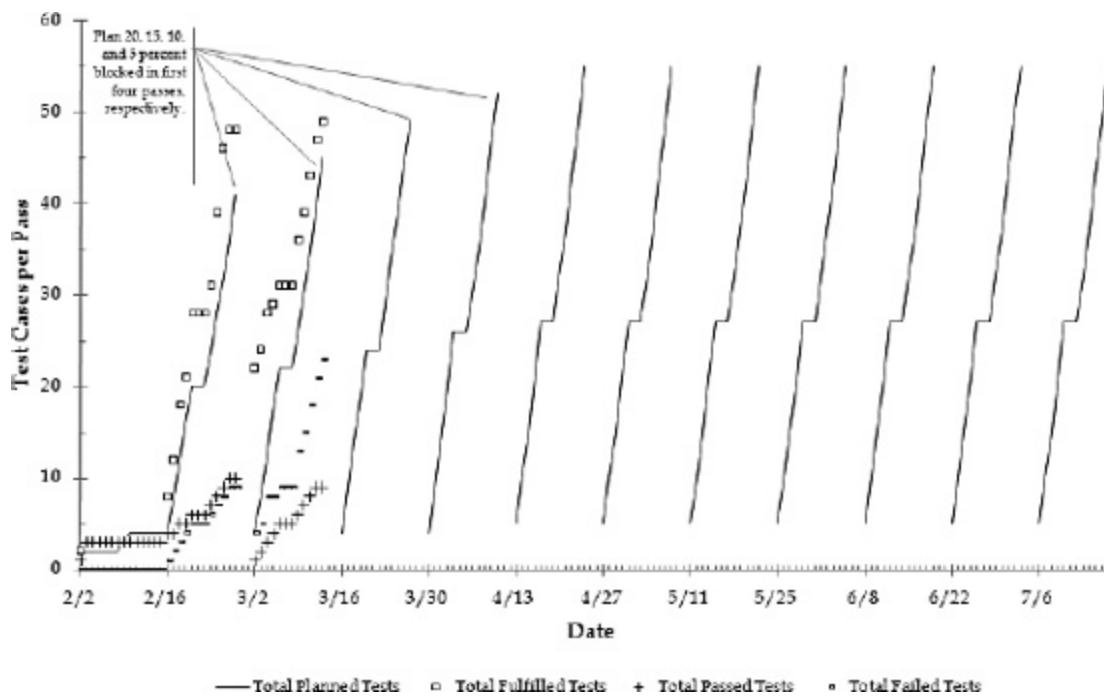


Figure 3-47. Test hours at the end of iteration one

3.6.5. Test Progress Monitoring and Control Exercise 1 Debrief

Looking at the charts, I noticed four areas for further investigation and possible improvement:

1. The cumulative defects reported/resolved chart shows a large jump in defects right at the end of the test period.

Possible improvement: Stabilize the product earlier in the iteration, say around week 3, and focus on bug fixing in the last half of testing for that iteration.

2. The closure period chart shows a steady upward trend in the closure period throughout the test period.

Possible improvement: Limit the size of subsequent iterations in terms of new functionality and focus on quick elimination of bugs.

3. The test case completion chart shows a large number of tests added in the second and third passes (i.e., the last two passes) of this iteration, with a large number of tests failing during the final pass.

Possible improvement: Try to run all tests in the first pass to identify as many problems as possible; then focus on resolution of defects so that most tests pass by the end of the last pass of the iteration.

4. The test hours chart shows limited traction for testing during the first pass of testing, with a late push in testing in the third and final pass.

Possible improvement: Remove any obstacles to efficient testing and completion of test hours during the first pass.

You might have noticed additional possible improvements based on these charts as well as further research to do to understand the reasons for the problems.

3.6.6. Test Coverage Metrics

Test coverage metrics are less common than test case metrics and defect metrics. One type of coverage metric, perhaps the most common, reports on requirements and design coverage. To produce such a metric, you need traceability from your tests back to the requirements or design elements they cover; i.e., bidirectionally between the test cases and each item in the test basis. The test cases, the traceability information, and the test basis documents themselves each must be fine-grained enough to allow this traceability to be meaningful. For example, if you have a single piece of paper labeled "System Requirements" with some hand-scribbled drawings on it and a collection of exploratory test charters that each mention the system requirements in a Test Basis section, that's not meaningful traceability and you won't get meaningful coverage metrics. (I'll discuss exploratory test charters more later in this chapter.)

Another type of coverage metric reports on risk coverage. You saw an example of that type of metric at the beginning of this section, with the pie chart consisting of dark, medium, and light gray sections. Again, for such a chart to work, you would need meaningful and sufficiently fine-grained traceability between test cases (or procedures) and the quality risk items.

Yet another type of coverage metric is the kind that looks at environmental or configuration coverage. This type can be particularly useful for shrink-wrap software and other mass-market software or systems that customers will use in many environments. You need to capture, for each important environmental factor, what option you selected for the test. For example, if your application runs on Windows PC, you might want to capture the specific version of Windows and even the service pack levels. In some cases, combinational issues like pairs of options, triples of options, and so forth are important, but you typically want to avoid getting caught up in combinatorial explosions. This topic of combinational testing is covered in the companion volume on the Advanced Test Analyst exam.

Another final example of test coverage is code coverage. The types of code coverage metrics typically used were discussed in the Foundation syllabus in [chapter 4](#). Unlike other forms of coverage measures discussed so far, code coverage is almost never measured with a traceability table or database but rather with a tool that keeps track of the total code coverage achieved cumulatively as test cases are run.

These metrics are important because they allow you as a test manager to establish whether what needed to be tested will be or was tested. In the case of tests having been run against some requirements element, risk item, or configuration, you can then use your traceability to trace the test result—pass, warn, fail, or block—back to the underlying item you are trying to test, reporting its status.

Finally, note that these coverage metrics can serve as a surrogate metric for confidence. We can roughly measure the level of confidence we should have in a system by establishing to what extent we have covered it with tests and to what extent what we have covered actually works.

A mistake that organizations make when they use coverage metrics as a surrogate metric for confidence is to rely on a single factor, like requirements coverage or code coverage, without considering other measures of coverage. Ideally, you would use multifactor coverage metrics—checking code coverage, requirements coverage, and risk coverage—

before deciding that you should have a high level of confidence in a system. For mission-critical and safety-critical systems, using multifactor coverage measures is a smart approach.

Figure 3-48. Risk coverage



3.6.7. Risk Coverage

The chart in [figure 3-48](#) analyzes the relationship between the key quality risk categories, the degree of risk coverage we've achieved for each category, and the number of defects we've found related to each risk category.

At the bottom, we see labels corresponding to each major risk category. We need traceability from our test cases back to the risks from which they were derived. We use that traceability, at the coarse-grained, risk category level, to create this chart. We will also need to classify our defect reports in terms of the risks to which they relate, and we'll use that traceability to create this chart too. If we have more fine-grained traceability between the specific test cases and the specific risk items, and between the specific defect reports and the specific risk items, we can produce a detailed report for each quality risk category, probably in a tabular form, showing passed tests, failed tests, resolved defects, and unresolved defects for each risk item.

We use the test coverage traceability information to plot the percentage of

planned risk coverage achieved in each category, as shown by the gray bars. As the gray bars grow toward 100 percent, the amount of residual risk in this category goes down because there is less and less we haven't tested yet. At 100 percent, we've reached the planned risk mitigation for this category. Assuming our risk analysis was good, there aren't any more unknowns left to explore and the only risk for this category relates to known defects. Note that we have no planned tests for "other" because we use this risk category to classify defects we find that do not trace to an identified risk. This category helps check our quality risk analysis during test execution, as I mentioned earlier in this chapter.

We use the defect coverage traceability information to plot the percentage of defects related to each category, as shown by the dark gray bars. These bars, if stacked on top of each other, will always total 100 percent. If the defect bar is high in comparison to the test bar, then that indicates significant risk because there are a large number of defects found already, even though we've run only a few tests. Conversely, if the defect bar is small next to a large test bar, then the risk is low because we've achieved a lot of test coverage and there aren't many defects here. If the defect bar over the "other" category is above 5 percent, we should start to question our quality risk analysis. If it is above 10 percent—in other words, more than 1 in 10 defects occurred in any area where we didn't expect to find many defects or where we said defects would be unimportant—then we should definitely revise our quality risk analysis.

3.6.8. Use of Metrics

You can use all of these metrics in a variety of forms. You can use a verbal narrative. You can use a detailed or summary numerical table. You can use visual graphics. You can determine the proper form to use in part by the purpose of the metric and the purpose of its use.

- Do you intend to analyze the project, process, or product, perhaps to discover or highlight some specific area of concern?
- Do you intend to report your test results to interested project and product participants and stakeholders?
- Do you intend to make control decisions based on the metrics, to adjust the course of the testing or the project as a whole, and to monitor the results of that course correction?

As a test manager, it's important to remember that, if the recipients of

these metrics are anyone other than yourself, the proper gathering, analysis, and reporting will depend on the audience. You should consider their needs, goals, and abilities.⁵

⁵ Three books that can help you learn to make better use of metrics and to present them better are Edward Tufte's *The Visual Display of Quantitative Information*, *Envisioning Information*, and *Visual Explanations*. I found that reading these books and putting Tufte's ideas into action greatly improved my ability to analyze and present ideas.

For example, if you were sending information about unresolved defects, you would send different reports to a programming lead and to a marketing manager. Managers' needs differ from technical staff.

As another example, if you were sending information about unresolved defects, you might send different reports to a project manager and a business-side project sponsor. A project manager might want detailed information on unresolved defects, while the business manager would prefer to know the status of the product risks.

If we think that the metrics from testing indicate some kind of problem, we have various options to resolve the problem and bring the project back under control. We can revise test road maps like quality risk analysis documents, test plans, or test priorities. For example, if we find that project deadlines have changed and will not allow completion of the tests, we can reduce test coverage back on our risk analysis and adjust downward the priority of the associated risk items.

We can add test resources or overall test effort. For example, if we find we don't have enough skilled people or systems to do performance testing, we could retain an outsource testing facility to do those tests for us.

We can decide to delay the release date to allow for more testing. This is seldom a popular option and is often only chosen when no other option exists.

A common decision, one that I see taken on many projects, is to change the test exit criteria if the release date or other project priorities are threatened. For example, you might relax a criterion that requires the defect discovery rate to be zero for the last two weeks of testing, particularly if the number of defects discovered was small and the particular defects of minor importance. In some cases, the exit criteria are strengthened, but I have to say I've never seen that happen.

While the test manager can—and often should—outline the project team's options as part of reporting test results, and the test manager can and often should provide the advice on which option to select, the actual selection and implementation of one or more of these options is typically outside the test manager's remit. It usually requires the consensus, consent, or at least direction of one or more product and project stakeholders and the manager. As a test manager and consultant, I have seen that making unilateral decisions that affect strategic company interests is a great way to be invited to find a new job at some other company.

Figure 3-49. Example: Test progress monitoring

	A	B	C	D	E	F	G	H	I	J
1	System Test 2.5 (CSA) Test Suite Summary									
2										
3										
4										
5	Suite	Total Cases	Block	Done	Fail	Weighted Failure	Pass	IP	Skip	In Queue
6										
7	Stress and Capacity	16	4	12	2	3.44	8	0	2	0
8	Performance	8	1	7	7	1.00	0	0	0	0
9	Error Handling/Recovery	18	5	13	7	1.53	5	0	1	0
10	Data Quality	26	4	22	9	4.53	13	0	0	0
11	Functionality	23	1	22	7	1.06	14	0	1	0
12										
13		91	15	76	32	11.56	40	0	4	0
14			16%	84%	35%	N/A	44%	0%	4%	0%

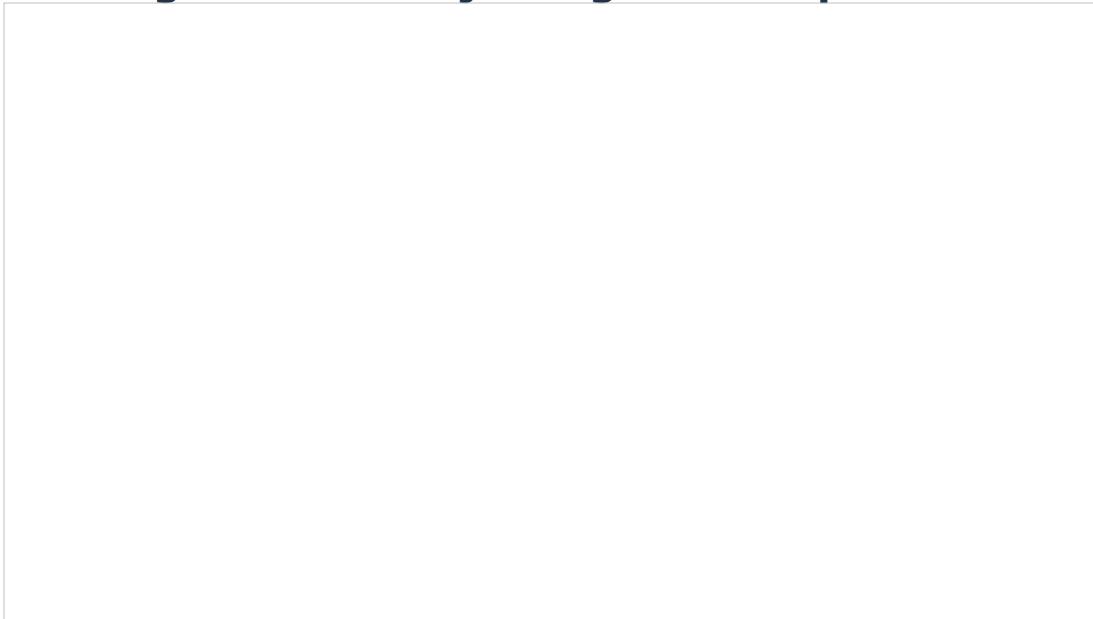
Figure 3-49 shows an example of the test suite summary used to monitor test progress. You'll notice that this figure shows a somewhat simpler and less-detailed variant of the test suite summary shown earlier. This is a test suite summary for the IVR project that I've mentioned a few times. It was captured during system testing of the Customer Service Application (CSA). You saw an example of the Failure Mode and Effect Analysis chart for this system earlier in this chapter.

On the test suite summary, you should note the large number of blocked and failed tests. Remember, blocked tests are those we would like to run but can't. At this point, to get testing back on track, it was critical for the network operations team to resolve the blocking issues (which were mostly environmental) and for the CSA developers to fix the defects that were causing test failures.

On the IVR project, it wasn't only the CSA application that had some serious trouble. The IVR application—the part that handled the incoming phone calls from the clients—violated system test entry criteria and then suffered many, many problems during system testing. About halfway through the system test period, I wrote the email you see in figure 3-50 to the project

manager. You can see that I am suggesting, based on the defect reporting metrics, to consider a delay in System Test exit and System Integration Test entry.

Figure 3-50. Adjusting the test process



The response from the project manager is not unusual. Project stakeholder bonuses became attached to the schedule at some point. Those bonuses introduced tremendous political pressures to achieve the schedule and explains why the project sponsor "didn't want to hear" about the problems with IVR application testing.

So the project manager declared, in his response, that schedule was paramount, that all must work to preserve exit and entry dates, and that my test team needed to work within those parameters. Okay, that was clear direction. I wasn't too happy about it, but it did make clear that we could address obstacles to resolving blocked or failed tests only in ways that would not endanger the schedule.

Does this email thread indicate that the project manager and project sponsors were bad people? Well, as I said, I wasn't happy with the situation at the time. However, on reflection, I realize that they were both responding completely rationally to financial incentives that were put in place by the company board of directors. By creating bonuses tied entirely to schedule, and by placing hard caps on the project budget, and by further having no bonus or acceptance criteria related to considerations of quality or even completeness of features, the board of directors set up for themselves the outcome that ultimately occurred: Every attempt was made

to squeeze the product into production, including settling for limited features and functionality and deferring a large number of quality problems, some quite serious.

That attempt failed, and eventually the bulk of the senior project stakeholders were fired. Ironically, one of the stakeholders, an outside contractor that supplied an extremely buggy component, not only ended up retaining the company as a client; one of their owners took over as a contractor-officer of the company, running their technical operations, after the senior project stakeholders were fired. Life is sometimes funny that way.

So, what do we do when test reality diverges from our carefully crafted test plan? As test managers, we must define and execute one or more test control options. We want to attempt to achieve the best possible outcome within the given constraints. What possible test control measures might we have at our disposal? As with project control, they are quite similar.

You can revisit the priority of test cases or even cancel some tests. You can do this based on your risk analysis or based on a constraint. If the hardware you need to run a particular test simply won't be available, then you might as well cancel that test.

You can trigger contingency plans defined in your test plan, if the divergent reality is one you anticipated and planned for.

You can try to obtain additional resources, if you have budget flexibility but not schedule flexibility. Conversely, if you have schedule flexibility but not budget flexibility, you might try to extend the test end date or release date. But don't forget the training time and cost.

If neither schedule nor budget can bend, then you can try to change the scope, either of the project as a whole or of testing only.

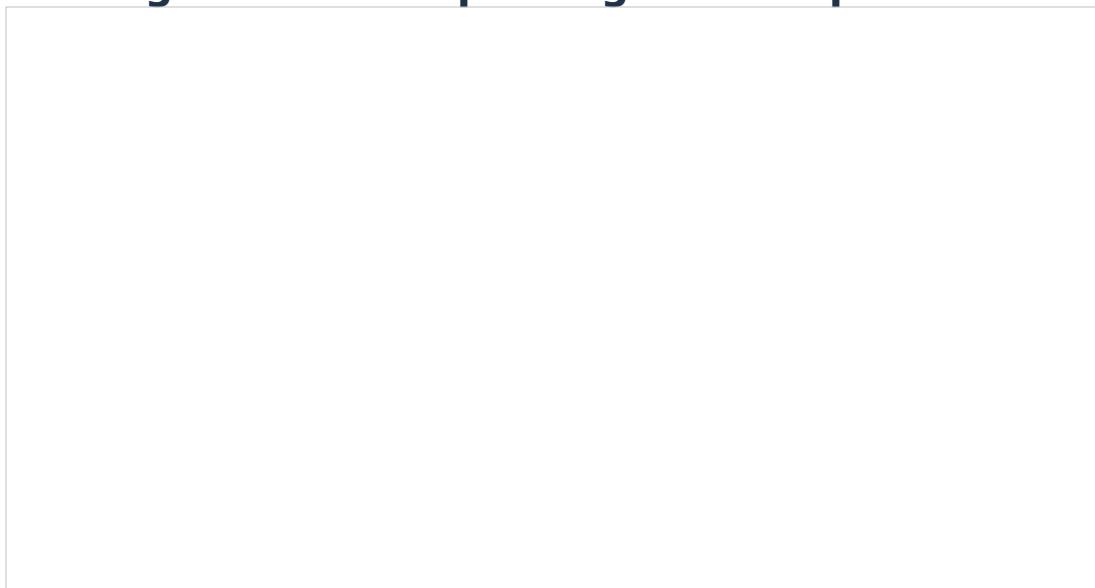
Finally, you can try to revise the test exit criteria to make it easier to "declare victory".

As with project control, the chosen options might require consensus, consent, or approval from one or more project stakeholders, managers, or sponsors. Be careful not to get into a situation where you silently decommit from some testing tasks, test coverage, or other responsibility you had to the project without adequate approval. It's an easy thing to do when you're working late on a Friday night, but don't succumb to the temptation

3.6.9. Two Case Studies of Test Reporting

figure 3-51 shows an email I sent on the Internet appliance project to the two client contacts we had. The problem was this: We had not received a requirements specification sufficiently detailed for us to develop all of our tests in any of the major product areas, which were provisioning, server, and client. This led to some delays in finishing our test design and implementation, to the point that informal test execution had started. So in this email I suggested a way to triage the test case development, based on quality risks. We would allow a decreasing level of rigor and detail in our test cases, based on criticality.

Figure 3-51. Improving the test process



In response to this email, one of the client contacts asked for some metrics on the number of tests not yet completed. I was able to provide that detail. Based on this information, we were able to proceed with test completion. However, the meeting I mentioned at the end of the email did not occur. Instead, we were instructed to do the best we could with the information available. So we adopted a "take your best guess" policy toward both test specification and test execution. This policy resulted in a rather large number of false positives during test execution, which in effect turned the bug reporting process into a retroactive requirements specification process.

Table 3-17 shows an example of a portion of a test results summary report we sent toward the end of system testing on the Internet appliance project. Specifically, this table is an excerpt of part of our Test Team Must-Fix Defect List report. Our two client contact managers asked us to list those problems that we thought were serious enough in their quality implications

to make it unwise to proceed with delivering the system.

Table 3-17. Example of a test results summary

Bug ID(s)	The Problem	The Quality Impact
153, 2259	Lack of Java support.	Some of the coolest stuff on the Web [won't work]. See also other bugs reported against the Mall...
825	Moving the device is not handled well.	[The provisioning] processes should be in place before we go live, and how far is this behind client software release?
43, 984, 1934, 2191, 2510	No working audio...	We have speakers. The customers will expect them to work.
1145, 1405, 2417, 2543	Security is poor.	Before we send this thing out into the world like a lamb amongst wolves, it needs to be secure. If this product succeeds, competitors and hackers will test the security.
1204, 1479, 2056, 2330	Dial-up unreliable.	[Dial-up has not worked well outside of Austin.] We are concerned that this service won't scale.
1252	No process to change user's email address.	Marriage, divorce, sex change? Whatever the reason, people should be able to change email addresses.

Normally, as a test manager, I prefer not to produce any kind of "must fix" list of bugs. I believe that job best done by a cross-functional team of project stakeholders. However, in this case, the client was asking for our opinion, and so we gave it. We had a good, relaxed relationship with these two managers, so we were able to use humor in our report. I've omitted another 15 or so line items because they don't add much to the list from the point of view of understanding what we did and why.

Notice that this is not a raw report from the bug tracking tool. Rather, we have analyzed the problems to look for common failure modes and then

assessed the customer impact of those failure modes. The details are accessible by reviewing the specific, cross-referenced bug reports.

3.6.10. Test Progress Monitoring and Control Exercise 2

Figures 3-52, 3-53, 3-54, and 3-55 on the next few pages show the same defect and test charts as the previous exercise, but this time at the start of the last week of testing for iteration two. Analyze these results to answer two questions:

- 1.** Do they show evidence for or against the improvements you outlined in the previous exercise?
- 2.** Do you believe the test results for iteration two show HELLOCARMS on track for deployment at the end of this iteration?

If you are working in a classroom, break into groups of three to five. If you are working alone, you'll need to do this yourself. If you are working in a classroom, once each group has finished, discuss the results.

I suggest allotting 60 minutes for this exercise, including the discussion.

3.6.11. Test Progress Monitoring and Control Exercise 2 Debrief

Let's start by assessing progress in terms of the four improvements that I suggested in the last exercise.

Figure 3-52. Cumulative defects reported and resolved toward end of iteration two

Figure 3-53. Closure period toward end of iteration two

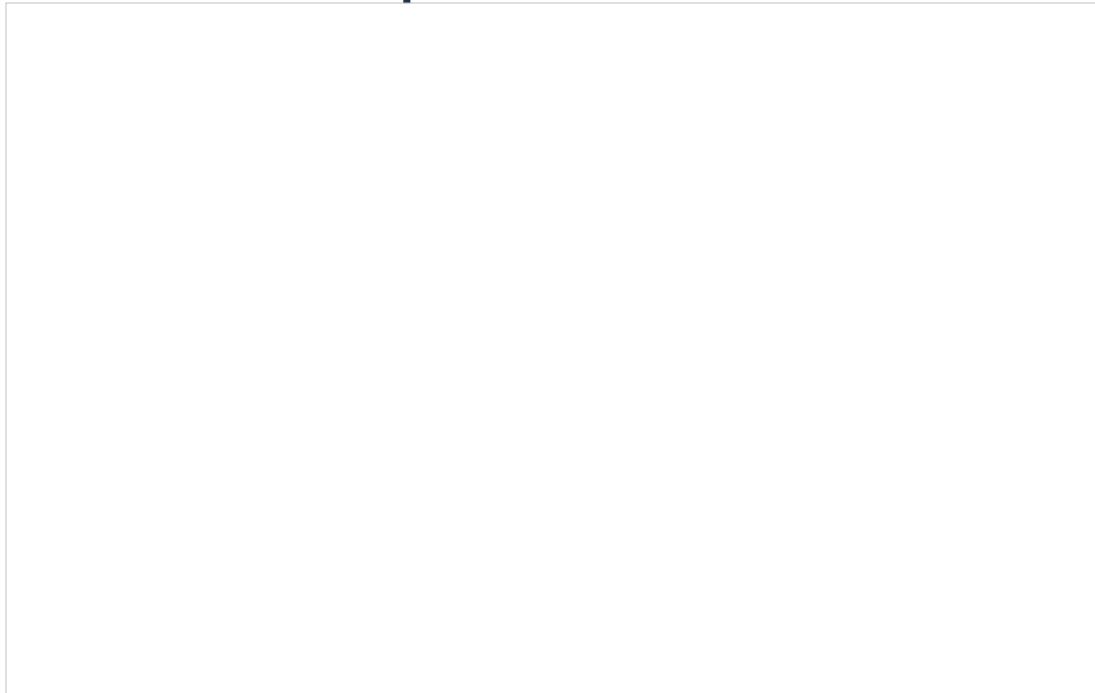
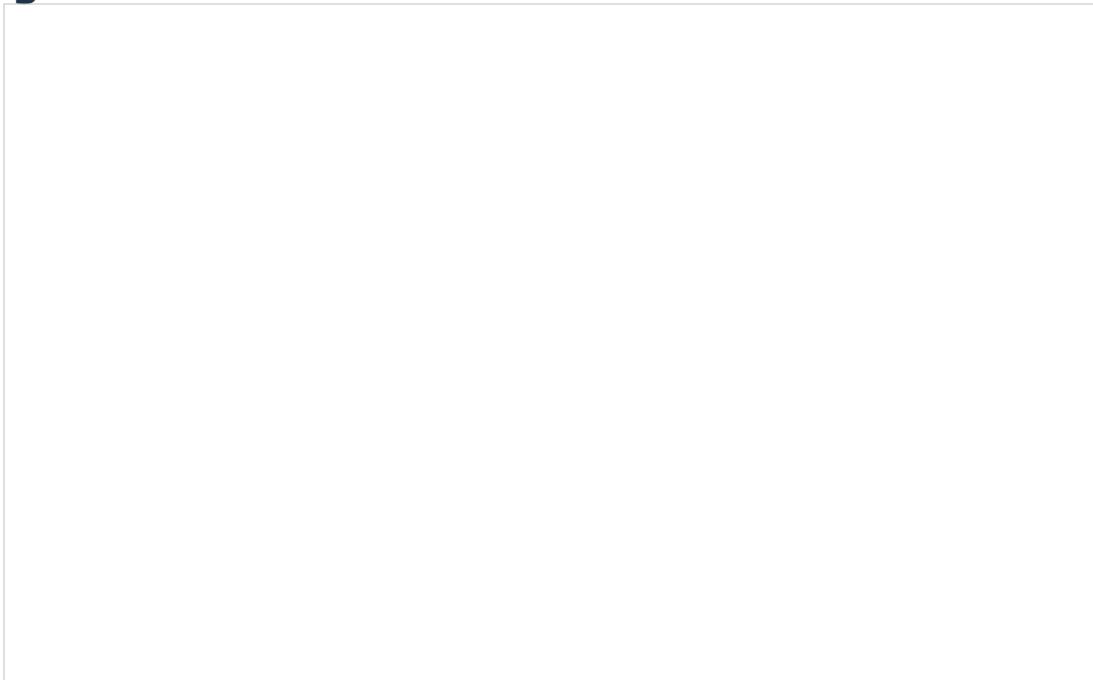


Figure 3-54. Test completion toward end of iteration two

Figure 3-55. Test hours toward end of iteration two



1. The cumulative defects reported/resolved chart for iteration one showed a large jump in defects right at the end of the test period.

Possible improvement: Stabilize the product earlier in the iteration, say around week 3, and focus on bug fixing in the last half of testing for that iteration.

Status for iteration two: While development has done an excellent job

of resolving defects—indeed, achieving convergence with the reported defect count multiple times during the iteration—the defect reporting rate remains extremely high throughout the entire test execution period for iteration two. The test manager and the project management team as a whole should carry out further investigation to discover the root causes of this disturbing outcome. There are five possible explanations:

1. High regression rate of the features in iteration one due to the high rate of defect repair during iteration two
 2. High rate of discovery of new bugs in features from iterations one and two due to an expansion of test scope
 3. High rate of new feature addition during iteration two
 4. Large number of features added in iteration two brought with them a large number of defects
 5. Insufficient upstream quality control activities, such as requirements and design reviews, code reviews and static analysis, and unit testing
2. The closure period chart for iteration one showed a steady upward trend in the closure period throughout the test period.

Possible improvement: Limit the size of subsequent iterations in terms of new functionality and focus on quick elimination of bugs.

Status for iteration two: We cannot conclude that the size of iteration two was limited, as that remains a possible explanation of the high rate of bug discovery throughout iteration two. However, the development team has certainly done an excellent job of reversing the negative trend in backlog and closure period. That said, it is possible, as mentioned above, that the development team is working too quickly and with insufficient attention to detail to resolve defects, resulting in a high regression rate.

3. The test case completion chart for iteration one shows a large number of tests added in the second and third passes (i.e., the last two passes) of this iteration, with a large number of tests failing during the final pass.

Possible improvement: Try to run all tests in the first pass to identify as many problems as possible, then focus on resolution of defects so that most tests pass by the end of the last pass of the iteration.

Status for iteration two: The test team did a fine job of accelerating the rate of test completion and the total number of tests completed in each pass. However, we still see a large number of tests failing. This problem is especially acute during the last pass, where more tests have so far failed than passed. This is, of course, consistent with the results shown for iteration two in the cumulative defects reported/resolved chart.

4. The test hours chart for iteration one showed limited traction for testing during the first pass of testing, with a late push in testing in the third and final pass.

Possible improvement: Remove any obstacles to efficient testing and completion of test hours during the first pass.

Status for iteration two: Again, the test team did a fine job of ensuring complete traction and engagement with the testing immediately at the start of iteration two test execution. The hours are a bit higher than planned throughout the iteration. This increase is consistent with the results shown for iteration two in the test case completion chart and the cumulative defects reported/resolved chart as an unexpectedly high rate of failing tests and reporting defects will increase required test effort.

So, based on this analysis, we can conclude that HELLOCARMS is not on track for deployment at the end of iteration two. The cumulative defects reported/resolved chart indicates that a large number of latent defects remain in the system. The test case completion chart indicates that a large number of features are failing.

3.7. Business Value of Testing

Learning objectives

(K2) Give examples (measures) for each of the four categories which determine the cost of quality.

(K3) For a given context, list the quantitative and/or qualitative values of testing that apply.

There are some people in the testing business—even some that I respect, such as Boris Beizer—who say, "Don't try to build a business case for testing." They think it is a waste of time, an exercise the results of which won't be accepted by managers anyway. That has not been my experience. I have found that a conservative, well-explained business case can persuade and influence stakeholders. I also have found that it's essential to changing the way testing is viewed.

Those of us in the testing world have a problem something like that of organized religion in the United States. In the United States, according to sociological surveys, there are very few atheists; i.e., people who will say, often and with conviction, that there is no God. There are agnostics, of course, people who aren't sure, but for the most part people say that they believe in some God or other.

However, if you then ask them, "When was the last time you were in a mosque, a synagogue, a church, a temple, or other place of worship?" many of the answers would along the lines of "my cousin's wedding", "my uncle's funeral", and "my nephew's bar mitzvah", but less frequently would you hear, "Oh, I'm there every week", and even less frequently, "I go three or four times a week". In other words, attendance at organized religious ceremonies is not a priority for many people. The financial situation of many of religious sects in the United States is correspondingly bleak. Many people in the United States are passively religious.^[1]

[1] See, for example, religions.pewforum.org/reports.

And so it is with testing. There are very few "testing atheists", those who would say that testing has no value. Most would say that testing has some value. However, when you ask them what they are willing to spend, how much delay they are willing to incur, what features they might be willing to forgo, for the reduced risk, confidence, long-term savings, and information

that testing can provide, you find very quickly that most people in IT are passively religious about testing.

I believe this problem arises from two main causes. First, very few managers, even seasoned test managers, can quantify, describe, or articulate the value of testing to peer managers, senior managers, or executives. Second, many testers happily spend time in their comfort zone, their zone of competence, dealing with tactical issues like finding bugs, writing test cases, and setting up test environments. They avoid the broader strategic issues that would allow them to connect what they do as testers to things that are important and valued by the organization.

So, I suggest that we, as testers, start to internalize the following three principles of testing business value:

- First, what gets measured gets done.
- Second, what gets valued gets funded.
- Third, testing has a value that we can measure.

If you can accept those three things as true, you are on a track toward attaching what you do as a tester to what organizations care about, and thus having fewer funding difficulties.

Running tests, by itself, has no value. It might sound like I'm contradicting something I said a moment ago, but I'm not. What I mean is that executing tests—pounding away on keyboards and databases, observing screens, checking reports, and so forth—has no intrinsic value. Testing has value when it connects with some other goal or objective of the organization.

Some of these goals are quantitative and quantifiable:

- If the organization wants to reduce long-term defect-related costs, we can help by finding must-fix defects before release.
- If the tech support or help desk staff are tired of being surprised by unknown issues, we can help by finding less-critical, deferrable defects before release.
- If the organization is concerned about the risk associated with releasing software, we can reduce the risk by running tests, whether those tests find bugs or not.

- If the organization wants to make confident, informed release decisions, we can help by delivering information.

Some of these goals are qualitative but nevertheless very real to those who hold them:

- If the organization wants to enjoy the marketing and sales advantages of an improved reputation for quality, we can help throughout the lifecycle.
- If the organization wants to have smoother, more predictable release cycles, we can help, and we can best help by being involved throughout the life-cycle.
- If the management team wants to have increased confidence upon release, we can help by working with them to ensure that what really matters gets tested—e.g., by using risk-based testing.
- If the organization wants protection from legal liability due to regulations or liability issues, we can help by ensuring compliant, defensible testing.
- If we are in the safety-critical or mission-critical business and the organization wants to ensure the lowest possible chance of lost missions or lives, we can work with the project team throughout the lifecycle to filter out as many problems as we can prior to release.

What matters to your organization, on your project, right now? If you can't answer that question, you're probably not focusing on the right things. Test managers should understand which of the quantitative and qualitative values apply and be able to communicate about these values.

3.7.1. Cost of Quality

There are techniques available to quantify most of the quantitative values of testing. However, the Advanced syllabus—and thus this book—focuses on cost of quality to measure the quantitative value and efficiency of testing. Cost of quality is sometimes called cost of poor quality, to remind people that it is an analysis that shows that poor quality costs money and good quality saves money.^[1]

[1] In this book, as in the Advanced syllabus, we cover only cost of quality. However, if you want to learn more about measuring the other

quantitative values of testing, I recommend my book *Critical Testing Processes*.

At its core, cost of quality is an accounting technique that classifies project or operational costs into four categories.

- The first category includes the costs of prevention. Costs of prevention are expenses we incur to prevent bugs from happening. Expenditures for training developers are costs of prevention.
- The second category includes the costs of detection. Costs of detection are expenses we incur to detect bugs and that we would have to incur even if we didn't detect any bugs. Expenditures for test planning, analysis, design, and implementation are costs of detection, as are some of the costs of test execution (for tests that pass) and test closure.
- The third category includes the costs of internal failure. Costs of internal failure are expenses we incur because we did detect bugs. The remaining expenditures for test execution, specifically rerunning tests, confirmation testing fixes, installing the second and all subsequent test releases, and the like, are costs of internal failure, as are the costs associated with programmers fixing bugs and with other project support staff, like release engineering creating test releases.
- The fourth category includes the costs of external failure. Costs of external failure are expenses we incur because we didn't detect and remove all the bugs before release. Half or more of technical support or help desk costs often arise from external failure costs. Programmers who spend time on fixing field failures are incurring an external failure costs. In some cases, there are tangible punitive or lost-business costs associated with external failure too. For example, some organizations have service level agreements with customers, with penalties associated with failures. Some organizations have to worry about potential customers choosing not to do business with them, such as when an e-commerce site goes down.

When RBCS does testing assessments, we typically measure the efficiency of testing as a bug finding operation by using cost of quality. In every case I have looked at as a consultant, the costs of detection and internal failure were much cheaper than the costs of external failure. Testing is thus an excellent value. I have yet to find a case where testing was not saving the company money on potential costs of external failure that were avoided by

the prerelease detection and removal of defects.

Table 3-18 shows cost of quality analysis applied to the Internet appliance project I've used as a case study a few times in this book. We'll start at the upper left of the table and then work our way down, across, and down again to see how the calculations work.

Table 3-18. Example: Cost of quality ROI

<i>Detection Costs</i>		<i>External Failure Costs</i>		
Test Budget	\$1, 000,000		Sustaining Costs	\$3, 000,000
Future Value of Assets	100, 000		Percentage Bug Related	50%
Retest Costs	500, 000			
Net Detection Costs	\$400, 000		Net Ext. Failure Costs	\$1, 500,000
Must-Fix Test Bugs	1, 500		Must-Fix Released Bugs	500
Detection Cost per Bug	267		Ext. Failure Cost per Bug	3, 000
<i>Internal Failure Costs</i>	<i>Return on Investment</i>			
Test Bug Fix Costs	750, 000		Must-Fix Test Bugs	1, 500
Re-test Costs	500, 000		Money Saved per Bug	\$1, 900
Net Int. Failure Costs	\$1, 250,000		Net Benefit of Testing	\$2, 850,000
Must-Fix Test Bugs	1, 500		Net Detection Costs	400, 000
Int. Failure Cost per Bug	833		Test ROI	713%

I first calculate the costs of detection. As I mentioned, a portion of the test budget forms the cost of detection. So I start with the test budget of \$1, 000,000. Next, I deduct from that figure the time spent creating and

capturing assets of value for subsequent projects (basically, costs of test closure). Finally, I remove the retest costs—the costs of confirmation testing bug fixes and regression testing features that might have been affected by a bug fix—since those are costs of internal failure. That leaves \$400, 000. As there were 1, 500 must-fix test bugs, that means our average cost of detection was \$267.

Next, I calculate the costs of internal failure. This includes the remainder of the test budget, the costs of retesting, which you see added back here. There is also the developer effort spent fixing the bugs, which I estimated based on the three months of system test during which the product was feature complete. The net internal failure costs were \$1.25 million, with an average cost of internal failure of \$833. Thus, the average prerelease cost per bug—both detection and internal failure—was \$1, 100. If a bug found after release costs more than \$1, 100, testing is saving the organization money.

Let's move to the right side of the table to see if we are saving money and, if so, how much. I first calculate the external failure costs. The sustaining costs—including developers, testers, and technical support—during the first six months of release were \$3, 000,000. I estimated that about 50 percent of that cost was due to bugs. (I actually think the true percentage might have been much higher, but as I said before, we want to keep these business cases conservative.) The net external failure costs were \$1.5 million, with 500 bugs found after release. That gives us an average postrelease bug costs of \$3, 000, meaning each bug found by test saved \$1, 900.

Finally, let's calculate our return on investment. Because each bug found saved \$1, 900 and 1, 500 bugs were found, testing saved the organization \$2.85 million. To put that in percentage terms, our cost of testing for this project—if we deduct closure costs and bug-related retesting costs—was the cost of detection, \$400, 000. So, if we divide \$2.85 million—the net benefit of testing—by \$400, 000, we get 713 percent. Not too shabby.

3.7.2. Other Values for Testing

While the Advanced syllabus focuses on cost of quality, let's briefly discuss some of the other quantitative and qualitative values delivered by testing.

Based on some further analysis, I found the following additional quantitative values for testing on the Internet appliance project:

- \$150, 000 in saved support-call time due to known bugs (often with test-identified workarounds)
- \$250, 000 in risk-reduction value from passed tests, based on the cost of a failure in the field and the likelihood and frequency of tests failing the first time they were run
- \$200, 000 in information value, based on the reduced risk of project failure due to bad project tracking^[1]

[1] I describe these values and the analysis behind them in an article on the RBCS Web site, "What IT Managers Should Know about Testing ROI". Capers Jones describes the risk of project failure due to bad project tracking in *Estimating Software Costs*.

There were also some additional qualitative values delivered on that project. For one thing, the organization was able to have some confidence on release, particularly in areas we had managed to successfully clean up, such as performance and reliability. (There remained a fair number of bugs, as you can see, since our defect detection percentage was only 75 percent.) For another thing, thanks in part to our thorough testing of the hardware and of the software update process, there were no recalls and few returns of completely unusable appliances due to failed software updates.

3.7.3. Business Value of Testing Exercise

Consider the HELLOCARMS project toward the end of iteration two (as shown in the previous exercise). List the quantitative and qualitative values delivered by testing at this point.

If you are working in a classroom, break into groups of three to five. If you are working alone, you'll need to do this exercise yourself. If you are working in a classroom, once each group has finished, discuss the results.

I suggest allotting 15 minutes for this exercise, including the discussion.

3.7.4. Business Value of Testing Exercise Debrief

Let's look at the values of testing mentioned in the course to see which apply.

Quantifiable values

- Finding must-fix defects: This is clearly demonstrated by the charts shown in the previous exercise.
- Finding deferrable defects: We don't have an exact breakdown of the deferrable defects, but we can assume that some defects are being deferred given the rate of defect resolution.
- Reducing risk by running tests: Again, this is clearly demonstrated by the charts shown in the previous exercise. Given the large number of test failures, though, this value is less than one might like.
- Delivering information: In the previous exercise, we could make a strong case based on just four charts that the test results did not support deployment. This information, if properly delivered to and received by the project management team, gave Globobank the opportunity to avoid a very embarrassing and costly premature release.

Qualitative values

- Improving reputation for quality: Again, if the findings of the previous exercise influenced project management not to deploy at the end of iteration two, then we have certainly delivered value here.
- Smoothing releases: We certainly have tried to smooth the release process, and based on improvements from iteration two we continue to make this better. However, we need to perform further investigation as to the causes of high defect discovery rates and high test failure rates at the end of iteration two to increase the value we can deliver here.
- Increasing confidence: We have not examined the coverage metrics for HELLOCARMS testing, so we cannot say whether we are doing an adequate job here.
- Protecting from legal liability: Because testing does cover security and regulatory compliance characteristics—two key areas of potential legal exposure for banks—we are adding value here.
- Reducing loss of missions or lives: This is not a safety-critical application, but it is business critical. Banks that cannot issue home equity loans, lines of credit, and reverse mortgages are missing a major revenue stream.

You might have found other quantitative and qualitative values that apply to this project.

3.8. Distributed, Outsourced, and Insourced Testing

Learning objectives

(K2) List risks, commonalities, and differences between the three test staffing strategies (distributed, outsourced, and insourced)

Not too long ago, software and systems projects often tended to follow an in-house approach. A single team carried out most projects. The test team was part of the same company. They all were in the same building. Now, in many cases, an organization might spread the test effort across multiple teams. Perhaps only one or two of these teams are actually employed by the same company as the organization that is developing the software. The testing, development, and other work might occur at two, three, five, or more locations. What is happening?

Well, it's not just that one thing is happening, it's that one, two, or three different things could be happening. Let's look at the variants:

- If the test effort occurs at multiple locations, that test effort is distributed. For example, suppose your company has one test team in Bangalore and another in Prague, and those two test teams collaborate on the testing of a single project. Distribution introduces a number of new management issues, such as coordination of the work.
- If people who are not fellow employees of the rest of the project team, and who are not colocated with the project team, carry out the test effort at one or more locations, that test effort is outsourced. If more than one location is involved, it is distributed outsourced. For example, if you hire RBCS's test team in New Delhi to do all of your testing for you, you have outsourced your testing, but if you retain some of the testing in your team and outsource the rest to our team in New Delhi, you have distributed out-sourced testing happening. Outsourcing introduces a number of new management issues too, some of which are exacerbated by the distance (both in kilometers and time zones) between the various entities and the number of entities involved.
- If people who are colocated with the project team carry out the test effort, but those people are not fellow employees with the rest of the project team, that test effort is insourced. For example, if you hire some of RBCS's testers to come work as part of your test team for a project,

you have insourced some of your testing. Insourcing introduces a number of new management issues as well, such as ensuring that the people you bring into your team are bringing in the skills you need.

We can apply the same classifications for development and other project roles. The potential combinations of test, development, project management, release engineering, and other project roles across these three classifications can become quite overwhelming—and can provide a great fog cloud in which culpable parties can hide accountability if an organization is not careful.

You'll notice that I'm leaving aside the terms *offshore*, *nearshore*, and so forth, as I consider these mostly ways in which marketers try to differentiate their services. To the extent that these terms have any meaning, the meaning is relative—i.e., offshore from whom? nearshore to whom?—and thus not easy to deal with generally.

In this section, we'll look at how these practices affect testing, particularly test management. Some effects are common, and some are specific. Let's start with the common issues that apply to distributed, outsourced, and insourced testing.

We want to make sure that the entire project team is pointed in the right direction, so well-defined expectations are important. In all the cases we are considering here, we have two or more groups involved. If those groups think they are doing—or trying to do—something different, don't be surprised when partial or total failure ensues.

If everything always went perfectly on projects, we could spray chunks of work around the globe and gather up the finished work products at the end. However, in reality people need to discuss, coordinate, collaborate, learn, and change direction during projects, which means that clear channels of communication are required. The channels of communication cannot be informal, like the coffeemaker, cigarette-break, and company gym channels that exist in a company of fellow employees.

As a test manager, you'll need to carefully coordinate things that in an in-house project happen almost automatically within the project team with minimal effort. You must carefully plan for and manage test logistics related to hardware, software, tools licenses, information flows, test data access, and a number of other issues. You will need to manage the various location, time zone, cultural, and language differences that can exist. The location

and time zone differences are obvious, but don't underestimate two key things.

The first is the power of traveling to the remote location to solve problems. I have solved tricky testing coordination and collaboration issues in one-hour face-to-face meetings in cities around the world. These issues had bedeviled the project for weeks before I finally said, "I'm going over there", and got on an airplane.

The second is the compounding difficulty of managing multiple time zones. Yes, you can find a mutually convenient—or perhaps a mutually inconvenient—time for a phone conference, videoconference, live chat, Webinar, or other group information channel, at least when there are only two time zones. And, if the time is mutually convenient, you shouldn't have too much trouble as a manager getting people to attend this as a standing meeting. Try having a standing meeting at 7:00 a.m. once a week and see how your nonmorning people (such as me) respond. Now, just for grins, try to find a time for a phone conference in the following cities: Sydney, Australia; Austin, Texas; and Berlin, Germany. How about London, Tokyo, and San Diego? How about all six? And when does daylight saving time start and end in each time zone?

The cultural and language differences are a lot less obvious, which surprises people. Yes, culture in part refers to things like food, religion, holidays, taboos, and the like, which do exhibit astounding variability around the world and provide endless opportunities for you to do or say something completely inappropriate in a social or business situation. Reading a book on the cultures and histories involved before you start working on projects with a group of people can prove quite useful and help you deal with issues that could arise.

But culture also refers to business culture, which can vary tremendously within a single country. For example, generally in East Asia, business dealings tend to be polite and nonconfrontational. Business people tend to exhibit a strong bias for consensus. When presented with a problem, groups tend to work indirectly and privately toward a solution. Therefore, matters tend to move more slowly. However, I once sat in a meeting in Tokyo, delivering our assessment of a test situation to a client, and was simply astounded by the degree to which the client tore into my business associate, attacking him on every point and assertion. I have done a lot of business in Israel, where businesspeople have a reputation for very direct and frank discussions in meetings, and had not seen anything like this

incident among the more voluble and blunt Israelis with whom I dealt.

As for things tending to move slowly due to the consensus building and indirect problem solving, yes, that is generally true. However, one of the largest courseware development and delivery deals RBCS has ever been involved with was with a company in East Asia. That deal went from discussions to contract in a few short weeks. In parallel, a similar deal with a U.S. company took months to complete. So be careful with generalizations about cultures. In this globalized, diverse world, stereotypes can lead you astray.

Regarding the issue of languages, here are two simple truths to remember that have many subtle connotations and interactions. First, understanding a language is not a binary state. For example, I can read Spanish. I can write Spanish reasonably well if I have a dictionary handy. However, I find speaking and listening very difficult. So, if someone says to you, "Oh, yes, all our testers understand English", don't automatically assume there will be no language issues on the project.

Second, when understanding and expressing yourself in a language—any language, including the one you grew up speaking—all the ideas necessarily pass through a complex filter composed of your past experiences, your culture, your biases, your personal feelings, and so forth. The less of these things you share with the person with whom you're trying to communicate, the easier it is to miscommunicate.

Another issue is the alignment of methodologies across the groups. How are bugs reported? How are requirements gathered and documented? What types of reviews occur? For example, if you think of all the activities in the ISTQB fundamental test process, and all the tasks within each activity, two groups could carry out many of those tasks properly, but each group could carry out the tasks in slightly or even completely different ways. You have to first identify these methodological misalignments, ideally during test planning, not during test execution. You then must decide which ones will affect your testing and only then take steps to deal with those.

Similarly, be sure to manage the lifecycle implications. If one group is using an agile methodology and another using a sequential one, that's an obvious misalignment. However, two groups could follow the same lifecycle but simply not have thought through the issues associated with scaling that across multiple groups.

Finally, in case this extended discussion didn't already open your eyes to the fact, understand that you will have to deal with more and different project risks than in an in-house project. Be sure to plan for and manage the increased project risks.

3.8.1. Specific Distributed, Outsourcing, and Insourcing Testing Issues

What issues are specifically related to distributed test work? The most important one is the need to divide the test work across the multiple locations. You must do this division explicitly and in some detail. You cannot rely on people to "just figure this out" based on some vague statements about what types of testing go where. You must do this division intelligently. Be sure to consider the skills and qualifications of the various groups, and put the work in the right place.

During test planning and throughout the test effort, watch for signs of gaps and overlaps between the groups. Gaps are situations where every group and manager assumes some test condition is covered somewhere else but actually no one owns it. Gaps lead to false confidence during the test effort, potentially unpleasant surprises during test execution, and usually an increase in the residual quality risk on delivery.

Overlaps are situations in which two or more groups and their managers assume some test condition is covered by their group and their group alone but all of the groups actually cover it. Overlaps lead to inefficiency, potential rework during test execution, and confusion about the meaning of the test results if the results for similar tests disagree.

Finally, ensure a single, unified test dashboard. All of the test results should funnel into some single set of reports. Producing these reports should not be a Herculean task, either. Ideally, everyone is putting information into a central testing repository from which all the reports are automatically generated.

So, with distributed issues raised, let's look at the issues specifically related to outsourced and insourced test work. First, a little history to introduce some of the issues. In the 1980s and 1990s, the computer hardware business underwent a powerful trend toward outsourcing development and production of laptops, desktops, servers, hard drives, CD-ROMs, and other components to remote vendors, particularly in East Asia. In the late 1990s and 2000s, the software business began to follow the same path along the

same powerful trend, this time with a more global distribution of work that includes not only East Asia, but also South Asia, Southeast Asia, Africa, South America, East Europe, and Eurasia.

In both cases, hardware and then software, the main driver has been cost savings. A perception of a possible quality gap between low-cost outsource vendors and local developers drove many vendors to pursue certification according to ISO 9000 (in the case of hardware) or SPICE/CMM (in the case of software). However, such certification serves primarily as a marketing mechanism, to overcome potential objections to sales. Even if the vendor sincerely and completely implements Capability Maturity Model (CMM) or Software Process Improvement and Capability dEtermination (SPICE), that doesn't necessarily help you from a testing point of view, which I'll address further in a moment.

Similarly, vendors developed slogans like "right-shoring" and "follow the sun" to deal with two other possible objections to sales. "Right-shoring" addresses the perceived skills issues that can exist in rapidly growing, emerging high-tech economies, particular in terms of the ratio of senior to junior people. For example, I sometimes hear testers from emerging markets describe themselves as "senior" because they have two or three years experience on three or four projects. In most seasoned high-tech economies, "seniority" begins with at least 10 years of experience, and certainly more years than that in the case of management. "Follow the sun" addresses both the very real and problematic time zone differences that exist in many cases and the very real and problematic project risks that arise from trying to marshal and manage efforts that are happening thousands of kilometers and 5 or 10 time zones away.

I've been able to participate in this entire arc of outsourcing. I've worked in U.S.-based outsource test labs. I've helped clients use emerging-market outsource test labs. RBCS provides insource and outsource testing services, around the globe. This stuff works. It can save you a lot of money and you can manage the risks, but you do need to disabuse yourself of the marketing spin and sales slogans.

First, make sure that you are organized for success. This refers to the method-ology and lifecycle alignment issues I mentioned earlier as well as communication channels and logistics. But it also refers to contracts and understandings. If you expect to be able to scrutinize a development vendor's test results, you'd better make sure that's in the contract. Otherwise, you're in for a rude surprise when you ask for their test data,

test cases, test tools, bug reports, and test case tracking information.

Second, think carefully about whether you should use a separate test partner, and, if so, select the partner. Of course, since RBCS provides outsource test services, we believe that having an independent test partner help you with outsource projects is a very smart way to go. Otherwise, no matter how the development vendor structures their organization, you are going to have test independence issues, with all the self-editing and management editing of the test results that goes with them.

Third, whether for outsource or insource, ensure adequate personal interaction between your key players and their key players. It's harder for outsource testing, of course, because you can't see them as frequently. However, isolation of insourced testers happens remarkably often too.

As mentioned before, you'll need to adapt to the cultures. In the case of outsource and insourced, an additional cultural issue is company cultures. Notice I said "adapt" not "change". If you've hired the outsource or insource vendor, there's no reason for you to change to accommodate their culture, but don't be naïve and expect they will change their culture to accommodate yours either. Adaptation on both sides is the best outcome here.

Finally, make sure that you maintain focus on the need to get work done. As much as petty bickering can erupt in distributed work when everyone's part of the same company, it can get nasty when outsource or insource companies are involved. This is especially true if people see the outsource or insource company as a threat to their job. Don't get caught up in smokescreens or diversionary issues as a manager. Maintain focus on getting the test work done.

Make sure your outsource or insource vendor maintains focus too. One not-so-nice services company trick is to have a "senior consultant" or someone like that on a project, possibly on-site, who spends their time exploring ways to increase the size of the account. And, yes, that person is billing you for their time when they're doing that.

Watch for that trick and don't tolerate it. Yes, services companies should propose additional services when they see that you have a need. Having them do so helps you because they might be able to solve other problems, including problems that you didn't even know you had. However, if you're paying a test consultant to work on your project, that person should be

working on your project, not billing you while looking for new projects.

Now, it might sound mushy and softhearted to say, but distributed, outsourced, and insourced testing require trust. Distributed, outsourced, and insourced testing are all collaborative approaches to getting work done. Trust is critical because it's hard to collaborate with someone and gather evidence against them at the same time.

Given all the issues I've mentioned, including those related to organizational, cultural, language, and geographical boundaries, it's easy to be skeptical of these approaches. However, I mention the issues to help you manage them, not to argue against using this technique. If you manage distributed, outsourced, and insourced testing carefully, each of the test teams will carry out its roles properly.

At that point, the only things that continued skepticism and a lack of trust will cause are inefficiencies and delays. In low-trust situations, I have seen people spend an immense amount of time verifying that activities were happening, apportioning blame every time some minor problem arose, and playing organizational politics.

If you plan and manage the distributed outsource or insourced testing properly, you can be confident that the groups are doing the right things. In this situation, the presence of trust will help everyone maintain proper focus.

Trust must be earned and re-earned by both sides. Certainly I am not suggesting blind, naïve trust in vendors. That's a good way to get robbed. What I am saying is that, if you can put common expectations in place, with clear metrics for performance attached to them, you should be able to determine that everyone is doing their job properly. If someone is not, then you have clear set of numbers that you can use to address the performance problem and guide the way toward improvement.

3.8.2. CMM and Testing

I mentioned that I would return to this question of CMM and SPICE and how they relate to testing. In [Table 3-19](#), I have shown the 12 critical testing processes. In this table, I evaluated CMM against each of the 12 critical processes. The italic numbers indicate the CMM level that introduces some elements of the test process. Notice that I'm not saying that all elements are present, or that the elements introduced are done well, but rather that at least something must be done about this test process at this level.[\[5\]](#)

□ The twelve critical test processes are from my book Critical Testing Processes. I used my own test process framework, but the particular test process framework you use for this analysis wouldn't change the outcome much. Try it with TMM or TPI or STEP or T-MAP and you'll see what I mean.

Table 3-19. CMM and testing

Critical Testing Process	CMM	Critical Testing Process	CMM
Testing (plan, prepare, perform, perfect)	2345	Test system design and implementation	3
Context-tailored testing	3	Test releases	23
Quality risk analysis	34	Test execution	3
Test estimation	234	Bug reporting	345
Test planning	23	Test results reporting	34
Test team staff, skills	35	Change management	2345

The bold numbers indicate the CMM level at which you can assume proper handling of the test process. In other words, all the critical elements are there and—assuming someone is actually practicing the CMM level at which they are accredited—you can assume they are being done well.

SPICE, being very similar in terms of approach and philosophy to CMM, would probably show the same results in an analysis like this. Some people have asserted that Capability Maturity Model Integration (CMMi) changes this picture. I don't believe that's the case. The reason that CMM, CMMi, and SPICE are lacking in the area of testing is that, philosophically, they don't believe in testing as the main path toward better quality. We'll examine this issue more closely in [chapter 8](#).

So, the important thing to remember is that, while CMM, CMMi, and SPICE can help companies establish a foundation for quality, they do not guarantee thorough testing. In fact, as you'll see in [chapter 8](#), a company that claims CMM accreditation should be examined all the more carefully in terms of its testing.

3.8.3. Case Studies of Distributed Testing

Let's close this section with two examples of distributed, outsourced testing. For the first example, one of successful distributed testing, consider the

Internet appliance project I've used as one of the running case studies in this course.

In that project, the vendor's test team did most of the hardware testing. We had a few of the most critical tests repeated by a third-party hardware test lab. I don't recall any situations where the vendor's testing proved insufficient. During subsequent software testing, with their production hardware integrated into our overall system, we did not encounter an unusual number of hardware problems in the production hardware units. So we had a completely outsourced and doubly distributed hardware testing effort. We can say that distributing testing to this vendor's test team worked.

For the second example, one of unsuccessful distributed testing, consider again the Internet appliance project. My client bought a mail server component from a vendor, much as they bought hardware from a vendor. And again, the vendor tested their component before delivery—at least they said they had. However, when we tried to integrate their software component into our system, unlike our hardware experience, matters did not go well. There were many serious mail-related problems discovered, including in the areas of functionality, performance, and reliability.

So, here we have a situation where distributing testing to this vendor's test team failed. Same project, same people in charge (namely, me and my team), so why did that happen?

One key difference is that we were able to engage with and manage the hardware testing. I spent time in Taiwan, working with the hardware test manager directly. He was very honest and forthright with me. Conversely, the mail server vendor hid their test results from us, refused to deliver a test tool they had promised, and, once forced to deliver the tool, insisted on charging for the use of one of their junior testers as a "consultant" (at about \$1,500 per day).

You could say that my client got lucky in their choice of hardware vendor and unlucky in their choice of software vendor. However, I'd say the underlying difference was not luck. My client contact chose the hardware vendor, I knew the hardware vendor and their test team, and everyone knew that we would want to be involved in their testing. The hardware vendor had nothing to hide. However, another manager chose the software vendor. I had not worked with this manager or the software vendor before. I don't believe the manager set the vendor's expectations properly in

advance. So, as with many matters of testing success and failure, it comes back to properly aligned, realistic expectations.

3.9. Test Management Issues

Learning objectives

(K2) Compare management issues for exploratory testing, testing systems of systems, and testing safety-critical systems, including topics of strategy, benefits and disadvantages, adequacy, and their impact on planning, coverage, monitoring, and control.

We're going to conclude this chapter on test management with a discussion of some test management issues that you, as an Advanced test manager, might have to manage. These issues are in four areas:

- Managing reactive test strategies and experience-based test techniques
- Managing system of systems testing
- Managing safety-critical systems testing
- Managing nonfunctional testing

We'll start with this issue of managing reactive test strategies and experience-based test techniques.

For years, three of the chief complaints about reactive test strategies, such as bug hunting, software attacks, and exploratory testing, have been that they are difficult to manage, don't document what was tested, and don't tend to scale to large teams of testers very easily. While some proponents of exploratory testing have reacted defensively to these criticisms, some of them have attempted to respond in a productive fashion by devising techniques for managing them.

Session-based test management is a solution to two of these problems. It was designed for use in managing exploratory testing, but you can apply it to any of the experienced-based techniques and to all reactive test strategies.

In session-based test management, you break the test execution effort into test sessions. A test session is the basic unit of testing work. It is generally limited in time, typically between 30 and 120 minutes in length. The period assigned to the test session is called the *time box*. If you remember our discussion of earned value earlier, I said that during test execution, we can

treat the test case as the unit of testing work. The concept here is the same.

A test session should be a period of uninterrupted test execution for the tester. Test sessions should focus on a specific test object. They should focus on a specific test objective, the one documented in the test charter. You can think of a test charter as a very low-detail test case, perhaps only one to three sentences.

Out of this session comes a report, which captures the details of what the tester tested. This report can have the same level of detail as a written test script, but it's created after test execution. This approach is in contrast to analytical test strategies, in which test design and implementation occur before test execution and which produce most of their documentation in the test design and implementation periods.

Recall the test case summary worksheet shown earlier in [figure 3-39](#); it lists test cases along with planned hours. Depending on the level of detail provided in each test case, you could easily use that worksheet to manage and track sessions. The difference, as I noted before, is that with an analytical approach, testers would have created most of the information during test design and implementation, but in reactive approaches testers capture that information in the session report during test execution.

Now, session-based test management provides a manageable process for testing, which we'll discuss further in a moment. We also now have a way of gathering and reporting test results—which really we always had, because using test case tracking worksheets like mine has been a common testing practice since at least the 1990s.^[5] However, the approach still doesn't scale well. I'll come back to that problem in a moment.

[5] I discussed them in the first edition of my book *Managing the Testing Process*, which came out in 1999, but even then using such spreadsheets was common practice.

First, though, what is the process? What should a tester do during the test sessions? The proponents of this technique describe three main stages:

1. **Session Setup.** This involves both setting up the test environment and making sure the tester understands the area of the product or system to be tested. To some extent, this stage corresponds to creating the preconditions that occur when running scripted testing under a preventive test strategy. However, it also includes some test

environment setup tasks that, in a preventive test strategy like analytical risk-based testing, would occur before test execution and off the critical path for release.

2. **Test Design and Execution.** This consists of running the test against the test object and looking for any problems. If you are using exploratory testing, this stage is the "simultaneously learn the product, run the tests, and decide what to test next" activity that exploratory test proponents talk about. However, you could also be using Whittaker's software attacks approach under this method.^[5] Notice that the design activity is included here. Again, in a preventive test strategy like analytical risk-based testing, test design would occur before test execution and be off the critical path for release.

[5] Whittaker describes these attack techniques in his books *How to Break Software* and *How to Break Software Security*. I cover these topics in the companion volume to this book on *Advanced Test Analysis*.

3. **Defect Investigation and Reporting.** This same process would be triggered when a scripted test finds a problem. However, since we don't have a detailed written test case, it would not make sense to refer the developer to it. So if anything, testers must use an even more meticulous and detailed defect reporting process with reactive test strategies than the defect reporting process under preventive test strategies; preventive test strategies typically have produced test scripts and test data to reference.

While I have described these activities in a way that might sound sequential, in most cases these activities would overlap. In addition, depending on results of the debrief session (which we'll discuss in a minute), we might iterate this process for a given test charter.

Now, while defect reporting is much the same under a reactive test strategy—if somewhat more detailed to compensate for the lack of documented test cases and easily supplied test data—the results reporting is different, at least if session-based test management is being used. In a preventive strategy that uses scripted tests, the tester makes some notes about any deviations or extensions to the test script; captures the status of the test, the environment used, and other such data required for the test case summary worksheet; and moves on to the next test.

However, with session-based test management, since a written test script does not exist, the tester must capture information about what was tested to backfill that absence of documentation. These are called session reports or session sheets. What goes in these session reports? The content can vary considerably, depending on the documentation needs for the project. However, the advocates of session-based test management propose the following areas:

- Session charter. As I mentioned before, this is a one- to three-sentence description of what is to be tested.
- Tester name or names.
- Date and time started, and presumably completed.
- Task breakdown. What did we do during testing? Notice that, if we need a very detailed description of the testing, we'll find ourselves in a situation in which we have effectively placed creating large amounts of test scripts on the critical path for release, unlike with preventive test strategies that place test design and implementation off the critical path, parallel with system design and implementation.
- Metrics, such as the amount of time spent on setup, testing, and defect reporting (which should total the time allocated in the time box).
- Data files. This should include both data files used to perform the tests and data files produced by the tests. On some projects, you might be able to create sufficient data on the fly as part of a 30- to 120-minute test session, but on many projects this will not suffice. Again, preventive test strategies would tend to take this work off the critical path, as part of test implementation performed before test execution.
- Test notes. This would be freeform information captured about the test session.
- Issues. Issues include concerns that arose, such as missing hardware that prevented complete execution of the charter. However, it could also refer to additional testing opportunities discovered during the session, which might result in creation of new charters.
- Defects. This would probably list the defect IDs from the defect tracking system, but it might also include the summary line from each defect report.

Each test session concludes with a one-on-one debrief between the tester and the test manager. These debriefs would occur as needed throughout test execution.

In the debrief, the test manager reviews the session report, asking clarifying questions and discussing findings with the tester. Based on these discussions, the tester and test manager might improve the underlying test charter or some other test charter, triggering a new session. The test manager can adjust the plan for the remainder of the test effort based on the tester's feedback, though presumably most of these adjustments would be minor. The test manager would also estimate and plan further sessions, trying to manage the time remaining in the test execution period.

Now, these session agendas can vary. Some proponents of session-based test management use an acronym, PROOF:

- Past: What happened during the session? What did you do, and what did you see?
- Results: What was achieved during the session? What bugs did you find? What worked?
- Outlook: What still needs to be done? What subsequent test sessions should we try?
- Obstacles: What got in the way of good testing? What couldn't you do that you wanted to?
- Feelings: How do you feel about this test session?

An agenda I've used during test debrief meetings is simpler:

- What worked?
- What didn't work?
- What bugs did you find (which usually leads to a review of the bug reports)?
- What happened that was interesting?

I mentioned earlier that session-based test management, while addressing some of the process and documentation issues with reactive test strategies, is only a partial solution. One of the problems it doesn't solve is scaling.

By scalability of a test management technique, I mean that the technique is truly robust in many situations and generally applicable to most projects. This implies the key attributes discussed in the following paragraphs, among others, which are lacking in session-based test management.

To be truly robust and general, a test management technique should be able to handle test teams with anywhere from 1 to 100 people. On-demand test debriefs between tester and test manager limit the scalability here. I have resolved this by having end-of-day or beginning-of-day group debrief sessions.

To be truly robust and general, a test management technique should handle test teams working multiple shifts, possibly in multiple locations and time zones. Again, on-demand test debriefs between tester and test manager limit the scalability here. I have found that the end-of-day or beginning-of-day group debrief sessions provide a way to handle distributed times if the time of the debrief session is chosen properly.

To be truly robust and general, a test management technique should accommodate language diversity among the test teams. The on-demand debriefs would be hard enough to handle over the phone with a monoglot team, but imagine if there are language differences. I have found that I can manage polyglot test team debrief sessions by teleconference or Webinar through increased documentation, including providing sufficiently detailed test charters or test cases or test scripts—whatever we care to call them—to ensure clear understanding across the team prior to starting the test sessions.

To be truly robust and general, a test management technique should gainfully employ teams of contractor, outsource, and insource testers. Obviously, all experience-based test techniques inherently rely on the experience and skill of the tester. The available pool of test contractors does not always include skilled testers—at least not at a price that the project can afford. If the right test consultancy is chosen, you might be in good shape here. However, if you have to make do with the contractors you can get, the testers who work at your outsource vendor's organization, or the on-site test team that won the business based on the lowest bid, this technique is going to leave you in the lurch. But by adding more detail to the test charters—transforming them into test scripts, albeit only at the level of detail required—and preparing them in advance for use by the resources available, you can resolve this problem.

Finally, to be truly robust and general, a test management technique should produce effort and duration estimates well before test execution, ideally during project planning. The proponents claim to have solved this problem: Just figure out how many sessions you need, estimate the time box for each session, do a little math, and voila, there's your estimate. Notice, though, that this is a circular argument. How do you know that you actually will be done finding bugs when you get to the end of the overall time box? You don't. How do you know that you actually will have covered all the test conditions you need to cover when you get to the end of the overall time box? You don't. That said, you can apply the mature test estimation techniques I discussed earlier—techniques based on project management best practices—to ensure that the right number of sessions—along with the proper number of scripted tests—are used. So, while session-based test management as initially proposed by its proponents does not scale well, it is possible to adjust it as discussed earlier to make it do so.

3.9.1. A Case Study in Managing Exploratory Testing

In the Internet appliance project, we mixed our risk-based analytical test strategy with a reactive test strategy. The reactive strategy involved the use of chartered, exploratory test techniques during test execution. The analytical strategy involved the prior design and implementation of test scripts based on a risk analysis performed in parallel with test planning and project planning. During test execution, the test manager and the three test engineers, who together had over 20 years total experience, did the exploratory testing. Test technicians ran the scripted tests. Some of the test technicians had no testing experience and others had just a little.

[Table 3-20](#) shows the results. During test execution, the technicians each spent about 6 hours per day running test scripts. The rest of the time, 3 to 4 hours per day, was spent reading email, attending meetings, updating bug reports, doing confirmation testing, and the like. The engineers and manager, being heavily engaged in other tasks, could only spend 1 to 2 hours per day doing exploratory testing. Due to their heavy experience, you can see that, even so, the experienced testers were the primary bug finders.

Table 3-20. Managing exploratory testing

Staff	7 Technicians	3 Engineers + 1 Manager
Experience	< 10 years total	> 20 years total
Test Documentation	Precise scripts	Exploratory charters

Test Hours per Day	42	6
Bugs Found	928 (78%)	261 (22%)
Bug Effectiveness	22	44
Scripts Run	850	0
Inputs Submitted	~5, 000-10, 000	~ 1, 000
Outputs Verified	~4, 000-8, 000	~ 1, 000

However, if you examine test coverage, we see a different situation. The technicians ran roughly 850 test scripts over the three months of system test. That covered a lot of ground, well-documented ground that yielded well-documented results that we could show to management. The exploratory testing produced very little clear documentation. We weren't using the session reports that I showed earlier, in part because we were relying on the technicians to gather the coverage evidence with the scripts.

Now, not only did scripted testing cover more ground, it also involved more hours of testing and megabytes of test data. We didn't measure this, but I estimate that the manual scripted tests resulted in somewhere between 5 and 10 thousand inputs of various kinds—strings, dates, radio buttons, etc.—while the exploratory testing was probably at most a fifth of that volume. Similarly, scripted tests involved more explicit checking of individual results. On a per-hour basis, the exploratory testing might have been equally effective, but it would have been less effective if we would have had to produce the session reports because that would have slowed us down.

So, which was better? Ah, it wasn't that kind of experiment. It wasn't an experiment at all; it was a proven way of mixing two strategies, each with different strengths. The exploratory testing was effective at finding bugs on an hour-per-hour basis, and we found a number of bugs that wouldn't have been found by the scripts. The reusable test scripts gave us good regression risk mitigation, good overall risk mitigation, and good confidence building. In sum, a successful blended approach.

In fact, the approach was more blended than it first appears. In [Table 3-20](#), I'm simplifying by creating a sharp division between scripted and exploratory testing. The reality is that the difference is entirely one of degree, not of kind. The more verbose your test charter, the more scripted your testing. The less detailed your test script, the more exploratory your testing.

On this project, as on many projects, the test technicians were given latitude to vary their execution of the test scripts. We told them, "A test script is a road map to interesting places; when you get somewhere interesting, stop and look around." We included some time in our planned test time—which we could easily have called a "test session"—for them to explore the system. In addition, as the test technicians became more experienced, we had them start writing test scripts themselves and doing some of the exploratory testing.

Some advocates of exploratory testing insist that exploratory testing is nothing like scripted testing. That's a good marketing position because you have to differentiate your services. However, the reality is not so black-and-white. I have seen advocates of exploratory testing produce test charters that were much more detailed than many test cases that my associates and I have written for scripted testing. The reality is that the level of detail in the test cases, as well as the point in time at which they are created, is a complicated issue that you'll have to struggle with on your projects as a test manager.^[1]

[1] I cover the issue of the level of detail in test cases in [chapter 3](#) of *Managing the Testing Process*.

As I mentioned in the earlier discussion about strategies, all strategies are a means to an end. You should select, adapt, and blend the right strategy for your situation. Session-based test management can help you blend reactive test strategies with preventive ones in a smoother fashion.

3.9.2. System of Systems Issues

Let's proceed to the issue of managing system of systems testing. A major test management issue with such systems is the complexity of distributed system testing. Generally, multiple different levels of testing will occur at different locations and different levels of formality performed by different groups. So, you have all of the distributed testing issues discussed earlier, usually compounded by outsourced issues because multiple companies are generally involved.

Ideally, test management will involve a master test plan that spans these various levels. This master test plan is much easier to write and to manage if the entire project team follows a single, formal lifecycle model.

From my own experience with system of systems projects, I can say that such project teams take a terrible risk if they rely on high-level testing

(such as system integration test) as their sole quality mechanism. It would be nice if the work products from 5 or 10 disparate projects simply converged into a nice, working whole at the end of months or even years of work, but that's not the way these things go.

So, a formal quality assurance process and plan that includes verification and validation throughout the lifecycle, including static and dynamic testing and formalization of all test levels, is very useful. Of course, that now means that the authors of the master test plan and all the level test plans must align those documents not only with each other but also with the quality assurance plan.

Because there is so much inherent complexity associated with testing system of systems projects, I prefer to see as much of that complexity resolved before test execution starts. Formal configuration management, change management, and release management plans and processes help. These plans and processes must define agreed-upon touchpoints and hand-offs between testing and the rest of the project in these areas. This level of formality is the only way to ensure that test objects come from controlled, known code bases; that requirements, design, and code changes occur in a managed (and diminishing) way during the end of the project; and that testing is occurring against known, versioned test items.

Systems of systems are inherently complex, and so their test environments are as well. I have worked on test projects where planning and configuring the test environments alone required almost 10 person-years of effort, making it larger than many projects. The effort required was high-skill, high-complexity work too, meaning that mistakes resulted in painful false positives and project delays.

Not only that, but systems of systems tend to process complex data. In order for our test results to be meaningful, we need equally complex test data. That also can be a mini-project in its own right. I've managed testing efforts where we had to create literally millions of test records across dozens of tables that had complex key relationships based on name, address, and identification number.

Some of RBCS's clients in the banking world have dozens of systems that share and manage related data. They want to use production data for testing but face serious privacy issues with doing so. Anonymization of those data sets can be a huge effort.

A number of times I've mentioned the IVR project, which was a complex system of systems project. You might recall my discussion about the "backbone integration" testing. In that case, we were doing integration testing of systems that would go into the system of systems before those constituent systems existed. So we had to use a variety of stubs, drivers, harnesses, and simulators. Some of these simulators were complex and required significant effort to build.

One of our clients is the United Space Alliance, that, among other things, creates the simulators used to train shuttle astronauts and perform various preflight tests. This effort requires an entire team, and even the simulators require extensive testing.

Finally, for most systems of systems, there are complex dependencies between the systems. In many cases, certain parts of one system must be completed before a test of another system can be run. On one project, as I mentioned before, I had to create a database to manage these dependencies.^[1] That's not to mention the need for complete system integration testing across all these dependencies, touchpoints, shared data repositories, and other interfaces.

[1] This database is described in [chapter 6](#) of *Managing the Testing Process*.

3.9.3. Safety-Critical System Issues

Let's now address the issue of managing safety-critical systems testing. For many safety critical systems, there are industry-specific standards that apply to testing and quality assurance. For automotive systems, there is the voluntary Motor Industry Software Reliability Association (MISRA) standard. For medical systems, government entities like the United States Food and Drug Administration apply strict mandatory standards. Military systems are generally subject to various standards. However, even systems that you might not think of as typically safety critical, like laptop computers, DVD recorders, and the like, are subject to mandatory and voluntary standards. If you have a laptop, turn it over right now and take a look at all the logos on the bottom of it. Failure to comply with these standards can mean exclusion from one or all marketplaces or clients, whether by law or by virtue of some purchasing checklist.

A safety-critical system is safety critical because it might directly or indirectly kill or injure someone if it fails. This risk creates legal liability as

well as moral responsibility on the vendor or vendors creating the system. While not all risks of failure can be reduced to zero likelihood, we can use formal, rigorous techniques, including the following:

- Bidirectional requirements, design, and risks traceability through to code and to tests
- Minimum test coverage levels, including perhaps the multidimensional coverage across various ways of measure coverage I mentioned earlier
- Quality-focused acceptance criteria, including ones focused on non-functional quality characteristics
- Standardized, detailed, and mandatory test documentation, including of results

This rigor reduces the risk of accidental oversights, omissions, and mistakes and provides evidence of compliance. People can use that evidence to support various audits, both preventive audits such as those done by the Food and Drug Administration prior to approval of a medical device and reactive audits such as those done by the Federal Aviation Administration after a plane crash.

As with systems of systems, and for similar reasons, we tend to see formalized development lifecycles defined for safety-critical systems. Historically, these have tended to be sequential lifecycles, but we have a client in the medical device business that uses a variant of the Scrum agile methodology, adapted for safety-critical devices.

Finally, as alluded to earlier, there is often a need to test nonfunctional quality characteristics with safety-critical systems. After all, the right answer given too late, or the right answer shared with the wrong people, can be just as dangerous as the wrong answer. So you should consider testing of reliability, availability, maintainability, safety, and security, along with performance, during test planning. Of course, if the safety-critical system is also a system of systems, the issues mentioned previously apply.

3.10. Nonfunctional Testing Issues

Let's conclude with the issue of managing nonfunctional testing. It's a common mistake for organizations to unintentionally or intentionally omit nonfunctional tests, but that approach is very dangerous. As I said a moment ago, the right answer given too late, or the right answer shared with the wrong people, can be just as dangerous as the wrong answer. Why do people omit nonfunctional tests?

Well, if the omission is unintentional, it's typically a matter of unawareness. The organization has been lucky. They've never had a performance, reliability, or security problem before, and they've never heard or read about a company that has. So, ignorance is bliss—at least, until the unmanaged risk becomes an undesirable event, perhaps a very undesirable event, perhaps a business-curtailed undesirable event.

Intentional omission of nonfunctional tests is often due to the high cost associated with many nonfunctional tests. Performance and reliability testing requires complex, production-like environments. Security testing requires expensive penetration tests. Usability testing involves usability test labs and user interface prototypes.

It's also true that nonfunctional tests can confuse the uninitiated. Different types of nonfunctional tests apply to different systems. Not all systems have security risks. Not all systems have usability risks. Not all systems have reliability risks. Does your system? Why or why not? How serious is the risk? Does the level of the nonfunctional risk justify the cost and complexity of the test? These are complicated questions with complicated answers, unlike the more straightforward answers to the same questions about functionality problems.

To properly plan, design, implement, and execute nonfunctional tests, we need to consider a number of important factors:

- Stakeholder requirements
- Tool needs
- Test hardware and environment requirements
- Organizational considerations

- Communications
- Data security

Let's examine each of these factors, starting with stakeholder requirements. One major challenge to nonfunctional testing arises from the typical state of nonfunctional requirements. Nonfunctional requirements are often poorly specified or missing completely.

The poor test analyst and test manager must embark on a process of discovery, eliciting stakeholder expectations rather than receiving a nice, neat, approved nonfunctional requirements specification. Further, this job won't wait until late in the project. Late discovery of nonfunctional problems generally indicates serious requirements and design problems, problems that are difficult to resolve with simple bug fixes and workarounds. Based on the requirements, the test analyst and test manager must work with the project stakeholders to identify and assess the nonfunctional risks. As was discussed earlier, this quality risk analysis requires multiple viewpoints; in this case, the discovery of nonfunctional requirements requires multiple viewpoints too.

Another major problem arises from untestable nonfunctional requirements. You can improve the testability of nonfunctional requirements by striving for the following:

- Specify requirements with testing in mind. Ask yourself, "Can I devise a test case that could clearly show that this requirement wasn't met?" This is not only a practical question. It also means that you must use clear, simple, consistent, and concise wording. Try to use words that have a clearly defined meaning on your project, such as those defined in a project or standard glossary. If you use words like *shall*, *should*, *must*, and *may*, be sure to understand what those mean in the context of requirements.
- Know your reader. Remember that written communication must be written *for the audience*. On distributed projects, think about the level of language mastery associated with the readers.
- Avoid ambiguous requirements. Reviews are helpful in spotting ambiguous wording.
- Use a standard format for requirements. This includes not only a template or tool used to capture them, but also the level of detail.

Remember, in the discussion about traceability, I said that we need to have the requirements and the test cases fine-grained enough that we can draw some conclusions about what works and what doesn't work based on our test results.

- Specify requirements quantitatively where possible and appropriate. For reliability and performance, this is both essential and straightforward—at least if you can get the stakeholders to give you realistic targets! For nonfunctional attributes like usability, this can be challenging.^[1]

^[1] I discuss ways to achieve this in the companion volume on advanced test analysis.

For nonfunctional attributes like maintainability, the wrong metric might actually be misleading or dangerous. For example, what is the proper number of lines of code in an existing code base of 100, 000,000 lines that should be allowed to remain in the code base in spite of one or more company coding standard violations? The correct answer is not zero. In fact, this isn't even the right question! The right question is, "What process can we use to improve maintainability of our code by reducing coding standards violations in those areas of the code that are most frequently maintained?"

3.10.1. Tools and Hardware Requirements

In the areas of tools, we find a mixture of good and bad news for us as test managers. It is good news that commercial tools and simulators are often available for nonfunctional testing. A good half-dozen tools, both commercial and open source, support load, performance, and reliability testing of Web-based applications. Security tools for both static and dynamic testing exist. Dynamic analysis tools for resource utilization are available.

However, it is often the case that these tools are not optional aids to your efficiency because they are with functional test automation tools. Indeed, many forms of nonfunctional testing, like performance, load, and reliability testing, simply cannot be done in any meaningful sense without the use of tools.

So, as a test manager, be sure that your test estimate includes tools. In addition, understand that many of the nonfunctional test tools—since they are addressing a complicated testing issue—are complicated. You can't simply give them to your functional testers and say, "Here, folks, go run a performance test with this", or, "Hey, Jane, I know you're not a

programmer, but please go and identify all coding standards violations in our existing code base with this static analysis tool."

You need knowledgeable experts to use most nonfunctional testing tools. This means that either you build that expertise in-house over time—which raises questions about the learning curve you can afford on your project—or you bring in outside specialists, testing consultants with expertise in the specific test type and test tool.

While there are commercial tools available for much nonfunctional testing, sometimes you have to build your own tool. As a test manager, you'll need to keep in mind that building a complex tool or simulator is a development project. Are you capable of managing a complex software or systems development project, or do you need to bring in a project manager or development manager to help with that? Having been personally involved in about a dozen complex test tool development projects, I can assure you it is challenging. At least you'll gain some sympathy for your development-manager peers if you've gone through it a few times

Whether they're purchased, open source, or homebuilt, make sure you are allowed to use the tools that you use for safety-critical systems. Certain regulatory standards require approval of all tools used to produce the regulated system, and that can include test tools.

Nonfunctional testing hardware requirements are often a huge challenge of nonfunctional testing. If any testing topic is more frequently associated with crestfallen project managers—and test managers—I'm not sure I've encountered it.

The root of the problem is that software and systems are not physical. Therefore, unlike with objects in the real world, it's not possible to extrapolate test results from scaled-down models. Our colleagues in civil, mechanical, and aeronautical engineering can rely on well-understood materials properties and the laws of physics to allow them to build models and extrapolate from them. Software and systems are not physical, so they are not built from materials with defined properties, nor do physical laws control them. You can program a world where people fly and where bullets do not kill them—indeed, many computer-graphics animators and videogame programmers have.

If you want to measure performance, reliability, load response, and such within these virtual worlds that the human brain has created, you have to

test the entire virtual world, which includes the hardware. So, you need production-like test environments for realistic nonfunctional test results in many cases.

I can almost hear managers wincing as they read that. I have been there before, so I understand that this situation creates significant test planning and funding challenges. But what will you pick: Cheap, misleading, invalid nonfunctional test results? Expensive, accurate nonfunctional test results? Or no nonfunctional test results?

Of course, people don't tend to like that choice, so they try to minimize the expense and still obtain accurate test results. Sometimes, you can do this. You can identify narrow time windows for running the most expensive tests.

For example, you could rent (rather than buy) the virtual user licenses needed to perform a full load simulation with your load testing tool, trying to get as much testing done as quickly as possible. That can work, but make sure you have programmer and designer resources standing by to fix bugs quickly when you find them—not if, when. It is a classic worst practice of testing to plan, design, or execute a test based on the assumption that the test won't find any bugs.

Here's another example. You could decide to hire a usability test lab to do a usability study in a dedicated lab and conduct a user survey, but do that only once. That can work, but when will you do it? Early, before everything is ready? Or late, when finding a major design-related usability bug might trigger major rework and schedule slippage on the project? Again, don't make the mistake of planning for usability testing on the assumption nothing will go wrong.

Another way organizations try to do nonfunctional testing on the cheap is by using the production environment for testing. This method results from a belief—justified or not—that it would be too expensive to replicate production, so use of the production environment itself is the only practical option. Clearly, if this decision is made, you will have to plan for test execution at just the right time, often overnight or on weekends, with careful plans, and in such a way that you can restore the production environment to a completely normal state during normal business hours.

The classic cautionary tale of testing on a production system is the Chernobyl nuclear plant explosion.^[5] If you must run tests in production—perhaps because there is no replica of the system obtainable for any

reasonable price, as with a nuclear plant or a space shuttle—then you must exercise great care. Identifying as many risks as possible, and having contingency plans in place for those risks that will limit the impact to acceptable levels, is essential.

□ See Dietrich Doerner's *The Logic of Failure* for a discussion of how that explosion occurred.

While often a bad idea, testing in production happens very often. Anytime you and your colleagues choose not to perform nonfunctional tests due to the cost of the environments, understand that you have chosen to test in the production environment. The only question remaining is whether that test will be deliberate, designed, planned, and controlled.

For performance, load, and reliability testing, just as important as a production-like test environment is a production-like incoming level of load. Don't spend a bunch of time and money getting your production replica environment in place and then use the wrong incoming loads. Make sure that load generation is precise and realistic. Remember, too, that the amount of data in the various repositories can have a profound influence on performance, load, and reliability results. Mistakes in any of these areas—realism of test environment, realism of test loads, or realism of test data—will typically lead to invalid—and perhaps misleading—test results.

It's typical for test execution to modify test data or to leave one or more of the host servers or cohabiting applications in an unusable, strange, or unknown state. Rebooting a server to restore normal operations, reloading test data, and restarting applications are common sense for testers when confronted with such situations.

It's less common for tests to unhinge the communication infrastructure. However, that can happen. A router, switch, or firewall could be affected by a test.

In some cases, it's also necessary to modify the way a piece of communication infrastructure is working in order to run a test. For example, there are certain firewall benchmarking tests that are commonly run with atypical incoming traffic and network settings.

If communication protocols either need to be changed for a test or might change due to a test, you'll need to ensure that the right tools are available. If the test suddenly starts failing because of a minor, harmless change in the way packets are routed on a network, say, then you'll have a

massive false positive to resolve before you can resume the nonfunctional testing you were trying to performing. Because nonfunctional testing tends to be expensive, the last thing you need is false positives in your results. Finally, if you must make an infrastructure modification to run a test, be sure your testers know how to reliably and completely undo that modification.

3.10.2. Organizational and Security Considerations

As if this matter of test environments weren't complicated enough, we also have to consider organizational issues. If we are testing a complex system that consists of several components, those components might not be all colocated. They might, in fact, be distributed both geographically and organizationally. For example, when we were testing a home equity loan system, one back-end system that was used to process the loan applications was located off-site. Another back-end system that belonged to another organization, a credit bureau, was also located off-site. We had to carefully plan and coordinate our interface testing with the credit bureau, but it went smoothly if we followed the rules. However, for the loan-processing system, we had a great deal of difficulty convincing these people—who were employees of the same company—not to do things like apply patches, reboot the system, and start bandwidth-consuming backups and other test-disrupting events during test periods.

If there are availability issues for test environments or test support personnel on those environments, be sure to address those during planning. Forgetting to plan for just one essential back-end or down-stream system—as easy as it is to do—can have serious schedule impacts.

As a last issue for nonfunctional testing—which really applies to all types of testing—be sure to consider how security measures will intersect with the testing you are to do, and plan accordingly. For example, in the United States, the Payment Card Industry Data Security (PCI DSS) standard requires various levels of data encryption. This includes encryption in transit—e.g., using WPA encryption for wireless routers that transmit credit card data—as well as in repositories—e.g., encrypted fields in databases that store sensitive information. Obviously, if we are testing security, we need to check for compliance to this standard (if applicable). However, if we are trying to test functionality and to compare data values to expected results, encryption can make that difficult.

As I mentioned a moment ago, it's necessary for performance, load, and

reliability testing to test with realistic-sized data sets. It's also useful to have data that actually contains the same kinds of errors that exist in the real data sets. There's no better way to do that than to use production data. However, doing so might bring us afoul of various laws, standards, or company regulations that preclude exposing sensitive data to people without a strict need to know. This problem can erupt suddenly when the decision to outsource or insource testing occurs because different rules might apply to nonemployees, particularly if they reside in different countries.

In some cases, organizations try to respond to these issues by anonymizing test data. In other words, they create some tool or algorithm that allows them to scramble the data in their production data in some irreversible, secure way. There are two huge challenges with doing so. First, the scrambling of data must result in data that still looks like production data. For example, names should still look like names, not a sequence of random characters; addresses are potentially valid (though fictitious) addresses; the numbers of siblings, living parents, and children are all realistic; and so forth. Otherwise, it defeats the entire purpose. Second, the scrambling of data must scramble keys that are shared across tables in such a way that delicate, critical linkages between records in different tables are preserved. If not, once again, you will not have test data that looks like production data in some of the most important ways, which is in terms of the interrelationships among data sets.

In this area, remember, planning beforehand is better than surprises. If you detect security-related obstacles to your testing, escalate the issue to the appropriate level for resolution—or acceptance of test scope reduction. I have clients in the banking industry that accept the fact that there are many defects related to data complexity and data quality that they simply can't find because their company security officers will not let them test with production data.

3.11. Sample Exam Questions

To end each chapter, you can try one or more sample exam questions to reinforce your knowledge and understanding of the material and to prepare for the ISTQB Advanced Level Test Manager exam.

Q: Which of the following is a project risk mitigation step that you might take as a test manager?

- 1.** Testing for performance problems
- 2.** Hiring a contractor after a key test analyst quits
- 3.** Procuring extra test environments in case one fails during testing
- 4.** Performing a project retrospective using test results

Q: 2. You are planning the testing for an integrated system that will use three off-the-shelf components to manage a bank's accounts-receivable system. You are conducting an informal quality risk analysis session with project and system stakeholders to determine what test conditions should be tested and how much each test condition should be tested. Which of the following is a quality risk item that you might identify in this quality risk analysis session?

- 1.** Failure of a component vendor to conduct adequate component testing
- 2.** Calculation of excessive late-payment penalties for invoices
- 3.** On-time payment of all invoices for international vendors
- 4.** Calculation of risk priority using likelihood and impact

Q: During a formalized quality risk analysis session following the Failure Mode and Effect Analysis technique, you are calculating risk priorities. Which of the following are major factors in this calculation?

- 1.** Severity and priority
- 2.** Functionality, reliability, usability, efficiency, maintainability, and portability

3. Loss of a key contributor on the test team

4. Loss of a key contributor on the development team

Q: Assume you are a test manager in charge of integration testing, system testing, and acceptance testing for a bank. You are working on a project to upgrade an existing automated teller machine system to allow customers to obtain cash advances from supported credit cards. The system should allow cash advances from \$20 to \$500, inclusively, for all supported credit cards. The supported credit cards are American Express, Visa, Japan Credit Bank, Eurocard, and MasterCard.

Which of the following statements best associates a key stakeholder with the kind of input that stakeholder can provide during a quality risk analysis?

- 1.** A tester can provide input on the likelihood of a risk item.
- 2.** A developer can provide input on the impact of a risk item.
- 3.** A business analyst can provide input on the likelihood of a risk item.
- 4.** A help desk staffer can provide input on the impact of a risk item.

Q: Which of the following is a situation in which you would expect an iterative quality risk analysis to result in the largest number of new or changed risk items and risk levels?

- 1.** You perform a risk analysis on the final requirements specification and subsequently receive a draft design specification.
- 2.** A tester leaves after test design is complete, and you hire a new tester to replace her.
- 3.** The development manager hires two additional programmers after the quality risk analysis is complete.
- 4.** You perform a risk analysis on the final requirements specification and then that document is placed under formal configuration management.

Q: Assume you are a test manager working on a project to create a

programmable thermostat for home use to control central heating, ventilation, and air conditioning (HVAC) systems. In addition to the normal HVAC control functions, the thermostat also has the ability to download data to a browser-based application that runs on PCs for further analysis.

During quality risk analysis, you identify compatibility problems between the browser-based application and the different PC configurations that can host that application as a quality risk item with a high level of likelihood. You plan to perform compatibility testing to address this risk.

Which of the following is a way in which you might monitor the effect of testing on the reduction of this risk during test execution?

1. Reduce the number of supported PC configurations.
2. Assign more testers to cover compatibility than testers to cover functionality.
3. Analyze the number of defects found that relate to this risk item.
4. Plan to test the most common PC configurations.

Q: Which of the following is an example of a project where failure mode and effect analysis would be a better choice for risk analysis?

1. It is the project team's first application of risk-based testing.
2. The system under test is both complex and safety critical.
3. The system under test is a financial system.
4. Minimizing the amount of documentation is a key concern.

Q: Assume you are a test manager in charge of integration testing, system testing, and acceptance testing for a bank. You are working on a project to upgrade an existing automated teller machine system to allow customers to obtain cash advances from supported credit cards. The system should allow cash advances from \$20 to \$500, inclusively, for all supported credit cards. The supported credit cards are American Express, Visa, Japan Credit Bank, Eurocard, and MasterCard.

In the master test plan, the Features to be Tested section lists the following:

- 1.** All supported credit cards
- 2.** Language localization
- 3.** Valid and invalid advances
- 4.** Usability
- 5.** Response time

Relying only on the information given above, select the features to be tested for which sufficient information is available to proceed with test design.

- 1.** I
- 2.** II
- 3.** III
- 4.** IV
- 5.** V

Q: Continue with the scenario described in the previous question. Which of the following topics would you need to address in detail in the master test plan?

- 1.** A strategy for regression testing
- 2.** A list of advance amount boundary values
- 3.** A description of intercase dependencies
- 4.** A logical collection of test cases

Q: Assume you are a test manager working on a project to create a programmable thermostat for home use to control central heating, ventilation, and air conditioning (HVAC) systems. In addition to the normal HVAC control functions, the thermostat also has the ability to download data to a browser-based application that runs on PCs for further analysis.

The company test strategy calls for each test case to be run on all combinations of configuration options. For this system, you identify the following factors and, for each factor, the following options:

- Supported PC/thermostat connections: USB and Bluetooth
- Supported operating systems: Windows 2000, Windows XP, Windows Vista, Mac X, Linux
- Supported browsers: Internet Explorer, Firefox, Opera

Because there are 10 test cases that involve downloading data, this would require running 300 test cases, each of which requires an hour to run.

With management approval, you decide to test five configurations, covering each option but not all the possible pairs and triples of options.

Which of the following statements describes the best option for documenting this deviation from the test strategy?

1. In the test design specifications, explain the alternate approach planned for this project and how to set up the test configurations.
2. In the test procedure specifications, explain which test cases should be run against which configurations.
3. In the master test plan, explain the alternate approach planned for this project and why this approach is sufficient.
4. In the test item transmittal report, explain the alternate approach planned for this project and which test items were tested against which configuration.

Q: Continue with the scenario described in the previous question. You are writing a master test plan to cover integration testing and system testing of the programmable thermostat. Select all of the following statements that are true.

1. The approach section should describe how to test the integration of the thermostat with other parts of the HVAC system.
2. The schedule section should describe when integration testing

should start and when system testing should start.

3. The environmental needs section should address who is responsible for each level of testing.
4. The test items section should describe the equipment required for each level of testing.
5. The test deliverables section should describe results reporting for each level of testing.

Q: 12. Continue with the scenario described in the previous questions, where you are a test manager working on a project to create a programmable thermostat for home use to control central heating, ventilation, and air conditioning (HVAC) systems. One critical quality risk identified for this system is the possibility of damage to the HVAC system caused by excessive cycling of the compressor (i.e., turning the unit on and off repeatedly in short intervals). Which of the following is a reasonable way to use an IEEE 829 test plan to direct appropriate testing for this risk?

1. Write a separate test plan for this level of testing.
2. List the feature that prevents excessive cycling as a feature to be tested.
3. Detail all of the requirements of the programmable thermostat in the introduction of the test plan.
4. Include a fully functioning compressor as one of the test items.

Q: Continue with the scenario from the previous question.

Historically, on seven past projects, the test team has found approximately 12 bugs during system test for each person-month of development team effort. Five developers are assigned to work on a new project that is scheduled to last six months. Assume that the cumulative number of bugs found, as shown in your convergence chart, has flattened at 351 defects.

Based on this information only, which of the following statements is most likely to be true?

1. You would expect to find exactly 20 more defects before the end

of system test.

- 2.** You have omitted tests for at least one critical quality risk category.
- 3.** You needed a test team of at least three testers for optimum testing.
- 4.** You have found roughly the number of defects you would expect to find during system test.

Q: 14. Continue with the scenario from the previous question.

Assume that this project is following an iterative lifecycle, while the previous projects for which you have bug metrics followed a sequential lifecycle. Assuming no other dissimilarities between this project and the previous projects exist, which of the following might be a reason to question the accuracy of the predicted number of defects?

- 1.** People factors
- 2.** Material factors
- 3.** Process factors
- 4.** Quality factors

Q: You are a test manager in charge of system testing on a project to update a cruise-control module for a new model of a car. The goal of the cruise-control software update is to make the car more fuel efficient.

You have written a first release of the system test plan based on the final requirements specification. You receive an early draft of the design specification. Identify all of the following statements that are true.

- 1.** Do not update the system test plan until the final version of the design specification is available.
- 2.** Produce a draft update of the system test plan based on this version of the design specification.

3. Check this version of the design specification for inconsistencies with the requirements specification.
4. Participate in the final review of the design specification but not any preliminary reviews of the design specification.
5. Review the quality risk analysis to see if the design specification has identified additional risk items.

Q: 16. Which of the following is the best example of a technique for controlling test progress in terms of the residual level of quality risk?

1. Counting the number of defects found and the number of defects resolved
2. Counting the number of test cases passed and the number of test cases failed
3. Counting the number of requirements that work properly and the number of requirements with known defects
4. Counting the number of tested risk items without known defects and the number of tested risk items with known defects

Q: You are a test manager in charge of system testing on a project to update a cruise-control module for a new model of a car. The goal of the cruise-control software update is to make the car more fuel efficient.

Halfway through test execution, you find that the test results do not conclusively determine whether fuel efficiency has improved. Identify all of the following actions that you might direct the test analysts to take to help to resolve this problem.

1. Redesign the fuel efficiency tests.
2. Revise the quality risk analysis.
3. Modify the test environment to gather more detailed actual results.
4. Check for consistency in tested fuel mixtures.
5. Report fuel efficiency as apparently unchanged.

Q: Assume you are a test manager in charge of integration testing, system testing, and acceptance testing for a bank. You are working on a project to upgrade an existing automated teller machine system to allow customers to obtain cash advances from supported credit cards. The system should allow cash advances from \$20 to \$500, inclusively, for all supported credit cards. The supported credit cards are American Express, Visa, Japan Credit Bank, Eurocard, and MasterCard.

During test execution, you find five defects, each reported by a different tester, that involve the same problem with cash advances, with the only difference between these reports being the credit card tested. Which of the following is an improvement to the test process that you might suggest?

1. Revise all cash advance test cases to test with only one credit card.
2. Review all reports filed subsequently and close any such duplicate defect reports before assignment to development.
3. Change the requirements to delete support for American Express cards.
4. Have testers check for similar problems with other cards and report their findings in defect reports.

Q: You are the manager of a bank's quality assessment group, in charge of independent testing for banking applications. You are working on a project to implement an integrated system that will use three off-the-shelf systems to manage a bank's accounts-receivable system. You are currently managing the execution of system integration testing.

Consider the following bug open/closed or convergence chart.

Which of the following interpretations of this chart provides a reason to not declare the system integration testing complete?

- 1.** The bug find rate has not leveled off.
- 2.** Developers aren't fixing bugs fast enough.
- 3.** The complete set of tests has not yet been run.
- 4.** A number of unresolved bugs remain in the backlog.

Q: Which of the following is an example of a cost of internal failure?

- 1.** Finding a bug during testing
- 2.** Training developers in secure coding practices
- 3.** Designing test cases
- 4.** Fixing a customer-detected bug

Q: You are the manager of a bank's quality assessment group, in charge of independent testing for banking applications. You are in charge of testing for a project to implement an integrated system that uses three off-the-shelf components to manage a bank's accounts-receivable system.

Which of the following is most likely to be a major business motivation for testing this system?

1. Avoiding loss of life
2. Having confidence in correct customer billing
3. Finding as many bugs as possible before release
4. Gathering evidence to sue the component vendors

Q: Which of the following is a risk of outsourced testing that might not apply to distributed testing?

1. Selection of an improper test partner
2. Communication problems created by time zone differences
3. Insufficient skills in some of the test team members
4. Inconsistent test processes across the testing locations

Q: 23. You are the manager of a bank's quality assessment group, in charge of independent testing for banking applications. You used quality risk analysis to allocate effort and prioritize your tests during test preparation. You are currently executing the system integration test.

You have finished running each test case once. You are not certain how much time the project management team will allow for additional test execution because regulatory changes might require the system to be activated ahead of schedule.

Based on the severity and priority of the bugs found by each test case, you have calculated weighted failure for each test case. You also know the risk priority number for each test case based on the quality risk item(s) the test case covers. Considering both the weighted failure and the risk priority number, you have reprioritized your test cases for the final days or weeks of testing.

Which of the following is a benefit this reprioritization will provide?

1. If testing is curtailed, you will have expended the minimum amount of effort possible.
2. If testing is extended, you will have time to run all of your tests.

- 3.** If testing is curtailed, you will have run the most important tests.
- 4.** If testing is extended, you will have covered all of the requirements.

Q: Which of the following is a benefit of exploratory testing and other reactive test strategies that would not apply to an analytical requirements-based test strategy?

- 1.** The ability to utilize a very experienced test team
- 2.** The ability to accurately predict the residual risk prior to delivery
- 3.** The ability to prevent defects during requirements analysis
- 4.** The ability to test effectively without a complete test basis

Chapter 4. Test Techniques

*"Eye of newt, and toe of frog,
Wool of bat, and tongue of dog,
Adder's fork, and blind-worm's sting,
Lizard's leg, and owlet's wing,
For a charm of powerful trouble,
Like a hell-broth boil and bubble."*

—William Shakespeare, from the play *Macbeth*

The fourth chapter of the Advanced syllabus is concerned with test techniques. The concepts in this chapter apply primarily for test analysts and technical test analysts. There are no learning objectives at any level defined for test managers in this chapter. However, as a test manager managing teams of test analysts and technical test analysts, it's good to be familiar with the main concepts and terms of their work. I recommend that you read [chapter 4](#) in the Advanced syllabus.

In addition, if you're studying for the ISTQB Advanced Test Manager exam, remember that certain concepts related to test techniques are covered at the Foundation level and thus are examinable. So you will certainly need to read [chapter 4](#) of the Advanced syllabus for familiarity and recall only. You should also review [chapter 4](#) of the Foundation syllabus, ensuring that you have mastered the learning objectives in that chapter.

ISTQB Glossary

boundary value analysis (BVA): A black-box test design technique in which test cases are designed based on boundary values.

branch testing: A white-box test design technique in which test cases are designed to execute branches.

cause-effect graphing: A black-box test design technique in which test cases are designed from cause-effect graphs.

classification tree method: A black-box test design technique in which test cases, described by means of a classification tree, are designed to execute combinations of representatives of input and/or output domains.

condition testing: A white-box test design technique in which test cases are designed to execute condition outcomes.

condition determination testing: A white-box test design technique in which test cases are designed to execute single-condition outcomes that independently affect a decision outcome.

control flow analysis: A form of static analysis based on a representation of sequences of events (paths) in the execution through a component or system.

D-D path: Not defined in the ISTQB glossary.

data flow analysis: A form of static analysis based on the definition and usage of variables.

decision table testing: A black-box test design technique in which test cases are designed to execute the combinations of inputs and/or stimuli (causes) shown in a decision table.

decision testing: A white-box test design technique in which test cases are designed to execute decision outcomes.

defect-based technique: See defect-based test design technique.

defect-based test design technique: A procedure to derive and/or select test cases targeted at one or more defect categories,

with tests being developed from what is known about the specific defect category.

defect taxonomy: A system of (hierarchical) categories designed to be a useful aid for reproducibly classifying defects.

dynamic analysis: The process of evaluating behavior—e.g., memory performance, CPU usage—of a system or component during execution.

error guessing: A test design technique where the experience of the tester is used to anticipate what defects might be present in the component or system under test as a result of errors made, and to design tests specifically to expose them.

equivalence partitioning: A black-box test design technique in which test cases are designed to execute representatives from equivalence partitions. In principle, test cases are designed to cover each partition at least once.

exploratory testing: An informal test design technique where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests.

experience-based technique: See experience-based test design technique. [Note: The glossary mistakenly refers to this an "experienced-based technique."]

experience-based test design technique: Procedure to derive and/or select test cases based on the tester's experience, knowledge, and intuition.

LCSAJ: A Linear Code Sequence and Jump, consisting of the following three items (conventionally identified by line numbers in a source code listing): the start of the linear sequence of executable statements, the end of the linear sequence, and the target line to which control flow is transferred at the end of the linear sequence.

memory leak: A defect in a program's dynamic store allocation logic that causes it to fail to reclaim memory after it has finished using it, eventually causing the program to fail due to lack of

memory.

multiple condition testing: A white-box test design technique in which test cases are designed to execute combinations of single-condition outcomes (within one statement).

pairwise testing: A black-box test design technique in which test cases are designed to execute all possible discrete combinations of each pair of input parameters.

path testing: A white-box test design technique in which test cases are designed to execute paths.

requirements-based testing: An approach to testing in which test cases are designed based on test objectives and test conditions derived from requirements; e.g., tests that exercise specific functions or probe nonfunctional attributes such as reliability or usability.

software attacks: See attack.

attack: Directed and focused attempt to evaluate the quality, especially reliability, of a test object by attempting to force specific failures to occur.

specification-based technique: See black-box testing.

black-box testing: Testing, either functional or nonfunctional, without reference to the internal structure of the component or system.

static analysis: Analysis of software artifacts—e.g., requirements or code—carried out without execution of these software artifacts.

statement testing: A white-box test design technique in which test cases are designed to execute statements.

state transition testing: A black-box test design technique in which test cases are designed to execute valid and invalid state transitions.

structure-based technique: See white-box test design technique.

white-box test design technique: Procedure to derive and/or

select test cases based on an analysis of the internal structure of a component or system.

test charter: A statement of test objectives, and possibly test ideas about how to test. Test charters are used in exploratory testing.

use case testing: A black-box test design technique in which test cases are designed to execute user scenarios.

wild pointer: A pointer that references a location that is out of scope for that pointer or that does not exist.

Chapter 5. Tests of Software Characteristics

Bridgekeeper: Stop! Who approaches the Bridge of Death must answer me these questions three, 'ere the other side he see.

Sir Robin: Ask me the questions, bridgekeeper. I'm not afraid.

Bridgekeeper: What is your name?

Sir Robin: Sir Robin of Camelot.

Bridgekeeper: What is your quest?

Sir Robin: To seek the Holy Grail.

Bridgekeeper: What is the capital of Assyria?

Sir Robin: I don't know that! [Flies screaming from the Bridge.]

Bridgekeeper: Stop! What is your name?

Sir Galahad: Sir Galahad of Camelot.

Bridgekeeper: What is your quest?

Sir Galahad: I seek the Holy Grail.

Bridgekeeper: What is your favorite color?

Sir Galahad: Blue. No yel— [Flies screaming from the Bridge.]

Bridgekeeper: [Laughs.] Stop! What is your name?

King Arthur: It is Arthur, King of the Britons.

Bridgekeeper: What is your quest?

King Arthur: To seek the Holy Grail.

Bridgekeeper: What is the air-speed velocity of an unladen swallow?

King Arthur: What do you mean? An African or European swallow?

Bridgekeeper: What? I don't know that! [Flies screaming from the Bridge.]

Squire Bedemir: How do know so much about swallows?

King Arthur: Well, you have to know these things when you're a king you know.

—King Arthur demonstrates his mastery of avian characteristics, in the geek-classic movie, *Monty Python and the Holy Grail*, written by the British comedy troupe, Monty Python.

The fifth chapter of the Advanced syllabus is concerned with tests of software characteristics based on the taxonomy of quality characteristics described in the ISO 9126 standard. The concepts in this chapter apply primarily for test analysts and technical test analysts. There are no learning objectives at any level defined for test managers in this chapter. However, as a test manager managing teams of test analysts and technical test analysts, it's good to be familiar with the main concepts and terms of their work. I recommend that you read [chapter 5](#) in the Advanced syllabus.

In addition, if you're studying for the ISTQB Advanced Test Manager exam, remember that certain concepts related to tests of software characteristics, especially the ISO 9126 standard, are covered at the Foundation level and thus are examinable. So you will certainly need to read [chapter 5](#) of the Advanced syllabus for familiarity and recall only. You should also review [Chapter 2](#), section 3, of the Foundation syllabus, to ensure that you have mastered the learning objectives in that chapter.

ISTQB Glossary

accessibility testing: Testing to determine the ease by which users with disabilities can use a component or system.

accuracy testing: Not defined in the ISTQB glossary.

accuracy: The capability of the software product to provide the right or agreed-upon results or effects with the needed degree of precision.

efficiency testing: The process of testing to determine the efficiency of a software product.

heuristic evaluation: A static usability test technique to determine the compliance of a user interface with recognized usability principles (the so-called "heuristics").

interoperability testing: The process of testing to determine the interoperability of a software product.

maintainability testing: The process of testing to determine the maintainability of a software product.

operational acceptance test (OAT): Operational testing in the acceptance test phase, typically performed in a simulated real-life operational environment by operator and/or administrator focusing on operational aspects; e.g., recoverability, resource behavior, installability, and technical compliance.

operational profile: The representation of a distinct set of tasks performed by the component or system, possibly based on user behavior when interacting with the component or system, and their probabilities of occurrence. A task is logical rather than physical and can be executed over several machines or in noncontiguous time segments.

portability testing: The process of testing to determine the portability of a software product.

recoverability testing: The process of testing to determine the recoverability of a software product.

reliability growth model: A model that shows the growth in

reliability over time during continuous testing of a component or system as a result of the removal of defects that result in reliability failures.

reliability testing: The process of testing to determine the reliability of a software product.

security testing: Attributes of a software product that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs and data.

suitability testing: Not defined in the ISTQB glossary.

suitability: The capability of a software product to provide an appropriate set of functions for specified tasks and user objectives.

Software Usability Measurement Inventory (SUMI): A questionnaire-based usability test technique to evaluate the usability—e.g., user satisfaction—of a component or system.

usability testing: Testing to determine the extent to which the software product is understood, easy to learn, easy to operate, and attractive to users under specified conditions.

Chapter 6. Reviews

NASA lost a \$125 million Mars orbiter because a Lockheed Martin engineering team used English units of measurement while the agency's team used the more conventional metric system for a key spacecraft operation, according to a review finding released Thursday. The units mismatch prevented navigation information from transferring between the Mars Climate Orbiter spacecraft team at Lockheed Martin in Denver and the flight team at NASA's Jet Propulsion Laboratory in Pasadena, California.

—From CNN, a news story on a bug that could have been found by a good set of design reviews.

The sixth chapter of the Advanced syllabus is concerned with reviews. As explained in the Foundation syllabus, reviews are a form of static testing where people, rather than tools, analyze the project or one of the project's work products, such as a requirements specification. The primary goal is typically to find and remove defects in that work product before it serves as a basis for further project activity, though other goals can also apply. The Advanced syllabus introduces additional types of reviews and covers strategies for effective and successful reviews. [chapter 6](#) of the Advanced syllabus has five sections:

1. Introduction
2. The Principles of Reviews
3. Types of Reviews
4. Introducing Reviews
5. Success Factors for Reviews

Let's look at each section and how it relates to test management.

6.1. Introduction

Learning objectives

Recall of content only

To succeed with reviews, an organization must invest in and ensure good planning, participation, and follow-up. A successful review is more than a room full of people reading a document.

Good testers make excellent reviewers. Good testers have curious minds and a willingness to ask skeptical questions, traits referred to as *professional pessimism* in the Foundation syllabus. They offer a technical viewpoint that is different from that of the programmers and other engineers, who are thinking about how to implement the software. Testers tend to think about how the software might break. That outlook makes testers useful in a review, though they have to remain aware of the need to contribute in a positive way.

That focus on a positive contribution is required not only of testers but of everyone involved. Because reviewing is a group activity, all participants must commit to well-conducted reviews. One negative, confrontational, or belligerent participant can damage the entire process profoundly.

Reviews are easy to do poorly and hard to do well, so many organizations abandon them after an initial period of unsuccessful reviews. However, done properly, reviews have one of the highest payoff rates of any quality-related activity.

6.2. The Principles of Reviews

Learning objectives

(K2) Explain the benefits of reviews compared to dynamic testing and other static testing techniques.

ISTQB Glossary

review: An evaluation of a product or project status to ascertain discrepancies from planned results and to recommend improvements. Examples include management review, informal review, technical review, inspection, and walkthrough.

reviewer: The person involved in the review that identifies and describes anomalies in the product or project under review. Reviewers can be chosen to represent different viewpoints and roles in the review process.

Let's discuss some review principles. These principles were also explained in the Foundation syllabus.

First, a review is a type of static test. So, the object being reviewed is not executed or run during the review. Like any test activity, reviews can have various objectives. One common objective is finding defects. Other objectives, typical of all testing, are building confidence that we can proceed with the item under review, reducing risks associated with the item under review, and generating information for the project team and for management. Unique to reviews are two additional common objectives. One is ensuring uniform understanding of the document and its implications for the project. The other is building consensus around the statements in the document.

Reviews usually precede dynamic tests. Reviews should complement dynamic tests. The cost of a defect increases the longer that defect remains in the system, so reviews should happen as soon as possible. However, because not all defects are easy to find in reviews, dynamic tests should still occur.

In addition to reviews, another type of static testing is static analysis. Like all static testing, static analysis does not involve execution of the object being tested. However, static analysis uses tools to detect defects. Reviews use the human mind to detect defects. Static analysis tools are methodical, unbending, not subject to boredom, and strict, which are not always traits that human reviewers can have. So certain tedious tasks, like checking millions of lines of code for coding standards violations or checking a

specification document with tens or hundreds of pages for spelling, grammar, and reading-difficulty problems, can and should be done through static analysis.

Woody Allen, the New York film director, is reported to have once said that "80% of success is showing up". That might be true in the film business, but Woody Allen would not be a useful review participant. Reviews require adequate preparation. If you spend no time preparing for a review, expect to add little value during the review meeting.

In fact, you can easily remove value by asking dumb questions that you could have answered on your own had you read the document thoroughly before showing up. You might think that's a harsh statement, especially in light of the management platitude that "there are no dumb questions". Well, sorry, there are plenty of dumb questions. Any question that someone asks in a meeting because of their own failure to prepare, resulting in a whole roomful of people having to watch someone else spend their time educating the ill-prepared attendee on something they should have known when they came in the room, qualifies as a dumb question. In fact, showing up for a review meeting unprepared qualifies as rude behavior to me, disrespectful of the time of the others in the room.

Because reviews are so effective when done properly, organizations should review all important documents. That includes test documents: test plans, test cases, quality risk analyses, bug reports, test status report, you name it. My rule of thumb is, anything that matters is not done until it's been looked at by at least two pairs of eyes. You don't have to review documents that don't matter, but here's a question for you: Why would you be writing a document that didn't matter?

So, what can happen after a review? There are three possible outcomes. The ideal case is that the document is okay as is or with minor changes. Another possibility is that the document requires some changes but not a re-review. The most costly outcome—in terms of both effort and schedule time—is that the document requires extensive changes and a re-review. Now, when that happens, keep in mind that, while this is a costly outcome, it's less costly than simply ignoring the serious problems and then dealing with them during component, integration, system, or—worse yet—acceptance testing.

6.2.1. Formal and Informal Reviews

In an informal review, there are no defined rules, no defined roles, no defined responsibilities, so you can approach these however you please. Of course, keep in mind that Capers Jones has reported that informal reviews typically find only around 20 percent of the defects, while very formal reviews like inspections can find up to 85 percent of the defects. If something is important, you probably want to have a formal review—unless you think that you and your team are so smart that no one is going to make any mistakes.^[1]

[1] See Capers Jones, *Software Assessments, Benchmarks, and Best Practices*. We'll discuss his findings further a bit later in this section.

A formal review should include the following essential roles and responsibilities:

- The manager: The manager allocates resources, schedules reviews, and the like. However, they might not be allowed to attend based on the review type.
- The moderator or leader: This is the chair of the review meeting.
- The author: The person who wrote the item under review. A review meeting, done properly, should not be a sad or humiliating experience for the author.
- The reviewers: The people who review the item under review, possibly finding defects in it. Reviewers can play specialized roles, based on their expertise or based on some type of defect they should target.
- The scribe or secretary or recorder: The person who writes down the findings.

Now, in some types of reviews, roles can be combined. For example, the author, moderator, and secretary can be the same person. In fact, as test manager, when I've had the test team review my test plans, I've often been the manager, the author, the moderator, and the secretary.

A review might involve some additional roles. We might involve decision makers or project stakeholders. This is especially true if an ancillary or even primary objective of the review is to build consensus or to disseminate information. In some cases, we can involve customer or user representatives. We recently did a review of mock-ups for our new website with our marketing team, our outsource web development team, and our

company executives. Because we had hired the web development team, for them we were the customers and, for some features, the users.

Certain very formal review types also use a reader, who is the person responsible for reading, paraphrasing, and explaining the item under review.

You can use more than one review type in an organization. For some documents, time is more important than perfection. For example, on our test teams, we apply the "two pairs of eyes" rule to mean that a tester must read another tester's bug report before it can be filed. However, for more visible documents like test plans, we use a walk-through with the entire test team. For critical documents, you can use more than one review type on a single item.

6.2.2. Case Study of an Informal Review

[figure 6-1](#) shows an example of feedback we received from the VP of software development when he did an informal review of our draft test plan. I've selected a few of the 10 or so comments he included in his email, summarizing his feedback. Let's see what kind of benefits we obtain from the feedback in [figure 6-1](#).

Figure 6-1. Example: Review benefits



Item 2, regarding hardware limitations at launch time, saved us a huge amount of potentially wasted effort developing load generators and test

scripts that could support 500, 000 users, since 100, 000 was the limit for the first release. That information was missing from the marketing requirements document, but the VP knew it. So, this is an example of a defect in a requirements specification showing up as a defect in a test plan, which fortunately was caught in this review.

Item 4 has the nice benefit of building consensus on the reliability requirements for the system. These types of nonfunctional requirements are often ill-defined in specifications, and this project was no exception. So, here we are using the test plan and the test plan review to build consensus on what expected results are for a particular test type.

Item 4 also includes, at the end, a request for clarification from the VP of software development about the amount of time required to run the tests. This provides the benefit of common expectations across the project team about an important detail of the project.

Item 5 indicates that we have pointed out a gap in the overall development process, namely the ability to triage and prioritize bugs for repair. In my experience, the best way to do this is via a cross-functional organizational team. This team is often called a Must Fix Committee, a Bug Triage Committee, or a Change Control Board. Whatever it should be called or is called in your organization, on this team it was missing. I had called attention to that in my test plan draft and the VP of software development had taken ownership of getting that broken process fixed. Because managing the list of must-fix bugs during test execution is a critical testing process, the VP of software development's acceptance of ownership to resolve this gap was a major benefit for the test team as well as the project team.

Item 8, like item 5, is another major benefit for testing. The VP of software development has accepted responsibility for cleaning up some problems we had with test release management so far during informal testing. We had asked for more structure, regular test releases, and detailed test release notes, and he agreed to all three.

ISTQB Glossary

informal review: A review not based on a formal (documented) procedure.

inspection leader (or moderator): The leader and main person responsible for an inspection or other review process.

walk-through: A step-by-step presentation by the author of a document in order to gather information and to establish a common understanding of its content.

6.3. Types of Reviews

Learning objectives

Recall of content only

The concepts in this section apply primarily for test analysts and technical test analysts. There are no learning objectives at any level defined for test managers in this section. However, if you plan to take the Advanced Test Manager exam, certain concepts related to management reviews and audits, as well as other types of reviews covered at the Foundation level, are examinable. In the course of studying for the exam, be sure to review the Foundation syllabus, [chapter 3](#), to cover reviews, and read this section of the Advanced syllabus.

It's also important for test managers, whether taking the Advanced Test Manager exam or not, to understand the main review types. The Foundation syllabus discusses four types of reviews, based on the IEEE 1028 standard, which we'll look at later in this chapter:

- At the lowest level of formality (and, usually, for defect removal effectiveness), we find the informal review. This can be as simple as two people, the author and a colleague, discussing a design document over the phone.
- Technical reviews are more formalized but still not highly formal. They are sometimes called peer reviews, which gives a clue about the typical participants.
- Walk-throughs are reviews where the author is the moderator and the item under review is the agenda. That is, the author leads the review, and in the review, the reviewers go section by section through the item under review.
- Inspections are the most formalized reviews. The roles are well defined. Managers may not attend. The author may be neither moderator nor secretary. A reader is typically involved.

As you can imagine, as the level of formality goes up, the rate of review—the number of pages per hour—goes down.

You should remember that the IEEE 1028 standard and the Foundation

syllabus are discussing idealized situations. In real-world practice, we commonly find organizations blending the parts they like from each review type and discarding parts they don't like. We also commonly hear organizations talking about walk-throughs, peer reviews, and inspections when the approach they use for that review type does not adhere to the IEEE 1028 rules.

While the Foundation syllabus ignored management reviews and audits from the IEEE 1028 standard, let's fill that gap here, starting with management reviews. Common purposes of management reviews are to monitor progress, assess status, and make decisions about some project, system, ongoing activity, or process. (Of course, in some organizations, management reviews are organized for various political reasons too, but we can ignore that for the moment.)

Managers involved with the item being reviewed often perform these management reviews. Various stakeholders and decision makers can assist them as well. The level of involvement of each of the participants can vary.

In some cases, organizations will hire outside consultants to come in and do reviews. For example, a large portion of RBCS's business is doing test process and quality process assessments of various kinds for organizations. These assessments are a hybrid between a management review and an audit, which we'll discuss in a moment. Test managers often drive these test assessments, in which case they are more like a management review. When outside test stakeholders drive these test assessments, the assessments are more like an audit.

Often, a management review involves assessing how a project is doing in terms of plans, estimates, project risk management, and so forth. A management review can also involve looking at the adequacy of various procedures and controls. Participants must prepare for these reviews, especially those who are going to deliver status information. We've done test assessments for organizations where people had so tenuous a grasp on what was going on in their testing group that our foremost recommendation was, "Get some metrics and tracking mechanisms in place immediately."

Typically, the outcome of a management review includes action items, recommendations, issues to be resolved, and the like. The decisions should be documented and the execution of action items and recommendations checked regularly. Unfortunately, organizations don't always follow through on these action items and recommendations.

Moving on to audits, these can be quite formal and, in some cases, quite adversarial. In an audit, there's a strong chance that people will be measured against a contract, a standard, an industry best practice, or the like; this can provoke defensiveness. The Advanced syllabus says that audits are the least effective review type for revealing defects, but that really depends on the auditing and audited organization. When we do testing audits for our clients, we are very, very good at finding defects, both project defects and process defects.

One essential element of an audit is the independent evaluation. Like a management review, we can measure a process, a project, an ongoing activity, or a system. However, another essential element of an audit is the idea of being in or out of compliance. Audits can be done by a lead auditor with an auditor team or by a single auditor. The auditors collect evidence by conducting interviews, witnessing, examining documents, analyzing metrics, and so forth. I find the interviews particularly interesting, especially when an audit has become high stakes for some of the participants. Attempts to spin, mislead, misdirect, convince, and stall the auditor occur in such situations.

As with the management review, the outcome of an audit can include action items, recommendations, issues to be resolved, and the like. However, it also includes an assessment of compliance (or noncompliance). This is often measured against a multidimensional scale or checklist, so even if 99 items are in compliance, if one is out of compliance, the organization might fail the audit. If noncompliance were the finding, then the audit findings typically include corrective actions for the item or items that failed. The organization should document the decisions taken based on the audit. It should regularly check the execution of action items, recommendations, and corrective actions along with periodically reassessing compliance. In regulated industries or for legally mandated audits, follow-through on audit results is usually excellent, but organizations in nonregulated industries often do not follow through completely on all action items and recommendations.

ISTQB Glossary

audit: An independent evaluation of software products or processes to ascertain compliance to standards, guidelines, specifications, and/or procedures based on objective criteria, including documents that specify (1) the form or content of the products to be produced, (2) the process by which the products shall be produced, and (3) how compliance to standards or guidelines shall be measured.

inspection: A type of peer review that relies on visual examination of documents to detect defects; e.g., violations of development standards and nonconformance to higher-level documentation. The most formal review technique and therefore always based on a documented procedure.

management review: A systematic evaluation of software acquisition, supply, development, operation, or maintenance process performed by or on behalf of management that monitors progress, determines the status of plans and schedules, confirms requirements and their system allocation, or evaluates the effectiveness of management approaches to achieve fitness for purpose.

technical review: A peer group discussion activity that focuses on achieving consensus on the technical approach to be taken.

6.4. Introducing Reviews

Learning objectives

(K2) Compare review types with each other and show their relative strengths, weaknesses, and fields of use.

(K3) Lead a review team through a formal review following the steps identified.

(K4) Outline a review plan as part of a quality/test plan for a project considering review techniques, defects to be found, available skills of staff, and alignment with appropriate dynamic testing approaches.

The following steps are useful in successfully introducing reviews:

- Secure management support: Reviews are not expensive from a budget point of view, like test automation is, but they do require a time commitment, especially when time is tight.
- Educate managers: You need to have an honest conversation about the business case for reviews, including the costs, benefits, and potential issues. Avoid exaggerating.
- Put structure in place: Have documented review procedures for the various types of reviews you'll use. Have templates and forms available. Establish an infrastructure such as a reviews metrics database, making sure that people understand the use of the metrics and will help you gather them. If you intend to do geographically distributed reviews, make sure you have the tools in place for that.
- Train: Educate the participants on review techniques and procedures.
- Obtain participant support: Make sure that those who will do the reviews and those whose work will be reviewed are comfortable and supportive.
- Do some pilot reviews: Plan to make some mistakes ... and learn from them.
- Demonstrate the benefit: You have a defined business case, right? Now show management that you achieved what you promised. You'll need metrics to do this.

- Apply reviews to all (or at least the most important) documents: Requirements, contracts, project plans, test plans, quality risk analyses, and similar high-visibility documents are obvious targets. However, I have found simply ensuring informal reviews of bug reports to be amazingly valuable.

You won't necessarily need to do every step in every organization, and you don't need to do these steps in perfect, sequential order, but you should think long and hard about why it's okay to skip a step if you think it is.

Your organization will invest time and money in reviews. Managers will expect a return on that investment. As a test manager, you should be able to demonstrate that return. The concept of demonstrating the return on the review investment is very similar to the concept of demonstrating return on the overall test investment, which I illustrated in [chapter 3](#).

To demonstrate a return on the review investment, you can use metrics like the reduced or avoided cost of fixing defects or dealing with failures. What does a defect cost in system test? How about after release? A simple spreadsheet can show the benefits of reviews and evaluate the success of the reviews after their implementation.

Don't forget to measure the return in terms of saved time too. Money is not always the biggest concern for managers. In fact, time to market is usually a bigger issue. So if you can document that a defect takes 5 hours to resolve when found in a review and 25 hours when found in system test, you have a solid business case for how time investment in reviews during the early stages of a project reduces the likelihood of project delay at the end of the project.

Having established metrics, it's important to continue to monitor them. It's easy for review processes to become ritualistic and stuck, and then the value goes down. If you see the benefit dropping off, ask yourself why. In fact, the benefit should constantly increase. You should look for metrics-based, measurable ways to improve the review processes. Make sure that you—and your managers and the participants—see reviews and review process improvement as a long-term investment.

Table 6-1. Review types and effectiveness

	Least	Average	Most
Requirements review	20%	30%	50%
High-level design review	30%	40%	60%

Functional design review	30%	45%	65%
Detailed design review	35%	55%	75%
Code review	35%	60%	85%

6.4.1. Defect Removal Effectiveness of Reviews

Earlier in this chapter, I mentioned the differences between review types and formality and how those differences influenced defect removal effectiveness. I specifically cited some figures from Capers Jones. Let's examine the data behind Jones's observations. In Jones's studies of thousands of projects across hundreds of clients, he has found some interesting data on reviews, their applications, and the effectiveness of various types of reviews. Jones mentions that the informal reviews are the least effective, while reviews that have some but not all elements of formality are about average, and the most effective are the highly formalized inspections. Of course, for reviews to be effective at any level of formality, you have to do them well and you have to have organizational support for the process.^[5]

[5] These figures and the figures in table 1 are derived from Capers Jones, *Software Assessments, Benchmarks, and Best Practices*.

Table 6-1 shows the range of defect removal effectiveness that Jones has observed for various types of reviews. Defect removal effectiveness is the percentage of defects that were present at the time of some static or dynamic testing activity and successfully detected and removed by that testing activity. Jones analyzes reviews based on two factors. The first factor is the type of work product being reviewed. The second factor is the level of formality used for the review. As you can see, both factors have a strong influence on the defect removal effectiveness. If you think of the reviews as a series of filters, here's a quick mathematical demonstration of how effective reviews can be.

First, imagine that you started with 1, 000 defects. You follow worst practices in reviews, but at least you review all types of items. In this case, you would enter testing with about 166 defects. Now, imagine that you started with 1, 000 defects again. However, this time you follow best practices (and again you review all types of items). This time, you go into testing with 3 defects.

6.4.2. Two Review Case Studies

Below you see an outline of the formal change management process used by one client. These change control board meetings were recurring, formal management review meetings.

Change Control Board Process

Revision 1.0

- 1. Scope:** The Change Control Board will facilitate change requests at Netpliance for initial release and changes to production released products.
- 2. Submit Change Request:** Change requests should be submitted through Development or Program Management. A change request must be written and should include, at minimum, the following criteria:
 - 1.** Definition of the change (what is the change and why is it needed?)
 - 2.** Areas impacted (including but not limited to Device HW, Client SW, Server Team, Network Team, ID, Manufacturing Process)
 - 3.** Documentation to be updated (including but not limited to Plan of Record document, Engineering Specification(s), UI spec, user guide)
 - 4.** Level of Change

Emergency—Requires immediate attention. Must have accompanying disposition plan

Normal—Process normally

Running—Process as part of next normal change
 - 5.** Dependencies (on other pending changes, on pending decisions, on assumptions)
 - 6.** Estimated dollar impact to the organization, as applicable (savings or cost)
 - 7.** Requested date of implementation
- 3. CCB Review:**
 - 3.1 Team Members:** Configuration Management shall solicit

approvals from required parties. Normal change requests will be queued for review at regularly scheduled CCB meetings. Urgent requests will be immediately hand-carried and/or emailed to the appropriate parties for approval. Required signatures will be on a case-by-case basis as defined in 1b. Guidelines for required signatures are as follows:

1. Development

- Device only (Qadeer, Craig, or Ken)
- Client only (Wayne, Craig, or Ken)
- Device+Client only (Craig or Ken)
- Server only (David G. or Ken)
- Server and Device/Client (Ken)

2. Test (Rex or Greg)

3. Support (Chris or Tom)

4. Network Operations (Steve)

5. Supply Chain (Maeve or Greg)

6. Program Management/Release Management (Albert or Greg)

7. Finance (Brandon or David H.)

8. Marketing (Munira or Annie)

4. *Signature Criteria:*

The following outlines the signature criteria for all changes (including initial releases)

- 1. Device Team—**Signature indicates that either the change has no impact to the functionality of the device or the impact to the device has been fully tested and there are no major customer impact issues.
- 2. Client Team—**Signature indicates that either the change

has no impact to the functionality of the software image or the impact to the software image has been fully tested and there are no major customer impact issues.

- 3. Server Team**—Signature indicates that either the change has no impact to the functionality of the server environment or the impact to the server environment has been fully tested and there are no major customer impact issues.
- 4. Test Team**—Signature indicates that the change has been subject to the complete set of tests, and that the results have been posted to the team.
- 5. Support Team**—Signature indicates that the change has not introduced any customer impact issues or that any issues introduced have a corresponding customer response documented and posted on the support database.
- 6. Network Operations**—Signature indicates that the network operation infrastructure is ready to support the change.
- 7. Supply Chain**—Signature indicates that supply is available to support the change and that the bill of material has been updated to reflect the change.
- 8. Release Management**—Signature indicates that all appropriate signatures from the CCB have been captured for the change. Signature indicates that all required attachments have been placed under document control. Signature indicates that the approved ECO will be communicated to all parties for implementation.
- 9. Finance**—Signature indicates that the financial impacts are understood and they are acceptable.
- 10. Marketing**—Signature indicates that this change is required.

5. *Attachment Requirements:*

- 1. Device changes**—Specification, drawing, test plan, test results

2. Client software changes—Release notes, tar file, test plan, test results
3. Server changes—Release notes
4. Documentation/packaging changes—Revised drawings and documentation
6. **Approved?** Each member of the Change Control Board, as outlined in section 2 and as required by item 1b, will be required to give approval for each change request. The approval should include a commit date representing the readiness date for the approver's functional area. Once all approvals are received by Configuration Management, a notice will be sent out to all parties. If one or more approvers do not approve a particular change, the initiator will be notified and the change will be withdrawn.
7. **Implement Change:** All affected parties will be responsible for implementing the change as committed to during the approval cycle. The Configuration Manager will be responsible for communicating all changes to the appropriate parties and for collecting related documentation as necessary.

Let me point out some key areas of this document.

First, notice that we have a well-defined scope. The change control board (CCB) is there to facilitate change requests in terms of the definition of the initial release as well as subsequent changes to production-released products.

There is a requirement that, before the meeting, the people who are proposing a change must adequately prepare their request. There are rules about how to submit a change request. Those rules address both the channel through which the submitter should submit the request and the content that the submitter must include.

The document then goes on to describe how the CCB should review these requests. It starts by defining the team members of the CCB. This team includes a complete cross-section of the project team and all stakeholders.

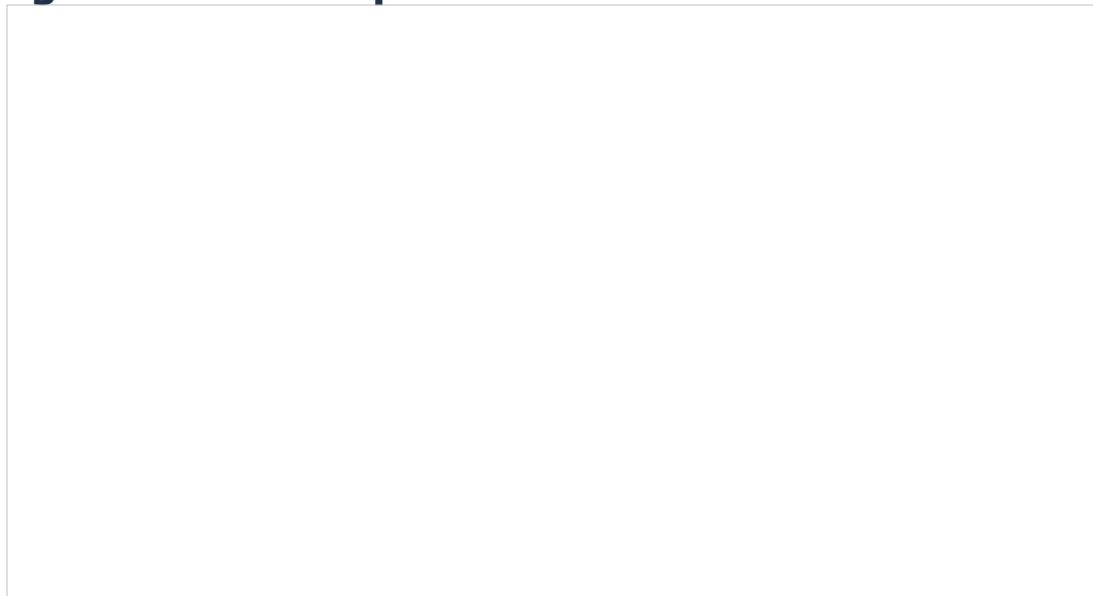
Each member group played a defined role. Subsection 3.2, "Signature Criteria", describes that role. The document also defines here what it meant when a team member signed a change request. Signing a change request did not necessarily indicate approval of the request. In some cases, the signature meant, "Yes, I understand and will support this change, but I don't think this is a good idea." The signature by the test team specifically means that we will test or have tested the change but not that we feel that the test results support its release.

In addition to the change request itself, in many cases there were additional attachments required for the document to be considered complete.

In section 4, "Approved?" the document talks about how to note the disposition of the change request. Finally, section 5, "Implement Change", describes the requirement for each group to play their part once the change is approved and going into production.

In [Figure 6-2](#), you see a list of ambiguous areas found in the marketing requirements document (or MRD) for the client-side system on the Internet appliance project that I've mentioned a few times in this book. This figure shows part of a six-page list. Each page described about a dozen or so ambiguities that we had found in this document during review.

Figure 6-2. Example: Defects found in MRD review



6.4.3. Introducing Reviews Exercise 1

Let's travel backward in time for a moment, to the beginning of the HELLOCARMS project, prior to the start of iteration one. Assume that due to your expertise with quality assurance, management has assigned you the

job of planning a review of the HELLOCARMS System Requirements Document.

Identify the following:

- Which review technique(s) you would apply and why
- What level of formality you would apply and why
- What defects you would expect to find
- What skills you would need in the review team

If you are working in a classroom, break into groups of three to five. If you are working alone, you'll need to do this yourself. If you are working in a classroom, once each group has finished, discuss the results.

I suggest allotting 15 minutes for this exercise, including the discussion.

6.4.4. Introducing Reviews Exercise 1 Debrief

Here are my answers to each of the four topics.

Which review technique(s) you would apply and why

My experience with doing reviews on documents like these and on projects like this one would lead me to use a walk-through. I find it the most accessible form of review, the one least likely to get sidetracked, and it's reasonably effective. It also has a strong side effect of building consensus and educating the project team, assuming you invite the right people. However, because it is not as effective as, for example, an inspection, I would also use static analysis and quality risk analysis (based on the requirements) to detect and remove additional problems.

What level of formality you would apply and why

Unless the Globobank project team has experience with highly formal reviews, I would keep the level of formality low and rely more on the multiple-filter approach discussed in the previous paragraph to ensure a high rate of defect removal for this critical document.

What defects you would expect to find

I would hope to find a large number of ambiguities, contradictions, and

untestable requirements because just a quick read of the document reveals such problems. If the review resulted in few of these and a disproportionate number of grammar, spelling, formatting, and other cosmetic errors, I would try to schedule a new review after cleaning up the document first.

What skills you would need in the review team

Basically, the same cross-functional team that participated in the quality risk analysis should participate here.

6.4.5. Introducing Reviews Exercise 2

If you are working in a classroom, break into groups of three to five. If you are working alone, you'll need to do this yourself. Select a review type and apply that review type to the HELLOCARMS System Requirements Document Focus on serious problems rather than superficial ones.

If you are working in a classroom, once each group has finished, discuss the results.

I suggest allotting 30 minutes for the review, followed by 15 minutes for the discussion.

6.4.6. Introducing Reviews Exercise 2 Debrief

Senior RBCS Associate Jose Mata reviewed the HELLOCARMS System Requirements Document. He used a checklist from Karl Wiegers that was discussed in the Advanced Test Analyst course and provided the following feedback.^[1]

[1] You can find this checklist in Karl Wiegers's *Software Requirements*.

- Are internal cross-references correct? If we refer to something in another document or within this document, is that reference valid?
 - No. Section 010-010-040 states, "Field validation details are described in a separate document," but that document is not identified anywhere in the requirements document.
- Is the level of detail consistent and appropriate?
 - No. As an example, see section 010-010-180: "Provide features and screens that support the operations of Globobank's retail branches." That is too vague to be actionable.

- Section 010-010-170 states, "Support the submission of applications via the Internet, which includes the capability of untrained users to properly enter applications." This is a huge, and vague, requirement.
- Sections 010-010-190 through 010-010-240 start with "Support the marketing, sales, and processing of ...," which is so vague that important functionality can be missed.
- Do the requirements provide an adequate basis for design?
 - No. Section 010-010-070 states, "Ask each applicant whether there is an existing relationship with Globobank; e.g., any checking or savings accounts," but the list is not complete, and it should be.
 - Section 010-010-080 states, "Maintain application status from initiation through to rejection, decline, or acceptance ...," but we don't know if these states are a subset or if they are comprehensive.
 - Section 010-010-150 states, "Provide inbound and outbound telemarketing support for all States, Provinces, and Countries in which Globobank operates," but the list is not defined.
 - Section 010-010-160 states, "Support brokers and other business partners by providing limited partner-specific screens, logos, interfaces, and branding," yet screens or areas of the interface, are not identified.
 - Section 010-010-250 states: "Support flexible pricing schemes including introductory pricing, short-term pricing, and others," but the "and others" needs to be defined.
 - Is the priority of each requirement included?
 - Yes.
- Are all external interfaces defined?
 - No. We don't know how complete the information is. Data structures are hinted at, but not defined. The implied interfaces are with the following:
 - LoDoPS: 010-010-050, 010-010-100, 010-020-050, 010-030-

040, 010-030-060, 010-030-070, 010-030-080, 010-030-120,
010-030-130, 010-030-140, 010-030-145

- GLADS: 010-010-070
 - Scoring Mainframe: 010-020-020, 010-030-020
 - GloboRainBQW: 010-030-010
- Is there any missing information? If so, is it clearly marked as TBD ("to be determined")?
 - Yes.
 - Is the expected behavior documented?
 - No. For example, section 010-010-080 states, "Maintain application status from initiation through to rejection ...," but how and where the status is maintained is not stated.
 - Is each requirement clear, concise, and unambiguous?
 - No. For example, section 010-010-070 states, "Ask each applicant whether there is an existing relationship with Globobank," but it is unclear how the applicant is asked.
 - Sections 010-010-100, 010-030-040, and 010-030-070 state, "Allow user to indicate on a separate screen which, if any, are existing debts that the Customer will retire...," but it's not clear what the screen is supposed to be separate from.
 - Section 010-040-010 states, "Support agreed-upon security requirements (encryption, firewalls, etc.)," which is vague.
 - Section 010-040-060 states, "Support fraud detection for processing of all financial applications." This is vague, which is especially bad for a priority 1 requirement.
 - Is each requirement verifiable? Could you design a test to show that this requirement was met or not met?
 - No. Section 010-010-150 states, "Provide inbound and outbound telemarketing support for ...," which is vague and thus not verifiable.

- Sections 010-010-190 through 010-010-240 start with "Support the marketing, sales, and processing of ...," and the marketing part is not verifiable.
 - Section 010-010-250 states, "Support flexible pricing schemes including introductory pricing, short-term pricing, and others," and the "and others" part is not verifiable.
 - Section 010-030-150 states, "Support computer-telephony integration to provide customized marketing and sales support for inbound telemarketing campaigns and branded business partners," which is vague and thus not verifiable.
- Is each requirement in scope?
 - No. For example, section 010-010-170 states, "Support the submission of applications via the Internet, which includes the capability of untrained users to properly enter applications." This is beyond the scope of section 003, because allowing Internet-based customers is slated for subsequent releases.
 - Section 010-040-030 states, "Allow outsourced telemarketers to see the credit tier but disallow them from seeing the actual credit score of applicants." This is beyond the scope of section 003.
 - Section 010-040-050 states: "Allow Internet users to browse potential loans without requiring such users to divulge ..." This is beyond the scope of section 003.
 - Is each requirement free from content and grammar errors?
 - Yes.
 - Can the requirements be implemented within constraints?
 - Possibly not. Section 010-040-060 states, "Support fraud detection for processing of all financial applications." This might not be possible to implement. Specific checks would need to be defined.
 - Are all security and safety considerations properly specified?
 - No. Specific types of users, and their permissions, are not defined. User name and password strength are not addressed. Encryption of specific data is not addressed. Maintenance and purging

requirements are not addressed. Server physical security requirements are not addressed.

- Is each requirement uniquely and correctly identified? Is the granularity of the requirements such that it will be possible to have traceability from tests to requirements?
 - No. For example, sections 010-010-190 through 010-010-240 start with "Support the marketing, sales, and processing of ..." The granularity of these requirements is too large.
 - Section 010-010-250 states, "Support flexible pricing schemes including introductory pricing, short-term pricing, and others." This and several other compound requirements would be clearer if they were separately numbered requirements. It may be somewhat repetitious, but the requirements would be clearer and there would be more balance in scoping development and test efforts.
- Have we stayed in the proper realm of requirements, not design? In other words, are all requirements actually requirements and not design or implementation solutions?
 - No. Sections 010-010-100, 010-030-040, and 010-030-070 state, "Allow user to indicate on a separate screen which, if any, are existing debts that the Customer will retire ..." Specifying a separate screen appears to be a design detail.

6.5. Success Factors for Reviews

Learning objectives

(K2) Explain the risks behind not considering the technical, organizational factors and people issues for performing reviews.

There are a number of factors that influence the success—or, if absent, the failure—of reviews. The syllabus classifies those into three groups: technical factors, organizational factors, and people factors. Let's start with the technical factors.

First, ensure that you are following the defined process correctly. This can be particularly tricky for formal types of reviews like inspection. Now, that doesn't mean you can't tailor these processes, but it's usually a good idea to master them first.^[1]

^[1] A good reference for inspections is Tom Gilb and Dorothy Graham's book *Software Inspection*.

Now, I mentioned this matter of the business case. To support your business case, you have to record the costs of reviews (particularly in terms of effort) and the benefits that the organization obtains. A problem with reviews is that the benefits accrue long after the organization incurred the cost. That's true for all testing, of course, but it's especially acute for reviews, particularly if you forget to measure the value.

Remember that you don't have to wait until a document is done before you start reviewing it. You can and should review early drafts or partial documents when you're dealing with something critical. This can help to identify and prevent patterns of defects before they are built into the whole document.

That said, make sure you have some rules about what it means for something to be ready for review. You can waste people's time by sending them materials that aren't sufficiently mature to be reviewed. You can also waste people's time and frustrate them by sending them stuff to review that's still changing. People who are frustrated because they are wasting their time on some activity tend to find ways to stop wasting their time on that activity, which means that the review process can wither away. So have some entry criteria, which should include the simple rule that everyone has to show up prepared.

Checklists are helpful for reviews. It's too easy to forget important areas without them. So have some checklists. You can start with checklists from reputable industry experts, but make sure to extend them to be specific to your organization. They should address common defects based on what you find. Also, have different checklists for different kinds of documents, such as requirements, use cases, and designs. Finally, have different checklists for different review processes.

The appropriate level of formality varies. Be ready to use more than one type of review. Consider your objectives. Is the idea to do a quick document cleanup before sending it to a client? To improve some technical design decisions? To educate stakeholders? To generate information for management?

I've mentioned my rule of "two pairs of eyes", and I try hard not to violate that rule. Sometimes, deadlines intervene. However, you should review—or, better yet, inspect—all documents that are vitally important. If an important decision will be made, such as whether to sign a contract, be sure to inspect the proposal, contract, or high-level requirements specification first. If a major expenditure is being contemplated, have a management review to authorize it.

For large documents, you can use a sampling of a limited subset to estimate the number of defects in the entire document. Based on your estimate of the total number of defects, you can determine if a re-review is needed. Keep in mind that this sampling approach won't work for a document cleanup or edit.

Watch out for distractions. It's easy to find a bunch of minor format, spelling, and grammar errors. Focus on finding the most important defects, based on content not format.

Finally, as I mentioned before, continuously improve the review process by applying what you learn.

Now, some organizational factors. First, make sure that you and the other managers on the project team will plan and estimate for adequate time, especially under deadline pressures. It is a false economy to think that by skipping highly efficient bug removal activities early in the process, somehow the schedule end date will be accelerated, but that kind of thinking is rampant in software engineering.

You need metrics to measure the benefits of reviews, but be careful with

the metrics. For one thing, remember that some reviews will find many defects per person-hour invested while others won't. There are some mathematical models for predicting defect density, but they are beyond the scope of this book. Be careful not to use simplistic models. Most importantly, never ever let review defect metrics be used for individual performance evaluations. That introduces a level of defensiveness that will kill the process.

Make sure to allow time for rework of defects. It's a classic testing worst practice to assume that a test activity will conclude without finding any defects.

Ensure that the process involves the right participants. A study by Motorola in the mid-2000s showed that the right participants were the strongest indicator of review success. One aspect of being the right participant for a review involves having technical or subject matter expertise, of course. Having the right participants means having balance, making sure the review team has representatives from all key groups. And, having the right participants means having participants who understand the review process, usually through training especially for formal types of reviews. The second-strongest indicator, the study found, was having the right number of participants, so be sure to think carefully about who and how many.^[1]

[1] Jeff Holmes, "Identifying Code-Inspection Improvements Using Statistical Black Belt Techniques", *Software Quality Professional*, December 2003, Volume 6, Number 1.

If you are in a medium to large organization that is using reviews, have a review forum to allow people to share their experience and ideas. This forum can be reserved for moderators or leaders.

There's no point in having people at a review if they don't contribute, so ensure that the participants participate. Part of this is ensuring proper preparation. Another part is drawing less-vocal participants into the meeting. Just because someone doesn't have a forceful personality doesn't mean they don't have good ideas.

When dealing with critical documents, apply the strongest, most formal techniques. Remember Jones's figures on review effectiveness. What percentage of defects can you afford to leave in each kind of document?

Make sure you have a process in place for review process improvement. If this isn't supported by metrics, it's likely to point you in the wrong

direction. Make sure that the review process improvement process includes a mechanism to recognize and celebrate the improvements gained.

Finally, some people issues. First, as with other managers, educate all stakeholders and participants to expect defects. Make sure that's not an unpleasant surprise to them. Make sure they have allowed for rework and re-review time. People tend to overbook themselves in today's workplace. If they do so, being confronted with a list of issues to resolve in their document is likely to be a traumatic experience because it means overtime.

The review leader is not Torquemada, the Grand Inquisitor of the Spanish Inquisition. The rack, the iron maiden, and waterboarding are not review tools or techniques. Reviews should be a positive experience for authors, where they learn how to do their job better from respected peers. I can still remember review sessions with two or three mentors early in my career that helped me grow significantly. That said, if authors have had bad experiences, be careful with forcing them to consent to a review. It's best if management handles this.

Given how efficient defect location and removal is during reviews, we should be happy, not unhappy, when we find defects. Make sure people see that as an opportunity to save time and money. Don't look to point fingers or assign blame when defects are found.

Monitor the dialog in the room. We want constructive, helpful, thoughtful, and objective discussion. Make sure that people are thinking about the deeper issues, including how the document under review fits into the broader picture of the project.

6.5.1. Revisiting an Earlier Case Study

Earlier this section showed some of the myriad issues identified during a review of the client-side marketing requirements document on the Internet appliance project. Interestingly, not all of the questions and issues the test team identified in its review of that document were resolved prior to—or even during—implementation. Of course, I guess you could say, "Well, Rex, ultimately some programmer made a decision and that resolved the question or issue."

I would agree with that—ultimately, the system *does* something in every given situation, and that something is generally what the programmer told it to do. However, I would suggest that this is not the best way to solve serious questions about how a system should behave.

What happened was that Development and Marketing decided to proceed with open questions. We were told to stop pestering them with these questions and to proceed with developing and ultimately executing tests. We did some estimating of where we were in test design and implementation versus where we needed to be and how hard it would be to get there without additional information. We concluded we would incur about a 20 to 25 percent test inefficiency due to the lack of clearly defined product requirements. To this, our client contact memorably responded, "I would rather pay RBCS the extra money than have to teach these guys how to write requirements right now."

I replied with a smile, "Well, as long as your checks don't bounce, that is your decision to make."

During test execution, this inefficiency manifested itself in part in a high rate of false positives and false negatives. It also reduced test coverage, since we had to spend time during test execution documenting what the system actually did, what we thought it should do, and the differences between the two. So, what we have here is a classic example of failing to plan and estimate for adequate time for reviews and the resolution of the issues they reveal under deadline pressures.

6.5.2. IEEE 1028 Standard for Reviews

Let's review the IEEE 1028 standard for reviews, shown in figure 6-3 and introduced in the Foundation syllabus. The first section of the standard is an overview. It covers the purpose of the standard, the scope of coverage, and guide-lines for conformance with the standard, the organization of the standard, and how to apply the standard in an organization.

Figure 6-3. IEEE 1028 standard for software reviews

The second section is "References", which, as you might imagine, refers to other documents, standards, and so forth. The third section, "Definitions", defines terms used in the standard.

The fourth section addresses management reviews. Management reviews were out of scope at the Foundation level. In this section, the standard talks about who has what responsibilities in a management review, the inputs to and outputs from a management review, the entry criteria to start such a review and the exit criteria to recognize when it's complete, and the procedures a management review should follow.

The fifth section addresses technical reviews. In this section, the standard talks about who has what responsibilities in a technical review, the inputs to and outputs from a technical review, the entry criteria to start such a review and the exit criteria to recognize when it's complete, and the procedures a technical review should follow.

The sixth section addresses inspections. As with the previous two sections, the standard talks about who has what responsibilities in an inspection, the inputs to and outputs from an inspection, the entry criteria to start an inspection and the exit criteria to recognize when it's complete, and the procedures an inspection should follow. However, because inspections are more formal than technical reviews, the standard also discusses collecting data from the review process and implementing improvements to the review process.

The seventh section addresses walk-throughs. As with the other sections, the standard talks about who has what responsibilities in a walk-through, the inputs to and outputs from a walk-through, the entry criteria to start a walk-through and the exit criteria to recognize when it's complete, and the procedures a walk-through should follow. Since walk-throughs are of a similar level of formality as inspections, the standard also discusses collecting data from the review process and implementing improvements to the review process.

Finally, the eighth section addresses audits. As with the other sections, the standard talks about who has what responsibilities in an audit, the inputs to and outputs from an audit, the entry criteria to start an audit and the exit criteria to recognize when it's complete, and the procedures an audit should follow.

6.6. Sample Exam Questions

To end each chapter, you can try one or more sample exam questions to reinforce your knowledge and understanding of the material and to prepare for the ISTQB Advanced Level Test Manager exam.

Q: 1. Which of the following is a type of defect that you can detect more easily in a review than by a dynamic test?

1. Regression
2. Maintainability
3. Performance
4. Reliability

Q: Which of the following is a type of review in which you would expect to detect the greatest percentage of defects present in the item under review?

1. Informal
2. Static analysis
3. Simulation
4. Inspection

Q: Assume you are a test manager working on a project to create a programmable thermostat for home use to control central heating, ventilation, and air conditioning (HVAC) systems. In addition to the normal HVAC control functions, the thermostat has the ability to download data to a browser-based application that runs on PCs for further analysis.

During quality risk analysis, you identify compatibility problems between the browser-based application and the different PC configurations that can host that application as a quality risk item with a high level of likelihood.

Select all of the following actions that should occur in the quality and

test plans to best ensure that the project team will minimize this risk upon release?

1. Carefully select the list of supported configurations.
2. Review the list of supported configurations with project stakeholders.
3. Support only a single browser based on technical attributes.
4. Downgrade the likelihood of the risk based on available test resources.
5. Test the supported PC configurations early in test execution.

Q: Assume you are a test manager in charge of integration testing, system testing, and acceptance testing for a bank. You are working on a project to upgrade an existing automated teller machine system to allow customers to obtain cash advances from supported credit cards. You have received a requirements specification that states that the system should allow cash advances from \$20 to \$500, inclusively, for all supported credit cards. Assume that the bank is required contractually to support the following credit cards: American Express, Visa, Japan Credit Bank, Eurocard, and MasterCard.

The bank has given you the responsibility of organizing a review of the requirements specification. Which of the following is a risk for this review?

1. You do not know the supported credit cards.
2. You do not include the proper stakeholders in the review.
3. You do not know which test levels to address.
4. You do not receive the requirements specification.

Chapter 7. Incident Management

"If I extrapolate the current trend, then 100% of the 2017 release budget will be spent on contemplating changes, with 0% actual change. On the positive side, this means testing the 2017 release will cost nothing."

—V.V.L., European test engineer.

The seventh chapter of the Advanced syllabus is concerned with incident management. As the Foundation syllabus explains, an incident has occurred any time the actual results of a test and the expected results of that test differ. The Advanced syllabus uses the IEEE 1044 standard to focus on incident lifecycles and the information testers should gather for incident reports. [Chapter 7](#) of the Advanced syllabus has six sections:

- 1.** Introduction
- 2.** When Can a Defect Be Detected?
- 3.** Defect Lifecycle
- 4.** Defect Fields
- 5.** Metrics and Incident Management
- 6.** Communicating Incidents

Let's look at each section and how it relates to test management.

7.1. Introduction

Learning objectives

Recall of content only

Incident management is an essential skill for all testers and test managers. As test managers, we are more concerned with the process. We need defects to flow, smoothly and timely, from recognition to investigation to action to disposition. Testers are mostly concerned with accurately recording incidents and then performing the proper confirmation testing and regression testing during the disposition part of the process.

Testers will have a somewhat different emphasis depending on their role. Test analysts compare actual and expected behavior in terms of business and user needs. Technical test analysts evaluate behavior of the software and might need to apply further technical insight.

7.2. When Can a Defect Be Detected?

Learning objectives

Recall of content only

We can detect defects through static testing, which can start as soon as we have a draft requirements specification. We can detect failures, which are the symptoms of defects, through dynamic testing, which can start as soon as we have an executable unit.

Testing is a filtering activity, so to achieve the highest possible quality, we should have static and dynamic test activities pervasive in the software lifecycle. In addition to filtering defects, if we have lots of earlier filters like requirements reviews, design reviews, code reviews, code analysis, and the like, we will have early defect detection and removal. Early detection and removal reduces overall costs and reduces the risk of schedule slips.

During dynamic testing, when we see a mismatch between expected and actual results, we should not automatically assume that this indicates a defect in the system under test. Defects can exist in tests too.

ISTQB Glossary

defect (or bug): A flaw in a component or system that can cause the component or system to fail to perform its required function—e.g., an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system.

error: A human action that produces an incorrect result.

failure: Deviation of the component or system from its expected delivery, service, or result.

incident: Any event occurring that requires investigation.

incident logging: Recording the details of any incident that occurred, e.g., during testing.

root cause analysis: An analysis technique aimed at identifying the root causes of defects. By directing corrective measures at root causes, it is hoped that the likelihood of defect recurrence will be minimized.

7.3. Defect Lifecycle

Learning objectives

(K3) Process a defect following the incident management lifecycle procedure as proposed by IEEE standard 1044-1993.

In [figure 7-1](#), you see a diagram that shows the IEEE 1044 incident management lifecycle, including a mapping from IEEE 1044 that shows how typical incident report states in an incident tracking system would fit into this lifecycle. Let's look at this lifecycle.

We assume that all incidents will follow some sequence of states in their lifecycle, from initial recognition to ultimate disposition. Not all incidents will travel through the exact same sequence of states, as you can see from [figure 7-1](#). The IEEE 1044 defect lifecycle consists of four steps:

Figure 7-1. IEEE 1044, incident management lifecycle

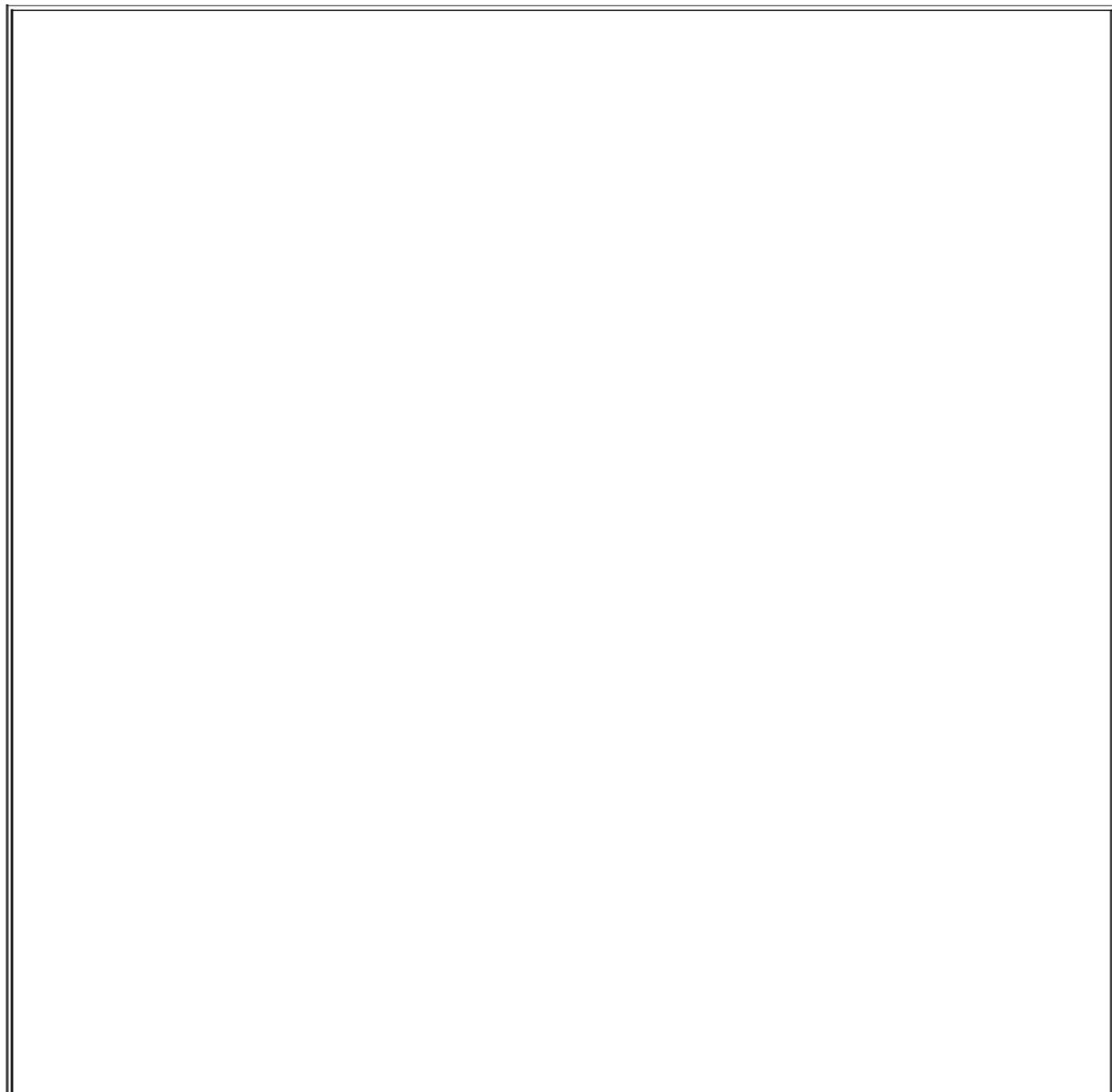


1. **Recognition:** Recognition occurs when we observe an anomaly (that observation being an incident), which indicates a potential defect. Recognition can occur in any phase of the software lifecycle.
2. **Investigation:** After recognition, investigation of the incident occurs. Investigation can reveal related issues and propose solutions. One solution is to conclude that the incident does not arise from an actual defect; e.g., it might be a problem in the test data.
3. **Action:** The results of the investigation trigger the action step. We might decide to resolve the defect. We might want to take action to

prevent future similar defects. If the defect is resolved, regression testing and confirmation testing must occur. Any tests that were blocked by the defect can now progress.

4. Disposition: With action concluded, the incident moves to the disposition step. Here we are principally interested in capturing further information and moving the incident into a terminal state.

Let's look at an example of how this worked on an actual project. In [figure 7-2](#), you see the lifecycle used on a defect tracking system we implemented for the IVR system of systems project. We created this defect tracking application for the client, using Microsoft Access. This project happened before freeware defect tracking systems like Bugzilla became widely available, so this was a cost-effective option.



ISTQB Glossary

anomaly: Any condition that deviates from expectation based on requirements specifications, design documents, user documents, standards, etc., or from someone's perception or experience. Anomalies may be found during, but not limited to, reviewing, testing, analysis, compilation, or use of software products or applicable documentation.

Figure 7-2. Example: Defect lifecycle

If you refer back to [figure 7-1](#), you can see that the states and step mappings are quite similar. There are a few differences. For one thing, we considered Deferred a final disposition. We did not have an Invalid state, but we would close a report as invalid if we decided to permanently reject it. Rejection is also a final disposition.

In addition, we used a single combined Assigned state rather than separate Submit and Build states because we didn't track build time separate from fix time. We used a single combined Test state rather than separate QA and Verified states because my test team was empowered to determine if the problem was fixed.

Finally, we did not have an Archive state because we presumed the reports would remain indefinitely in their final state.

7.3.1. Defect Lifecycle Exercise

Assume that a select group of Telephone Bankers will participate in HELLOCARMS testing as a beta test. The bankers will enter live applications

from customers, but they will also capture the information and enter it into the current system afterward to ensure that no HELLOCARMS defects affect the customers.

The bankers are not trained testers and are unwilling to spend time learning testing fundamentals. So, to avoid having the bankers enter poorly written incident reports and introduce noise into the incident report metrics, management has decided that when a banker finds a problem, they will send an email to a test analyst to enter the report.

A test analyst receives the following email from a banker describing a problem:

I was entering a home equity loan application for a customer with good credit. She owns a high-value house, though the loan amount is not very large.

At the proper screen, HELLOCARMS popped up the Escalate to Senior Telephone Banker message. However, I clicked "continue" and it allowed me to proceed, even though no Senior Telephone Bank Authorization Code had been entered.

From that point forward in this customer's application, everything behaved normally.

I had another customer with a similar application—high-value house, medium-sized loan amount—call in later that day. Again, it would let me proceed without entering the authorization code.

Answer the following three questions:

1. What IEEE 1044 step is this report in when the test analyst receives the report?
2. What actions should the test analyst take to move this report to the next state in the IEEE 1044 lifecycle?
3. What state in the IEEE 1044 lifecycle would you expect the report to be in after these actions?

The answers are shown in the defect lifecycle debrief.

If working in a classroom setting, you may choose to work in small groups of three to five people.

7.3.2. Defect Lifecycle Exercise Debrief

Here are my answers to each of the three questions.

What IEEE 1044 step is this report in when the test analyst receives the report?

It is in the *investigation* step, though it is possible that the Telephone Banker did not gather all the information needed in the *recognition* step.

What actions should the test analyst take to move this report to the nextstate in the IEEE 1044 lifecycle?

First, evaluate each of the pertinent recognition and recognition impact classifications and data fields to see if this email or other information provides that information. If not, obtain that information to finalize the recognition step.

Second, evaluate each of the pertinent investigation and investigation impact classifications and data fields to see what information should be gathered at this stage.

What state in the IEEE 1044 lifecycle would you expect the report to be in after these actions?

It should now be in the *open* state, ready to be *assigned* to a particular developer. Of course, the bug triage committee or change control board might decide that the behavior is correct—an odd decision but within their purview to make—and then make the defect report *closed*. Alternatively, the bug triage committee or change control board might decide that the behavior is incorrect but acceptable for the time being, and then make the defect report *deferred*.

ISTQB Glossary

configuration control board (or change control board or bug triage committee or incident triage committee): A group of people responsible for evaluating and approving or disapproving proposed changes to configuration items and for ensuring implementation of approved changes.

7.4. Defect Fields

Learning objectives

(K3) Evaluate defect reports against IEEE standard 1044-1993 and the applied defect taxonomy in order to improve their quality.

Of course, what's driving the incidents from one state to another, and thus from one step in the lifecycle to another, is what the test analysts, programmers, and other project participants learn about the incident as they move it toward final disposition. States are needed because defects move from one owner to another owner; a defect tracking system must capture that learning. Therefore, within each step—and indeed, embedded in each state—are three information capture activities:

- Recording
- Classifying
- Identifying impact

The way this works is shown in [Table 7-1](#).

During the recognition step, we record supporting data. We classify reports based on important attributes that we have observed. We identify impact based on perceived impact, which might differ from the final impact assessment.

During the investigation step, we update and record more supporting data. We update and add classification information about importance based on attributes uncovered during the investigation. We update the impact based on investigation too.

Table 7-1. IEEE 1044 classification process

Step	Activities		
Record...	Classify...	Identify impact...	
1. Recognition	Include supporting data	Based on important attributes	Based on perceived impact
2. Investigation	Update and add supporting data	Update and add classification on important	Update based on

		attributes	investigation
3. Action	Add data based on action taken	Add data based on the action taken	Update based on action
4. Disposition	Add data based on disposition	Based on disposition	Update based on disposition

During the action step, we record new supporting data based on action taken. We also add classification data based on the action taken. We update the impact based on the action too.

Finally, during the disposition step, we record final data based on disposition. We adjust and finalize the classifications based on the disposition. We capture the final impact assessment.

Notice I've been talking about data and classifications. The IEEE 1044 standard includes mandatory and optional supporting data and classifications for each activity in each step. We'll review these as we continue. By the way, when I say "mandatory supporting data and classifications", I mean mandatory for IEEE 1044 standard compliance.

Each of these data items and classifications is associated with a step or activity. The IEEE 1044 standard assigns a two-character code: RR (recognition), IV (investigation), AC (action), IM (impact identification), and DP (Disposition).

We'll go through these data items and classifications as we continue. As we do so, don't get lost in the trees and fail to see the forest. The important thing to ask yourself is not—usually—"Is my incident management system IEEE 1044 compliant?" but rather, "Might this data or classification be useful to capture?"

The following list shows the recognition step classifications:

- Project Activity (code RR1XX): What were you doing when the incident was observed? This is a mandatory field for IEEE 1044 compliance.
- Project Phase (code RR2XX): What phase was the project in (mandatory)? You will have to tailor the Project Phase to your lifecycle.
- Suspected Cause (code RR3XX): What do you think might be the cause (optional)? I've found that, in many cases, capturing this data can help

the programmers trying to fix the defects, especially if you have very technical testers.

- Repeatability (RR4XX): Could you make the incident happen more than once (optional)? I have a problem with the IEEE calling this information optional as I think that information about reproducibility (or the lack of it) is essential to good incident reports.
- Symptom (RR5XX): How did the incident manifest itself (mandatory)?
- Product Status (RR6XX): What is the usefulness of the product if the incident isn't resolved (optional)? I disagree with the IEEE's optional designation for this one too.

The XX characters at the end of the codes indicate that these are hierarchies. Subclassifications exist within each one. For example, each of the project activity choices has a specific code like RR110, RR120, etc. IEEE 1044 defines choices for these as well, but we're not going to review the standard to that level of detail here.

The following list shows the recognition step data:

- What environment were you working in when you saw the incident? You should capture product hardware, product software, database, test support software, platform, firmware, and other useful information.
- What origination details can you capture? You should capture tester name, the date the incident was observed, the code or functional area, the distribution (which is the version of the test object), and contact information like email address, address, phone number, and company ID.
- At what time did you see the incident? This is operating time (i.e., time since last reboot or total uptime), wall clock time, system time, and CPU time.
- What, if any, vendor information applies? This includes company, contact name, vendor ID, expected resolution, and expected resolution date.

The following list shows the investigation step classifications:

- Actual Cause (IV1XX): What really caused the incident (mandatory)?

- Source (IV2XX): What was the incident's origin (mandatory)? What was the underlying mistake?
- Type (IV3XX): What type of defect caused the failure (mandatory)? This is a question of defect taxonomy.

Remember that you can update classifications from previous steps during this step.

The following list shows the investigation step data:

- What acknowledgement information can we capture? What data was received, what report number was assigned, who is the investigator, when are estimated start and complete dates of the investigation, when (subsequently) did the actual start and completedates of the investigation occur, how many person hours were spent, on what date did we receive this acknowledgement, and what documents were used in the investigation?
- What verification information can we capture? What was the source of an anomaly (or incident) and how did we verify the data from the recognition process?

Remember that you can update data from previous steps during this step.

The following list shows the action step classifications:

- Resolution (AC1XX): When and how should the incident be resolved (mandatory)?
- Corrective Action (AC2XX): What should be done to prevent such incidents in the future (optional)?

Remember that you can update classifications from previous steps during this step.

The following list shows the action step data:

- What resolution identification information can we capture? What test itemis to be fixed, what specific component within the item is to be fixed, how can we describe (in text) the fix, when is the planned date for action com-pletion, who is the person assigned, what is the planned date of fix comple-tion, or, if the fix is deferred, where is our reference or authority for that?

- What resolution action information can we capture? What is the date on which it was completed, which organization is assigned to verify resolution, and which person is assigned to verify resolution?

Remember that you can update data from previous steps during this step.

The following list shows the disposition step classifications and supporting data:

- Disposition (DP1XX): How was the problem finally resolved (mandatory)?
- What anomaly (or incident) disposition information should we capture? What action was implemented, on what date was the report closed, on what date was document updating complete, when was the customer notified, and what reference document numbers might exist?
- What verification information should we capture: What is the name of the person doing the verification, on what date did the verification occur, what version and revision levels were verified, what method did we use to verify, and what is the test case we used to verify?

Remember that you can update classifications and data from previous steps during this step.

Now, throughout the lifecycle, impact classifications are made and revised. Let's look at some of those impact classifications:

- Severity (IM1XX): What is the impact on the system (mandatory)?
- Priority (IM2XX): What is the relative importance of the incident (optional)? I disagree with this being optional. In fact, I'd say it's more

important than severity in many cases.

- Customer Value (IM3XX): How does this incident affect customer(s) or market value (optional)? Again, this strikes me as essential information.
- Mission Safety (IM4XX): How does this affect mission objectives or safety(optional)? This would only apply to certain systems, of course.
- Project Schedule (IM5XX): How will resolving this incident affect the project schedule (mandatory)?
- Project Cost (IM6XX): How will resolving this incident affect the projectcost (mandatory)?
- Project Risk (IM7XX): What is the project risk associated with fixing this incident (optional)? I think this is like another classification that should be mandatory.
- Project Quality/Reliability (IM8XX): What is the project quality/reliability impact associated with fixing this incident (optional)? Yet another important variable.
- Societal (IM9XX): What are the societal issues associated with fixing this incident (optional)? This would only apply to certain systems.

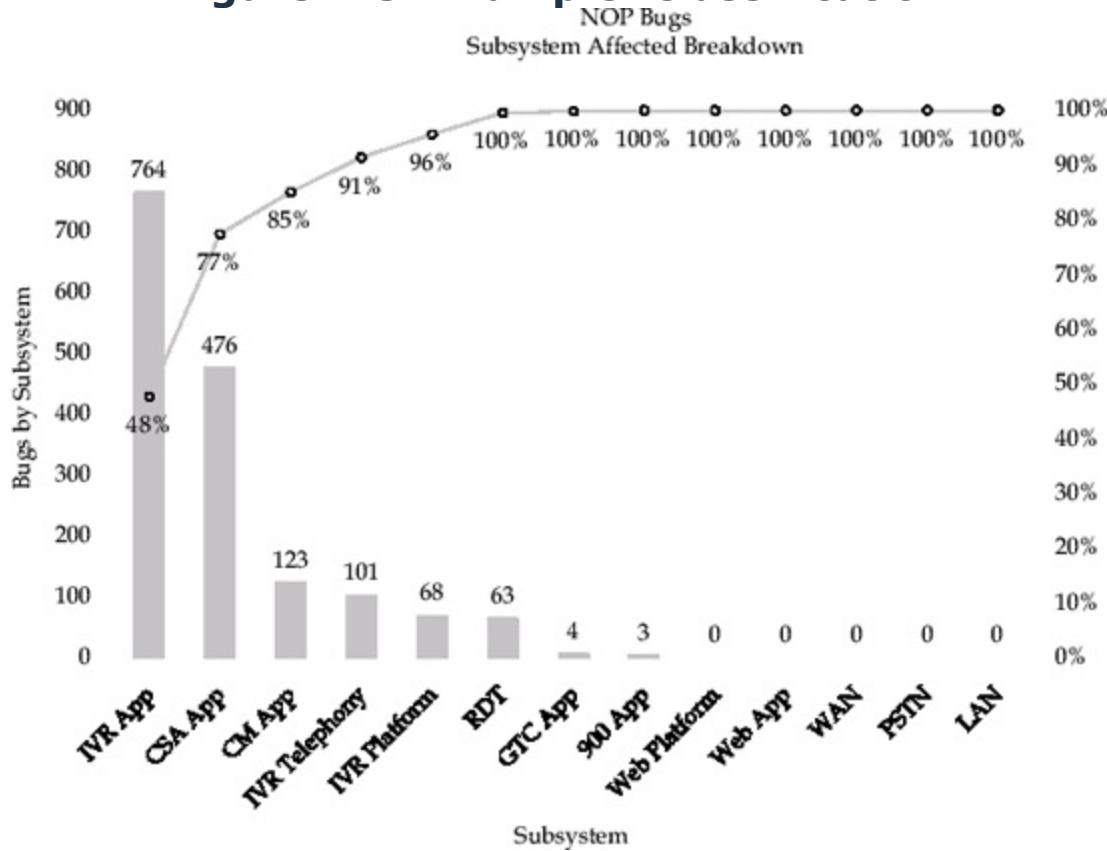
The following list shows the impact data:

- What is the cost impact of this incident? That includes cost to analyze, estimated cost if the fix is done, estimated cost if the fix is not done, and other costsof resolution.
- What is the time impact of this incident? That includes estimated time required if the fix is done, estimated verification time if the fix is done, estimated timeif the fix is not done, and actual implementation time.
- What is the risk of this incident? This is a text description.
- What is the schedule impact? This includes assuming that the incident is resolved, assuming that it's not resolved, and if it is resolved, what the actual schedule impact was.
- What is the contract change, if any?

figure 7-3 shows an example of using classification information to learn

something interesting about a project. This Pareto chart analyzes the number and percentage of bugs associated with each major system in the IVR system of systems project. This project, called the NOP project, tied together 10 systems via a wide area network, a local area network, and the phone system to implement a large distributed entertainment application.

Figure 7-3. Example: Classification



As you can see in [figure 7-3](#), the interactive voice response (or IVR) application accounts for about half of the bugs. The customer service application (or CSA) adds about 30 percent more. The rest of the applications are relatively solid. The content management (or CM) application is less than 10 percent of the bugs. The interactive voice response server's telephony and OS/hardware layers each are around 5 percent, with the remaining applications and infrastructure accounting for the other 4 percent.

This figure tells us that the technical risk—the likelihood of finding bugs—is highest for the IVR application. If we wanted to use this information to reallocate the test effort, we could update our risk analysis based on this information and then see how that updated risk analysis would affect our testing. If we wanted to use this for a project retrospective, then we might do further analysis to see why so many bugs were in that application. What

changes can we make in subsequent projects to reduce the incidence of bugs in the IVR application?

7.4.1. Defect Fields Exercise

This exercise is based on the same scenario as the previous exercise. To review, assume that a select group of Telephone Bankers will participate in HELLOCARMS testing as a beta test. The bankers will enter live applications from customers, but they will also capture the information and enter it into the current system afterward to ensure that no HELLOCARMS defects affect the customers.

The bankers are not trained testers and are unwilling to spend time learning testing fundamentals. So, to avoid having the bankers enter poorly written incident reports and introduce noise into the incident report metrics, management has decided that, when a banker finds a problem, they will send an email to a test analyst to enter the report.

A test analyst receives the following email from a banker describing a problem:

I was entering a home equity loan application for a customer with good credit. She owns a high-value house, though the loan amount is not very large.

At the proper screen, HELLOCARMS popped up the Escalate to Senior Telephone Banker message. However, I clicked "continue" and it allowed me to proceed, even though no Senior Telephone Bank Authorization Code had been entered.

From that point forward in this customer's application, everything behaved normally.

I had another customer with a similar application—high-value house, medium-sized loan amount—call in later that day. Again, it would let me proceed without entering the authorization code.

Answer the following three questions:

- What recognition classifications and data were provided directly or indirectly in the email from the banker?
- What recognition impact classifications and data were provided directly

or indirectly in the email from the banker?

- What information was missing and what further investigation is needed?

The answers are shown in the Defect Fields debrief.

If working in a classroom setting, you may choose to work in small groups of three to five people.

7.4.2. Defect Fields Exercise Debrief

First, I evaluated each of the pertinent recognition and recognition impact classifications and data fields to see if this email or other information I can assume that I have is presented. My analysis is shown in [Table 7-2](#).

Next, I have annotated the report below with some additional information I would want the test analyst to gather before putting it into the system. The original information is shown in italic font, while my text is shown in regular font.

I was entering a home equity loan application for a customer with good credit.

I would want to find out her exact data, including income, debts, assets, credit score, etc.

She owns a high-value house, though the loan amount is not very large.

I would want to find out the exact value of the house and the loan amount.

I would test various combinations of values and loan amounts to see if I could find a pattern.

At the proper screen, HELLOCARMS popped up the Escalate to Senior Telephone Banker message. However, I clicked "continue" and it allowed me to proceed, even though no Senior Telephone Bank Authorization Code had been entered.

I would want to find out if the banker entered anything at all into that field.

I would test by leaving it empty, inputting blanks, inputting valid characters that were not valid authorization codes, and doing some other checks to see whether it is ignoring the field completely.

From that point forward in this customer's application, everything behaved normally.

Table 7-2. Incident report IEEE 1044 coverage

IEEE Information	Available?
Project Activity	Presumably we know this for all such beta tests.
Project Phase	Presumably we know this for all such beta tests.
Suspected Cause	Not available.
Repeatability	Available, but more isolation and replication of this issue is needed.
Symptom	Available.
Product Status	Not available, but we can presume that it's unacceptable for the product to allow the Telephone Bankers to bypass a risk management policy like this.
Environment	Presumably we know this for all such beta tests.
Originator	Presumably provided in the sender information for the email.
Time	Not available.
Vendor	Some of the vendor information we can presume to know, while the other information, such as about when they will supply a fix, is not applicable at this point.
Severity	Available.
Priority	Not available, but again we can presume this is a high priority.
Customer Value	Not available, but inferable.
Mission Safety	Not applicable.
Project Schedule	Not applicable at this point, as investigation is required.
Project Cost	Not applicable at this point, as investigation is required.
Project Risk	Not applicable at this point, as investigation is required.
Project Quality/	Not applicable at this point, as investigation is required.

Reliability	
Societal	Not applicable at this point, as investigation is required.
Cost	Not applicable at this point, as investigation is required.
Time	Not applicable at this point, as investigation is required.
Risk	Not available, but again we can make some inferences and describe the risk associated with letting Telephone Bankers bypass bank risk management policies.
Schedule	Not applicable at this point, as investigation is required.
Contract Change	Not applicable at this point, as investigation is required.

I would test to see whether such applications are transferred to LoDoPS or are silently discarded. If they are transferred to LoDoPS, does LoDoPS proceed or does it catch the fact that this step was missed?

I had another customer with a similar application—high-value house, medium-sized loan amount—call in later that day. Again, it would let me proceed without entering the authorization code.

Here also I would want to find out the exact details on this applicant, the property value, and the loan amount.

7.5. Metrics and Incident Management

Learning objectives

(K4) Analyze the defect reports created over time and update the defect taxonomy.

The use of the textual descriptive information in the incident reports is usually obvious to test analysts, but it's easy to get confused about the use of the classifications. I had one client tell me about spending hundreds of thousands of dollars on consulting to improve their incident tracking system to use the latest in classification schemes, orthogonal defect classification. However, they tried to save money on the project by not training people in how to use the fields, so all of the classification information was worthless!

You should look at incident classification information from the immediate point of view of the project and from the long-term point of view of organizational and process improvement. From the project point of view, incident classifications should support test progress monitoring as discussed in the Foundation syllabus and in earlier chapters of this course. We can use various metrics like bug cluster analysis, defect density analysis, and convergence (also called open/closed charts) during a project to manage defect trends and check readiness for release.

From the organization and process point of view, we want to assess how we're doing and figure out how to do better. Incident classifications should support process improvement initiatives. We should be able to assess phase containment, which is the percentage of defects that are detected and removed in the same phase they were introduced. We should be able to assess root causes so we can reduce the total number of defects. And, we should be able to assess defect trends across projects to see where best—and worst—practices exist.

[figure 7-4](#) shows an example of the closure period chart for the Internet appliance project I've mentioned a number of times. Closure period charts analyze defect trends over time. The chart in [figure 7-4](#) shows something interesting. During the first 12 weeks—from August 1 to about October 17—the rolling closure period gradually trended upward. After a brief unstable period right at the start, it settled to around 11 days and over a two-month period moved to about 14 days.

Figure 7-4. Example: Defect closure over time

However, on October 17, some discontinuity must have occurred. In the month from October 17 to November 17, the rolling closure period moved up another 3 days to around 17 days. Previously in the project it had taken two months to creep up by 3 days, but now it has jumped up 3 days in just one month, a rate twice as fast as before. Remember, too, that the rolling closure period becomes harder to pull up or down as the project continues because the set of bug reports already closed becomes larger and thus the rolling closure period is "pinned in place" by the past numbers.

You can see there were over 20 daily closure periods above the rolling closure period during that month, while only 4 daily closure periods were below the rolling closure period. In addition, the daily closure periods that were above the rolling closure period were more significantly above the rolling closure period than the daily closure periods that were below the rolling closure period were below it. Thus, you can think of the daily closure period as acting to sharply "pull up" the rolling closure period.

What happened? It's not possible to say for sure, based on a trend chart. Trend charts show trends in time, not causality. To find the answer in such a situation, I would start with two avenues of investigation. First, I would check to see if a major project milestone occurred on October 17. That could be entry into a new level of testing, especially a more formal level of testing that would tend to have a slower-moving process. Alternatively, the introduction of a large chunk of new functionality could lead to tricky problems being discovered. Or, perhaps some members of the development team were redeployed to another project.

The second avenue lies in the bug taxonomy. I would run three analyses:

- Percentage distribution of bugs by affected subsystem for the entire project
- Percentage distribution of bugs by affected subsystem for the period from August 1 to October 17
- Percentage distribution of bugs by affected subsystem for the period from October 17 to November 17

I showed this type of graph in the chapter on test management, when discussing metrics for test progress monitoring and control.

This analysis might reveal a change in where the bugs are. That in turn would lead to the next round of investigation. Finally, we might adjust the defect taxonomy, our risk analysis, and our plan for the rest of the testing based on this information.

7.5.1. Metrics and Incident Management Exercise

This exercise is based on the same scenario as the previous exercise. To review, assume that a select group of Telephone Bankers will participate in HELLOCARMS testing as a beta test. The bankers will enter live applications from customers, but they will also capture the information and enter it into the current system afterward to ensure that no HELLOCARMS defects affect the customers.

The bankers are not trained testers and are unwilling to spend time learning testing fundamentals. So, to avoid having the bankers enter poorly written incident reports and introduce noise into the incident report metrics, management has decided that, when a banker finds a problem, they will send an email to a test analyst to enter the report.

A test analyst receives the following email from a banker describing a problem:

I was entering a home equity loan application for a customer with good credit. She owns a high-value house, though the loan amount is not very large.

At the proper screen, HELLOCARMS popped up the Escalate to Senior Telephone Banker message. However, I clicked "continue" and it allowed me

to proceed, even though no Senior Telephone Bank Authorization Code had been entered.

From that point forward in this customer's application, everything behaved normally.

I had another customer with a similar application—high-value house, medium-sized loan amount—call in later that day. Again, it would let me proceed without entering the authorization code.

Now, for this exercise, assume that you see a few emails similar to the one shown here being sent to test analysts. How might you update the IEEE 1044 classifications to capture information that helps to analyze the prevalence of this type of problem?

7.5.2. Metrics and Incident Management Exercise Debrief

To understand how prevalent this problem is, a Pareto analysis comparing this type of failure to other types of failures would be very helpful. So, you could update the Symptom (RR5XX) classification. This classification asks how the incident manifested itself. By including this specific kind of failure, not escalating loans that require Senior Banker approval, you could then run an analysis later to compare the incidence of this type of failure to others and take appropriate action.

7.6. Communicating Incidents

Learning objectives

Recall of content only

Bad incident reports are a major cause of friction and poor relationships in project teams. My associates and I see it all the time during test team assessments. To maintain good relations in the team, keep the following in mind:

- It is usually not the tester's job to apportion blame or affix fault. Avoid any statements that could be construed as accusations or implying blame. Avoid comments that someone could take personally.
- A good incident report should provide objective information. Stick to the facts. If you are going to make an assumption or state a theory, state your reasons for doing so. If you do decide to make such assumptions or theories, be sure to remember the first rule about not getting personal.
- An incident report is usually an assertion that something is wrong. When you are saying that a problem exists, it helps to be right. Strive for utmost accuracy.
- Finally—and this is more of a mindset but it's really an important one—start to see incident reports as a service you provide, not just to managers but also to developers. Ask developers what information you can include in your reports to help them out. You'd be surprised what a difference this can make.

Some testers get frustrated when "their" bugs don't get fixed. When we see that during an assessment, my first thought is that something is broken in the incident management process. Ideally, a bug triage or incident triage meeting involving a cross-functional group of stakeholders occurs to conduct prioritization of incidents. It's seldom good to rely on just developer or tester opinions about what should be fixed or deferred. That's not to say that developers' and testers' opinions and input doesn't count, but rather that good incident management requires careful consideration of the options for handling an incident. Few projects have the luxury of fixing every single incident that comes along.

In summary, good communication and relations within the team, good defect tracking tools, and good defect triage are all important for a good incident management process. Incident management is a testing fundamental that all test analysts and all test managers should master.

7.7. Sample Exam Questions

To end each chapter, you can try one or more sample exam questions to reinforce your knowledge and understanding of the material and to prepare for the ISTQB Advanced LevelTest Manager exam.

Q: Assume you are a test manager working on a project to create a programmable thermostat for home use to control central heating, ventilation, and air conditioning (HVAC) systems. In addition to the normal HVAC control functions, the thermostat has the ability to download data to a browser-based application that runs on PCs for further analysis.

During quality risk analysis, you identify compatibility problems between the browser-based application and the different PC configurations that can host that application as a quality risk item with a high level of likelihood.

Your test team is currently executing compatibility tests. Consider the following excerpt from the failure description of a compatibility bug report:

1. Connect the thermostat to a Windows Vista PC.
2. Start the thermostat analysis application on the PC. Application starts normally and recognizes connected thermostat.
3. Attempt to download the data from the thermostat.
4. Data does not download.
5. Attempt to download the data three times. Data will not download.

Based on this information alone, which of the following is a problem that exists with this bug report?

1. Lack of structured testing
2. Inadequate classification information
3. Insufficient isolation

4. Poorly documented steps to reproduce

Q: Continue with the previous scenario. Your test team is still executing compatibility tests. Consider the following excerpt from the failure description of a compatibility bug report:

1. Install the thermostat analysis application on a Windows XP PC.
2. Attempt to start the thermostat analysis application.
3. Thermostat analysis application does not start.
4. Reinstall the thermostat analysis application three times. Thermo—stat analysis application does not start after any reinstallation.
5. This test passed on the previous test release.

Based on this information alone, which of the following is the most reasonable hypothesis about this bug?

1. The bug might be a regression.
2. The bug might be intermittent.
3. The application didn't install on Windows XP PCs before.
4. The bug might be a duplicate.

Chapter 8. Standards and Test Process Improvement

The United States standard railroad gauge of 4 ft 8 1/2 in derives from the original military specification (MilSpec) for an Imperial Roman army war chariot. MilSpecs...live forever!

—Tom O'Hare, Professor of Germanic Languages, University of Texas at Austin, posting a possibly apocryphal story at www.spikesys.com/Trains/st_gauge.html.

The eighth chapter of the Advanced syllabus is concerned with standards and test process improvement. As such, it is not particularly coherent, given the sharp division between the first half on standards and the second half on test process improvement. The material on standards introduces a number of international, national, and domain-specific standards. The material on test process improvements covers the generic concept of process improvement and then addresses four models for test process improvement and one model for software process improvement. chapter 8 of the Advanced syllabus has nine sections:

- 1.** Introduction
- 2.** Standards Considerations
- 3.** Test Improvement Process
- 4.** Improving the Test Process
- 5.** Improving the Test Process with TMM
- 6.** Improving the Test Process with TPI
- 7.** Improving the Test Process with CTP
- 8.** Improving the Test Process with STEP
- 9.** Capability Maturity Model Integration, CMMI

Let's look at each section and how it relates to test management.

8.1. Introduction

Learning objectives

Recall of content only

Managing a test project, an ongoing test effort, and a test team is challenging. Fortunately, there are various sources of inspiration, ideas, and support for the job.

For one thing, various standards can apply to and affect testing. Standards can vary based on industry, on safety-criticality, and by country. In some cases, they are mandatory, either by law or by the practical realities of the market. To some extent, you might view mandatory testing standards as a burden, but at least a mandatory testing standard removes some of the guesswork from your job. Also, it's my observation as a consultant with some clients that are subject to mandatory standards and some that are not, that test teams that are subject to mandatory standards are often better funded and better supported by management than test teams that are not.

If you are not subject to mandatory testing standards, those testing standards can still serve as a source of useful and interesting ideas about how to test. Be sure to consider carefully why a particular idea is in the standard before adopting it, though, as some ideas might not fit your situation.

In addition to standards, there are techniques for improving your test process. Like any business process, the test process should be improved. Even a good test process will become bad over time, not necessarily because of disuse by the test team but because the organization will eventually have new and different testing needs.

When you embark on a test process improvement endeavor, you can use general process improvement techniques. In addition, there are specific test assessment and maturity models available to you.

Some see these models, standards, and maturity measures as onerous, hidebound, distracting, or not applicable. Certainly, consultants, training companies, and other entities have sold some standards snake oil, but simply because there are a few shameless opportunists at work doesn't mean there aren't good ideas in these standards. You can derive benefit

from models, standards, and maturity measures by carefully evaluating the role they can play in your organization.

In fact, those who reject all models, standards, and maturity measures are often engaging in their own self-deception, as these rebuffs are often a form of "not invented here" syndrome, where someone insists that their situation is so different and unique that ideas from outside won't work. As a consultant who has visited hundreds of sites around the world to lecture, consult, and work on projects, I can tell you that people, projects, and products aren't as different as some like to think.^[5]

[5] Kaner, et al., in their book *Lessons Learned in Software Testing*, tend to take a skeptical view of standards, though ironically enough some of the authors have served on standards committees.

So, in this chapter, we'll look at standards, models, and maturity measures that can help you as a test manager.

8.2. Standards Considerations

Learning objectives

(K2) Summarize sources of software standards and explain each source's usefulness for software testing.

Let's start with a quick overview of standards. Standards can cover a variety of topics of direct or indirect interest to us as test managers:

- Software development lifecycles
- Software testing and methodologies
- Software configuration management
- Software maintenance
- Quality assurance
- Project management
- Requirements
- Software languages
- Software interfaces
- Defect management

These standards come from a variety of sources. There are international standards developed by international standards committees. There are national standards, developed by national standards committees or adapted by national standards committees from international standards. And, there are domain specific standards, where an international or national standard has been adapted to a particular domain or developed specifically for that domain.

It's important to select and adapt standards appropriately. So be sure to understand the history and source of the standards you are considering as well as the proper application of the standard. Here are some general aspects to keep in mind about standards.

As mentioned before, standards come from various international, national, and domain-specific standards bodies. Ultimately, a standards body is a group of recognized professionals. Unlike physical laws, such as gravity, standards are created and assembled by these professionals based on their collective wisdom, experience, empirical evidence, preferences, and biases. So, a standard's credibility and value ultimately should arise from the people who create it. Not all mandatory standards are necessarily valuable, and some voluntary standards are valuable.

Being developed by people, standards evolve and change over time. Thus, when evaluating a standard, make sure you know which version you're evaluating. If the standard is under revision, you should at least review the information available on that revision as part of your evaluation. Better yet, if you can get draft copies, review those.

Will standards prove useful to you? Well, if you are subject to a mandatory testing standard, it will prove quite useful, probably in at least three important ways. First, you can't do your job properly without referring to the standard, so it's your handbook for testing in a way. Second, if you think back to our earlier discussion on test management documentation in [chapter 3](#), you will probably find that the standard serves as a source and valuable reference for your test policy and test strategy documents. Third, as I noted earlier, you might find that the standard helps create support for testing and unlocks funding you might not otherwise get. After all, how can management question a particular testing activity if it's mandated by a standard to which you must comply?

Even when standards are not mandatory, they can still prove useful. For one thing, standards can promote a preventive testing strategy. For another, they can provide adaptable process checklists, ideas on test techniques, potential objectives for testing, and other types of reference frameworks. In a nonmandatory setting, you can use a standard as a starting point for establishing structure in your testing.

It is also true that standards can hinder you. As I mentioned in my earlier discussion about types of test strategies, a standards-driven test strategy has the weakness that, if the standard was derived for situations vastly different than yours, you'll find it a poor fit. You'll end up not doing things you should do and doing things you really don't need to do. You can't afford to make either of these two mistakes—the first dulls your effectiveness and the second dulls your efficiency—but you will make these mistakes if you fail to apply or to tailor standards properly.

Another potential pitfall with standards arises when you are subject to or are adapting multiple standards. Just as laws are developed by people and thus vary from country to country and even from town to town, so too do standards. In some cases, they are simply inconsistent, but in other cases, they are conflicting. In a situation where two or more standards apply, watch for such conflicts and incoherence and, if they arise, adapt the standards to resolve these problems.

Important international standards bodies include those organized by ISO and IEEE. Until recently, ISO stood for International Standards Organization, but it recently changed its name to International Organization for Standardization. Analogous to the structure of the ISTQB, it is composed of representatives of national standards boards. It promulgates the following standards of interest to testers:

- ISO 9126 covers software engineering aspects of product quality. It consists of four standards and technical reports. ISO 9126-1 addresses the quality model, including the characteristics and subcharacteristics covered earlier. ISO 9126-2 addresses external quality metrics, such as those measurable during dynamic testing. ISO 9126-3 addresses internal quality metrics, such as those measurable during static testing. ISO 9126-4 addresses quality in use metrics, such as those measurable during field or production.
- ISO 12207 covers software lifecycle processes. This standard was addressed in the Foundation syllabus.
- ISO 15504 covers process assessment and is sometimes called SPICE. It is similar to Capability Maturity Model (CMM) and Capability Maturity Model Integration (CMMI).

IEEE is the Institute of Electrical and Electronics Engineering. It is a professional organization based in the United States. However, it has national representatives from over 100 countries. It promulgates the following standards of interest to testers:

- IEEE 610 is the IEEE standard computer dictionary. It compiles definitions from across the standards. Note that conflicts exist between the IEEE 610 glossary and the ISTQB glossary.
- IEEE 829 is the standard for software test documentation, which was covered extensively in the Foundation and reviewed in this book.

- IEEE 1028 is the standard for software reviews, covered at the Foundation level.
- IEEE 1044 is the guide to classification for software anomalies, which was covered in the previous chapter.

For both ISO and IEEE, additional standards exist that might prove useful to you.

National standards, as you might expect, vary from country to country but can have international applicability. As an example of this, consider the UK's British Standard 7925. It covers a number of test design techniques, including these:

- Equivalence partitioning
- Boundary value analysis
- State transition testing
- Cause-effect graphing
- Syntax testing
- Statement testing
- Branch/decision testing
- Data flow testing
- Branch condition testing
- Branch condition combination testing
- Modified condition decision testing
- Linear Code Sequence and Jump (LCSAJ) testing
- Random testing
- Component test process

These topics are covered in detail in the companion volumes on test analysis.

Other standards are business-domain or technical-domain specific, and some address testing and quality. For example, in the avionics industry, we find the DO-178B standard, which in Europe is called the ED 12B standard. It is officially named "Software Considerations in Airborne Systems and Equipment Certification". It applies to civilian aircraft, but I've also seen it used on military systems. It covers software issues, including embedded systems in aircraft and software used to test the embedded systems in aircraft. This standard was covered in the earlier chapter on test management.

In the space industry, we find the standard provided by the European Cooperation on Space Standardization. As with the FAA 178B standard, it recommends methods and techniques based on the criticality of the software. These techniques can include the following:

- SFMECA—Software Failure Modes, Effects and Criticality Analysis
- SFTA—Software Fault Tree Analysis
- HSIA—Hardware Software Interaction Analysis
- SCCFA—Software Common Cause Failure Analysis

For medical systems, we can look at the United States Food and Drug Administration (or FDA) standard Title 21 CFR Part 820.

FDA Title 21 CFR Part 820 lays out a set of testing principles, including the following:

- The expected test result should be predefined. There must be a test oracle.
- Testing is a search for defects, not just a trip down the happy path to build confidence. Therefore, the standard includes the principles that a good test case has a high probability of exposing an error, that a successful test is one that finds an error, and that examining only the usual, typical, correct case is insufficient.
- To deal with the author-bias and possible vendor problems, the standard calls for tester independence from coding.
- The standard recognizes the multi-disciplinary nature of testing skills, calling for both application and software expertise.

- The standard calls for the use of tools and recognizes that testers use different tools than programmers.
- Finally, the standard calls for test documentation that permits its reuse. It also calls for possible auditing and independent confirmation of the test status, either periodically or in response to some issues in the field.

So, the principles just mentioned would help us define a test policy. However, what specific tasks should we address in our test plans if we are subject to FDA Title 21 CFR Part 820? The standard calls for the following:

- Test planning, covering the topics addressed in [chapter 3](#).
- Structural test design, including statement, branch, condition/multi-condition, loop, path, and dataflow. I cover these techniques in the companion volume on technical test analysis.
- Functional test design, including the normal cases, all outputs, robustness, and input combinations. I cover techniques for these types of tests in the companion volume on test analysis.
- Traceability between unit tests and detailed design, integration tests and high-level design, and system tests and requirements.
- A number of test levels, including unit test, integration test, functional test, system test, and acceptance test.
- Bug triage that includes the proper set of stakeholders.
- Results evaluation and final report.

There are also other standards—indeed, many other standards—that can apply. Some might be company specific. In some cases, a company or a group might adapt a broader industry standard. Whatever standards might apply, as a test manager, you have to ensure appropriate compliance with a standard.

8.3. Test Improvement Process

Learning objectives

Recall of content only

One of the main themes in the ISTQB Foundation and Advanced syllabi is that testing can improve software. It does so both via prevention when testers are involved early in the project and via detection of important defects during test execution. However, testing is not the only route to improving software. Better software processes help. Because test processes themselves are software processes, the test processes can also be improved. These improvements should allow testing to improve software more effectively and efficiently than before.

Now, since you're reading a book on test management, unless you've been working under a rock for your entire career, you've probably heard of standard software process improvement models like CMMI. If you study those models—as I have—you've probably realized that they don't really talk much about testing. These gaps in the standard software process improvement models have inspired various people, including me, to create test process improvement models.

In the next few sections, we'll cover the four main test process improvement models:

- Critical Testing Processes (CTP)
- Systematic Test and Evaluation Process (STEP)
- Test Maturity Model (TMM)
- Test Process Improvement (TPI)

I've listed these in alphabetical order, not necessarily in order of likely occurrence. Each of the four books that describe these models remain in print and sell reasonably well, so they are in common and current application in a large number of organizations.^[1]

[1] My own book *Critical Testing Processes* describes the Critical Testing Processes model. Rick Craig's book, *Systematic Software Testing*, describes the Systematic Test and Evaluation process. Ilene Burnstein,

et al.'s book, *Practical Software Testing*, describes the Testing Maturity Model. Martin Pol and Tim Koomen's book, *Test Process Improvement*, describes the Test Process Improvement model.

We won't cover the following additional four test process models, but I'll mention them here in case you're interested in doing some more research on them.

- Test Organization Maturity (TOM)
- Test Improvement Model (TIM)
- Software Quality Rank (SQR)
- TMap

The first three in this list are unusual. I've never encountered anyone using them anywhere in the world, in any industry. TMap is in wider use, but it is a semi-proprietary model promulgated by European-based consultancy Sogeti. That said, concepts in TMap have cross-pollinated into the ISTQB program.^[1]

[1] TMap is described in Tim Koomen, et al.'s, book *T-Map Next*.

As I mentioned, process improvement can and should apply to the entire software process, including testing. The basic idea of process improvement is that we learn from our mistakes. We can discover and subsequently exploit opportunities to do better next time every time we do something, if we look for those opportunities.

The quality control expert Shewhart put a process in place for process improvement, which is shown in [figure 8-1](#). W.E. Deming popularized this approach, so it is referred to variously as the Deming cycle or the Shewhart cycle.^[2]

[2] For more on Deming's idea on quality management, see Mary Walton's *The Deming Management Method*.

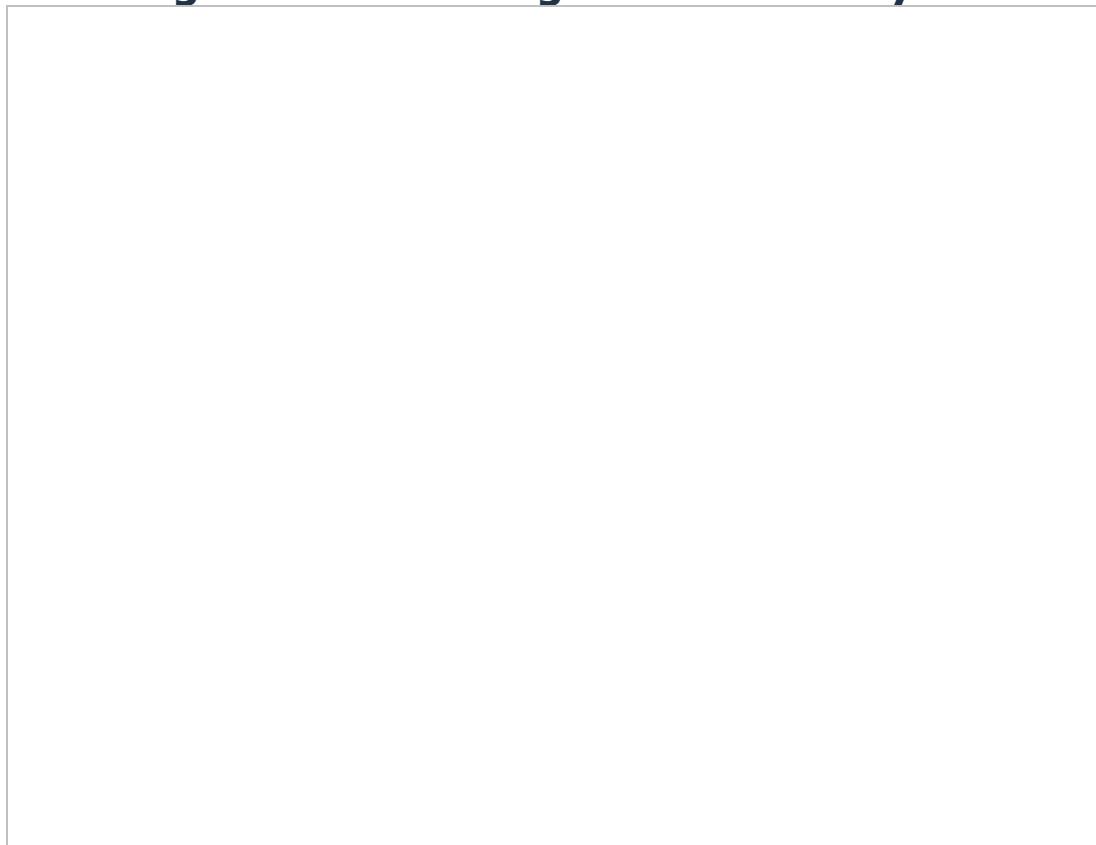
The basic idea is that process improvement consists of four sequential but iterative steps:

- Plan. Think about how to do a process better.
- Do. Carry out the process.

- Check. Evaluate the improvements.
- Act. Take action based on the evaluation, which might lead to further planning for the next iteration.

The fundamental idea of process improvement is that the quality of the product is highly influenced by the quality of the process used to produce it. Rather than focusing entirely on test execution at the end of the process to reveal defects that are then removed—or worse yet, subjecting our users or customers to defects—we can create better software to begin with by using better processes. This will provide the business benefits of better quality that we discussed in [chapter 3](#), but it will start even earlier in the project and provide those benefits at a lower cost.

Figure 8-1. Deming or shewhart cycle



Now, Deming's cycle provides us with a framework for assessing and improving a process with reference to itself and its context. In other words, we can start with whatever processes we have, apply plan-do-check-act to one or all of them, and, over time, get very good.

However, this approach provides incremental improvement in most cases. In addition, you can only improve a process you have, so if you are not doing something you should be doing, this approach has trouble detecting

that. Further, while you know that Deming's cycle will lead to you doing better over time, at any given time how do you know if you're doing well or poorly?

So, process models can provide a starting point, a standard framework, and a way of measuring your processes. We can perform a process assessment using a process model and identify opportunities to improve your current process.

In some cases, assessment using the process model will result in the assignment of some maturity rating to your current process. Because the maturity ratings are one-dimensional—or, in some cases, one-dimensional on a process-by-process basis—I generally refer to these as prescriptive models. I call them prescriptive because they prescribe the order in which specific processes should be improved. The syllabus refers to these as process reference models.

ISTQB Glossary

Test Maturity Model (TMM): A five-level staged framework for test process improvement, related to the Capability Maturity Model (CMM), that describes the key elements of an effective test process.

Test Maturity Model Integration (TMMi): A five-level staged framework for test process improvement, related to the Capability Maturity Model Integration (CMMI), that describes the key elements of an effective test process.

Test Process Improvement (TPI): A continuous framework for test process improvement that describes the key elements of an effective test process, especially targeted at system testing and acceptance testing.

My problem with prescriptive models is that, as a consultant, I believe that businesses should make improvements based on the business value of the improvement and the organizational pain that improvement will alleviate. A one-dimensional maturity rating might lead a business to start making improvements in parts of the overall software process or test process that are actually less problematic or less important than other parts of the process simply because the model listed them in order.

I prefer nonprescriptive process models. The process model describes the important software processes and what should happen in them, but it doesn't put them in any order of improvement.

Now, that doesn't mean that assessment via a nonprescriptive process model doesn't result in recommendations and an order in which to implement them. When my associates and I do an assessment for clients, our report definitely includes both. The key is that, if you look at any two of our assessment reports, you might see very similar recommendations but in very different order. Why? Because each client has a different level of opportunity associated with the recommendation. In some cases, constraints or preconditions can influence the order. Those constraints and levels of opportunity tend to be unique from one organization to another, and a nonprescriptive model adapts to those unique organizational needs.

By the way, the syllabus refers to nonprescriptive models such as these content models.

Whichever approach you use, don't use process assessments as a one-time activity. Having done an assessment and found opportunities to improve, you should—of course—improve. At some point in the future, you should reassess to see the effect of the changes and of course correct your process improvement; this is simply the application of the Deming cycle to the chosen process model.

8.4. Improving the Test Process

Learning objectives

(K3) Write and a test improvement plan using the generic steps involving the right persons.

You can think of test process improvement models as part of an overall trend toward greater maturity and professionalism of testing. By adopting standard models, organizations can generate comparable metrics and measures. In other words, they can compare across projects within the company, across test groups within the company, and with other test groups in other companies that used the same models.

As I mentioned previously, there are two main types of models. The first are the staged or prescriptive models. These usually result in a maturity measure or perhaps a collection of maturity numbers, one for each main process area. Though I am skeptical of the meaningfulness of these maturity numbers to sequence process improvement efforts, they do allow for comparison across companies and organizations, assuming that the assessors measured the maturity numbers consistently across those organizations.

The second type of models, the continuous or nonprescriptive models, do not specify the sequencing of improvement. They allow the organization to focus on the highest-priority issues, based on their own business needs.

There are four test process improvement standard models covered in this course, as I mentioned.

- Critical Testing Processes (CTP)
- Systematic Test and Evaluation Process (STEP)
- Test Maturity Model (TMM)
- Test Process Improvement (TPI)

The first two, Critical Testing Processes (CTP) and Systematic Test and Evaluation Process (STEP), are nonprescriptive. The second two, Test Maturity Model (TMM) and Test Process Improvement (TPI), are prescriptive.

All four allow assessment of the testing. However, once an assessment is performed, TMM and TPI give a prescriptive sequence of process improvements. Making these improvements in this order will cause your test process to move steadily and incrementally up the maturity model they provide. Interestingly, these maturity models differ in the order in which they suggest changes. Conversely, STEP and CTP leave it up to the organization to determine where its greatest return on investment will occur, based on a particular set of process improvements in a particular order.

In one way, the prescriptive models are easier to use. The assessment is generally simpler because it does not require the assessor to build a business case for the sequence of recommended changes associated with each process area. The assessor using the nonprescriptive models must create such a business case—to some meaningful degree of accuracy and precision—in order to determine the sequence of recommended changes and the road map for improvement.

However, in my opinion—and this is only my opinion, not the position of the ISTQB Advanced syllabus—the business alignment of the assessment road map from the nonprescriptive models, which takes into account business pain and business benefit, makes an organization more likely to support implementation of the improvements. As a consultant who spends a lot of time working with clients on this question of process improvement, a plan based on sound business benefits has a better chance of acceptance and funding than a plan based on an assertion that the organization must make improvements in a particular order.

8.4.1. A Generic Process Improvement Framework

Let's look at a generic process improvement process, which can harness one or more of the process models to a finer-grained plan/do/check/act Deming cycle.

- **Initiate:** This step defines the goals, scope, objectives, and coverage of the process improvements to be made. If you use a standard process model or models, then you should choose those models at this point. If you will use a home-grown model, then you should develop or refine it now. Based on the objectives, you should carefully choose metrics for the outcomes—the means by which you'll measure success of the improvements—if the model or models do not provide them. Analogous to a test policy, you now have a policy for your improvement. You must

confirm stakeholder support for this improvement policy.

- Measure: You now perform the assessment. If this is the first assessment, this is sometimes referred to as establishing a baseline. You will evaluate any changes you make—hopefully for the better—against this baseline. This assessment gives you a list of possible process improvements.
- Prioritize and Plan: At this point, you define the sequence for the process improvements. With a prescriptive model, the sequence is inherent in the model. For nonprescriptive models, you must consider the business value for each improvement based on factors such as return on investment; risks and constraints; alignment with organizational strategies, priorities, and projects; measurable quantitative or qualitative benefits; and the level of organizational pain associated with the way particular processes are now.^[5] With the sequence of improvements defined, you can now develop and launch your plan for the delivery of the improvements.

[5] Organizational pain matters because you must sell the plan for process improvement to the organization. A basic rule of sales says that people will expend more energy to escape pain than they will to approach a desirable improvement, and they will expend that energy more urgently.

- Define and Redefine: You now implement the process improvements. This involves defining new processes and updating existing ones. You ready the improved processes for deployment.
- Operate: You now deploy the process improvements. The steps required to do this can vary based on the readiness of the people and the organization, constraints, political considerations, and more. Common activities for this step would include any training and mentoring of the people involved, running one or more pilots of the improved processes, and perhaps full deployment of the processes across the organization.
- Validate: With the process improvements deployed, you now use your metrics to evaluate success. At this point, you might find yourself more successful than you expected, as successful as expected, not as successful as expected, or, in the worst case, not successful at all. The less successful you have been, of course, the more effort you'll need to expend to keep the improvement process moving forward.

- **Evolve:** Here you determine what the next steps should be. You might decide to revisit these improvement activities for the current processes that you just improved, especially if you did not get the results you desired. If you are using a prescriptive model with a maturity formula, you might evaluate your current level of maturity after this set of improvements. And, you might decide to move to the next process on the list for improvement.

While you can use this approach in conjunction with a standard model, it's also possible to use this by itself. The model serves to focus the attention on particular test processes that the industry experts who developed the model have found important. However, you can apply it to your current test processes directly, using analysis of the effectiveness and efficiency of the current processes along with postproject retrospective meetings to ensure alignment with actual delivered quality and results.

8.4.2. Case Study: Results of a Test Assessment

My RBCS consultants and I do many assessments for clients. We typically use the Critical Testing Process framework, which we'll discuss shortly. [figure 8-2](#) shows an extract from an RBCS assessment report based on an assessment I did in conjunction with our operation in India for a client with operations in both the United States and India. This initial section of the report listed a dozen or so prioritized recommendations. These are from the high-priority group of recommendations. I suggest you read [figure 8-2](#) closely before continuing to read my description of some key elements.

As you can see, this client was overly reliant on system test as the sole filter for defect detection and removal. Consequently, the test team found many bugs during that period. The development team then removed a large percentage of these bugs. This was cheaper than removing the bugs in production, as discussed in [chapter 3](#), but more expensive than it would have been had the project team found and removed the bugs during early phases of the lifecycle. The inability to deal with all the bugs led to both delays in release and excessive overtime.

Figure 8-2. Recommendations for improvement

I made five recommendations to address this problem:

1. Ensure that every development task has an associated code review task and unit testing task. Metrics showed that code reviews and unit testing, while they were performed, did not remove the same number of defects as we saw in those companies that followed industry best practices. Root cause analysis found that underperforming code reviews and unit testing together accounted for the largest number of bugs escaping into system test.
2. Continue to formalize the requirements and design processes. A major source of defects was the requirements specification process. Holes in the requirements and design specification, especially in the area of nonfunctional requirements, led to holes in the tests. So, the organization needed a better process, including ensuring proper reviews, to reduce the total number of bugs introduced in the lifecycle.
3. Focus on prevention of bugs via requirement reviews or early removal of bugs during unit test. By this I meant that it was not enough just to do requirements reviews and unit tests, but also that these activities explicitly focus on finding and removing bugs.
4. Gather metrics on the percentage of bugs introduced and removed in each major project phase, and take action to drive the percentage of bugs escaping from one phase to the next as close to zero as possible. Here I am calling for a measured process of defect management, one that leads to an awareness of where quality problems are introduced

and where they are not found in sufficient numbers.

5. Adopt a uniform process of formal bug tracking across all projects and a consistent phase at which formal bug tracking is introduced. Formalized bug tracking processes like those described in [Chapter 7](#) are not necessary to gather all bug metrics. Early bug-detecting activities in the lifecycle, like requirements and design reviews and unit testing, can often use informal techniques to track bugs and gather metrics, while formal test phases like system test and acceptance test require more formal techniques.

I provided additional supporting details behind these recommendations, including the business case for them.

8.5. Improving the Test Process with TMM

Learning objectives

(K2) Summarize the testing improvement process as defined by TMM.

(K2) Explain the evaluation criteria of the test improvement model TMM.

The Testing Maturity Model, or TMM, was developed by Ilene Burnstein and her colleagues. Dr. Burnstein is a professor at the Illinois Institute of Technology. The Testing Maturity Model was initially published in 1996.

Burnstein and her coauthors developed TMM to be complementary to and aligned with the CMM. What I mean by "complementary" is that it fills testing-related gaps in CMM. By "aligned with" I mean that it preserves the same idea of five maturity levels, each representing a level of testing maturity similar to CMM's maturity levels representing development process maturity.

With the advent of CMMI and the obsolescence of CMM, TMMi is gradually replacing TMM. TMMi is based on TMM and designed to complement CMMI. It uses the CMMI structure of process areas, generic goals, generic practices, specific goals, and specific practices. TMM defines testing as encompassing all software-quality-related activities. For example, requirements reviews are seen as testing activities. This is similar to the ISTQB Foundation and Advanced syllabi.

As I mentioned, TMM has five maturity levels for process evaluation and improvement. Each of the levels contains a number of defined process areas. The organization must completely fulfill each process area, in terms of the key practices being in place, before it can advance to the next level. This is sometimes called a staged representation. In ISTQB Advanced syllabus parlance, TMM provides both a process reference model and a content reference model. This means that you can use TMM both prescriptively—i.e., in order to achieve a desired level of maturity in some phased fashion—or nonprescriptively—i.e., in order to solve specific business problems and to identify and to relieve specific organizational sources of pain.

Let's look at the five levels of TMM, along with their associated characteristics and process areas. The first level is called initial. Testing is chaotic and undefined. There are few if any documented tests, no real test

process, no structure to the testing, and no concept of test levels. Tests are often developed immediately after coding, generally in an ad hoc way without any application of test design techniques. Testing is often part of debugging both process-wise and organizationally. There are usually no articulated goals for testing, but, to the extent that people think about what testing is supposed to accomplish, they tend to assume that testing should prove that the software works. Since there is no test process, there are no key process areas to "achieve" this test level.

Notice that reactive test strategies, poorly managed, would tend to fit into this level. In addition, agile development methodologies, if not tempered with good testing practices, would tend to devolve into this kind of unmanaged muddle.

The second level is called definition. At this level, testing is defined and clearly separated from debugging, in terms of process and possibly also organizationally. To achieve level two, the organization needs processes in place to establish test policies and goals; to create and execute test plans; to apply formal test design, implementation, and execution techniques and methods; and to set up and maintain a proper test environment.

The third level is called integration. At this level, testing is fully integrated into the software lifecycle, along the lines of what is discussed in the ISTQB fundamental test process and in this book. The test process is documented. It follows applicable standards. To achieve level three, the organization needs processes in place to ensure adequate test training for the staff, to establish a testing process and integrate it into the software lifecycle, and to exert test monitoring and control during test execution. It also needs an independent test organization.

The fourth level is called management and measurement. At this level, testing is thoroughly defined; well-founded and well-aligned with project, product, and organizational needs; and measurable. Testing includes preventive activities like reviews or inspections. To achieve level four, the organization needs processes in place to ensure that peer reviews occur, that test results are measured properly, and that those measures are used for a meaningful evaluation of software quality.

The fifth level is called optimization. At this level, testing is completely defined. It is controlled in terms of effectiveness and efficiency via appropriately chosen metrics. It is continuously improving through the use of those metrics. To achieve level five, the organization needs processes in

place to gather and use valid data in support of defect prevention, test process optimization, and end-to-end quality control.

Notice that there is a philosophy embedded in this model. The philosophy is one of establishing principles first and then growing from there. The test policy is seen as necessary before the independent test team is even required. When RBCS performs test assessment, we rarely find that the organization went about growing its test maturity in that direction. However, if you see a need to grow test maturity in a "top-down" fashion, this model might fit well.

Notice, too, the emphasis on definition and documentation. So this technique is likely to result in a test group that focuses heavily on producing documentation of its process and its work products. In certain situations such as regulated industries, this bias toward documentation might make this model fit well.

How do you use TMM for assessment and improvement? You perform an assessment against the key process areas. For each area, you need to achieve a number of predefined maturity goals and subgoals. Based on the fulfillment or nonfulfillment of a particular process area, you can assess the current maturity.

The goals and subgoals are defined in terms of overall activities, specific tasks, and responsibilities. You assess your achievement of these goals and subgoals according to the perspective of the managers, developers, testers, customers, and users.

Remember that the maturity model is staged. Staging has two implications. First, until you have achieved all the goals and subgoals for a process, that process area is not fulfilled. So, a process area is either unfulfilled—meaning we get no credit for any improvements made to it—or fulfilled—meaning we don't have to improve it again.

The binary nature of process fulfillment is likely to greatly simplify the assessment process. With clear criteria for fulfillment, either a process area is or is not fulfilled. Different assessors looking at the same organization should arrive at the same maturity number. However, attempts to reduce complex situations to black or white evaluations can simplify away important distinctions.

The second implication of the staged model is that you must fulfill all the processes at the lower levels in order to claim a higher level. For example,

suppose you have fulfilled the process areas for level 2 and level 3 except that you do not have an established test policy. In that case, you score at the initial or level 1 maturity, because you can't fulfill all the process areas for the definition, or level 2 maturity, not to mention integration, or level 3 maturity.

Similarly, suppose you have fulfilled the process areas for levels 2, 3, and 4 except that you don't have a training program in place for your testers. In this case, you score only at definition, or level 2 maturity, because you can't fulfill all the process areas for the integration, or level 3 maturity, not to mention the management and measurement, or level 4 maturity.

This is why I described this model as prescriptive. To make regular, smooth, incremental progress toward the highest level of maturity, level 5, optimization, you have to improve processes in the order prescribed by Burnstein and her colleagues. That sequence of improvements will work for you if you subscribe to the top-down, documentation-heavy philosophy inherent in the model too. In other words, if the philosophy and approach described by Burnstein strike you as a good fit for your organization, then following the TMM model will probably be a good fit too.

I could postulate a number of situations in which the TMM philosophy—and thus the TMM order of process improvement—would make sense. For example, in contract development on a project in a regulated industry, the need to clarify the goals of testing in advance and to establish a solid, auditable paper trail would recommend this approach.^[1]

[1] You can find more information about TMM in Ilene Burnstein, et al.'s book, *Practical Software Testing*, and on TMMi at www.tmmifoundation.org.

8.6. Improving the Test Process with TPI

Learning objectives

(K2) Summarize the testing improvement process as defined by TPI.

(K2) Explain the evaluation criteria of the test improvement model TPI.

The Test Process Improvement model, or TPI, was developed by Tim Koomen and Martin Pol based on their experience with various clients. Koomen and Pol helped to develop Sogeti's proprietary TMap model. While we don't cover TMap directly in this book or in the Advanced syllabus, the foundations of the TPI model lie in TMap.

In comparison to TMM, TPI is a finer-grained measurement. I mean this in two ways. For one thing, it breaks the test process into more process areas. For another thing, it allows multiple levels of maturity for most of the process areas rather than having a process area be either fulfilled or unfulfilled. I'll return to this key distinction later.

As with TMM, you can use TPI both to measure test process maturity and to suggest improvements. Because of the maturity measurement aspect, it is prescriptive for much the same reasons that TMM is prescriptive.

Unlike TMM, TPI is not aligned with CMM. It is not aligned with ISO 15504 either. Of course, being unaligned with CMM, it is not aligned with TMM.

TPI has 20 key process areas. Eighteen of these process areas fit into one of four cornerstones of the test process:

- Lifecycle
- Organization
- Infrastructure and tools
- Techniques

Two process areas apply to all cornerstones. This concept of test process cornerstones is from TMap.

As I mentioned before, rather than a process area being either fulfilled or unfulfilled, each process area has 1, 2, 3, or 4 levels of maturity. There is a

"level 0" level of maturity for each area as well. These are rated from level A (the first level of maturity) all the way up to level D for some process areas.

You can use the process area maturity levels to determine overall maturity on a scale of 0 to 13, using the TPI maturity matrix, which is shown in [figure 8-3](#). This figure also shows the number of levels for each of the process areas. We can then group the maturity levels into four maturity levels, from initial to optimizing.

TPI considers 20 process areas, organized by cornerstones. For the lifecycle cornerstone, there are three test process areas:

- Test strategy
- Lifecycle model
- Moment of involvement

For the techniques cornerstone, there are four test process areas:

- Estimating and planning
- Test specification
- Static testing
- Metrics

For the infrastructure and tools cornerstone, there are three test process areas:

- Test automation
- Test environment
- Office environment

For the organization cornerstone, there are eight test process areas:

- Commitment and motivation
- Testing functions and training
- Scope of methodology

- Communication
- Reporting
- Defect management
- Testware management
- Test process management

Spanning all cornerstones are two test process areas:

- Evaluation (verification and validation)
- Low-level testing (unit testing and integration testing)

In [figure 8-3](#), the 20 process areas are laid out in a matrix in relation to their levels of maturity and the aggregate measures of maturity those process area measures generate. Let me explain this matrix column by column. In the leftmost column is a code to help you remember into which of the four cornerstones each test process area is classified. The column next to that gives the test process name. Now, in the main body of the chart, we have 14 columns, headed with integers from 0 through 13. In each row of the chart, we see a sequence of letters for each process. Each row contains at least an A. In one case, the row contains only a single letter, A. Office Environment contains only an A in the column headed 1. This means that the Office Environment process, like all the process areas in TMM, is either fulfilled or unfulfilled.

However, look at the Metrics row or the Testware Management row. Each of these rows contains four letters, A through D. In other words, there are four *degrees* of fulfillment, if you will. We are expected to improve each process area gradually, in parallel with gradual improvements to most other process areas.

So, there are two dimensions of maturity shown in this table:

- Process by process: Each process has some degree of maturity, measured on one of four possible scales: none or A; none, A, or B; none, A, B, or C; or none, A, B, C, or D.
- Overall: Based on the maturity of each process, we can assess the overall maturity by moving along the upper row from 0 to 13 until we find that one or more processes has not achieved the required level of

maturity to move to the next number.

Let's look at how this might work. Suppose you start at the level 0 of overall maturity. To move to level 1, you need to make the following improvements:

- Take the Test Strategy process area to level A.
- Take the Lifecycle Model process area to level A.
- Take the Test Specification Techniques process area to level A.
- Take the Office Environment process area to level A.
- Take the Commitment and Motivation process area to level A.
- Take the Reporting process area to level A.
- Take the Defect Management process area to level A.
- Take the Test Process Management process area to level A.

Now, to move from level 1 to level 2, you need to make the following improvements:

- Take the Moment of Involvement process area to level A.
- Take the Communication process area to level A.
- Take the Testware Management process area to level A.

Now, you can see that the columns are shaded from dark gray to white to light gray gray to dark gray again. This shading shows the TPI grouping of the overall maturity scale into four maturity levels.

The first overall maturity, 0, is termed Initial. This term implies the same kind of formless chaos discussed earlier in the context of TMM. Of course, with TPI and with TMM, this characterization of some maturity level as initial, with no structure or process, really means that the test process under consideration doesn't attend to at least some of the process areas the authors of the process models said were important.

That makes sense to me. However, it's a bit confusing that Koomen and Pol on the one hand and Burnstein and her colleagues on the other hand don't

agree on what process areas a test team must address—and to what degree—to escape from this initial test process Gehenna.

The next overall maturity level, 1 through 5, is termed Controlled. At this level, the test process is broken into test levels and executed according to a strategy and a plan. You employ sufficient test specification techniques. You record, report, and manage incidents. You have both the testware and test environment under control. You provide training for the test staff.

The next overall maturity level above that, 6 through 10, is termed Efficient. With the test process under control, you move to improve efficiency. You can do this in a number of ways, according to TPI. You can automate testing. You can better integrate the test process at various levels and better integrate testing with the lifecycle.

The ultimate overall maturity level, 11 through 13, is called Optimizing. Now you want to set a course for continuous test process improvement. You use data to drive this process.

As with TMM, there is a philosophy embedded in TPI, but it's a different philosophy. The philosophy focuses on establishing a working process first and then gradually focusing more on alignment, efficiency, and improvement. For example, a test team following TPI would likely establish the test policy in the efficient level of maturity. This is well after the point where TMM would have instructed the team to do so. So this is a "bottom-up" model. It is more common for us to encounter organizations that went about growing their test maturity in this direction, though often more by accident than by design.

Notice, too, that TPI does not place the same emphasis on definition and documentation. So this technique is likely to result in a test group that can better operate in a lightweight documentation environment.

How do we use TPI for assessment and improvement? To perform the assessment, we check each of the 20 key process areas against maturity checkpoints in the TPI model. TPI assessments involve both quantitative metrics and qualitative interviews

If all the checkpoints for a given level of maturity for a process are satisfied, then that process area has achieved that level of maturity. Suppose that all checkpoints involved in level A and level B for the key process area of reporting are satisfied. In that case, reporting has achieved level B.

Figure 8-3. TPI test maturity matrix



This gradual progression, as I said before, gives TPI its finer-grained nature. The Advanced syllabus describes this as a continuous representation, contrasted to the staged representation of TMM. TPI is continuous in the sense that there are degrees of maturity for each process area.

However, TPI is prescriptive. The overall maturity scale and levels, along with the dependencies between various key areas and levels, impose an ordering on the improvements. You improve processes incrementally, but TPI guides you to improve the processes in a particular order. I gave you an example of this ordering in the discussion previously about how to move from level 0 to level 1 and from level 1 to level 2.

So, like TMM, TPI prescribes the order in which to improve processes. Indeed, TPI also prescribes the order in which to improve within process areas, based on the ordering of the checkpoints. In other words, you must satisfy the checkpoints for level A before those for level B, and so forth. Thus, as with TMM, the TPI model makes sense if you subscribe to the TPI philosophy, which as I said is more bottom-up and documentation light.

Again, I could postulate a number of situations in which the TPI philosophy—and thus the TPI order of process improvement—would make sense. For example, in a start-up company building mass-market PC applications or websites, the need to establish a test team that adds value quickly and then

improves steadily but incrementally over time would recommend this approach.^[5]

[5] For more information, see Martin Pol and Tim Koomen's book, *Test Process Improvement*. I also recommend Koomen, et al.'s, *T-Map Next*, since TPI's process model derives from TMap.

8.7. Improving the Test Process with CTP

Learning objectives

(K2) Summarize the testing improvement process as defined by CTP.

(K2) Explain the evaluation criteria of the test improvement model CTP.

When I wrote my book Critical Testing Processes in the early 2000s, I started with the premise that some test processes are critical, some are not. I designed this minimalist approach to test process improvement to focus the test team and test manager on a few test areas that they simply must do properly. This approach is in contrast to what I feel are the more expansive models inherent in TPI and TMM.

Which is better? Obviously, I prefer a focused, minimalist model. However, if you prefer a model that gives you more complete guidance—at the cost potentially of spending time looking at less-important areas—you might want to use TMM or especially TPI. If the prescriptive nature of TMM or TPI bothers you, you can always choose to omit the maturity models and use only the process content assessment portions of the models.

Returning to the CTP model, I'll give more detail about what I mean by a critical testing process in a moment. However, here's a quick way to think about it: If a test team executes the critical testing processes well, it will almost always succeed, but if it executes these activities poorly, even talented individual testers and test managers will usually fail.

I not only wanted a focused model, but also a flexible one. Therefore, CTP allows for tailoring. It allows you to identify and deal with specific challenges to your test processes. It identifies various attributes of good processes, both quantitative and qualitative. It allows you to use business value and organizational pain to select the order and importance of improvements. It is also adaptable to all software development lifecycle models.

Because I wanted to focus on a small number of processes, how did I select them? What is a critical testing process?

First, I defined a process as a sequence of actions, observations, and decisions. This definition takes some of the TMM and TPI process areas and puts them into larger settings. For example, the setting of a test policy

becomes part of a larger test context establishment process, and office environments are subsumed implicitly within the test system development process. Next, I defined testing as the activities involved in planning, preparing, performing, and perfecting the assessing of the quality of a system.

So, with a definition of a test process firmly in hand, what makes a test process a critical test process? I applied four criteria:

- Is the process repeated frequently, so that it affects efficiency of the test team and the project team?
- Is the process highly cooperative, involving a large number of people, particularly cross-functionally, so that it affects test team and project team cohesion and cooperation?
- Is the process visible to peers and superiors, so that it affects the credibility of the test team?
- Is the process linked to project success, in such a way as to affect project team or test team effectiveness?

In other words, a critical test process directly and significantly affects the test team's ability to find bugs, build confidence, reduce risks, and generate information.

Based on these criteria, I identified the following 12 critical testing processes:

- Testing. The overall process, viewed at a macro, strategic level. It consists of 11 constituent critical testing processes.
- Establishing context. This process aligns testing within the project and the organization. It clarifies expectations on all sides. It establishes the groundwork for tailoring all other testing processes. As you can see, when appropriate, the creation of test policy and test strategy documents would fit into this process. However, those are not prescribed since some situations might need them and some might not.
- Quality risk analysis. This process identifies the key risks to system quality. It aligns testing with the key risks to system quality. It builds quality and test stakeholder consensus around what is to be tested (and how much) and what is not to be tested (and why). As you can see, this

process requires tailoring to accommodate a strategy other than risk-based testing.

- Test estimation: This process balances the costs and time required for testing against project needs and risks. It accurately and actionably forecasts the tasks and duration of testing. It demonstrates the return on the testing investment to justify the amount of test work requested.
- Test planning. This process builds consensus and commitment among test team and broader project team participants. It creates a detailed map for all test participants. It captures information for retrospectives and future projects.
- Test team development. Because testing is only as good as the team that does it, this process matches test team skills to the critical test tasks. It assures competence in the critical skills areas. It continuously aligns team capabilities with organizational value of testing.
- Test system development. This process ensures coverage of the critical risks to system quality. It creates tests that reproduce the customers' and users' experiences of quality. It balances resource and time requirements against criticality of risk. It includes test cases, test data, test procedures, test environments, and other support material.
- Test release management. If we don't have the test object, we can't test it. If the test items don't work in the test environment, we can't test them. If each test release is not better than the one before, we're not on a path for success. So, this process focuses on how to get solid, reliable test releases into the test environment.
- Test execution. This process, the running of test cases and comparison of test results against expected results, generates information about bugs, what works, and what doesn't. In other words, this is where the value of testing is created. This process consumes significant resources. It occurs at the end of the project and gates project completion.
- Bug reporting. This process creates an opportunity to improve the system (and thus to save money). While test execution generates the value of testing, this process delivers part of the value of testing to the project team, specifically the individual contributors and line managers. It builds tester credibility with programmers.
- Results reporting. This process provides management with the

information needed to guide the project. It delivers another part of the value of testing to the project team, particularly line managers, senior managers, and executives. Because test results are often bad news, it separates the message from messenger. It builds tester credibility with managers.

- Change management. This process allows the test team and the project team to respond to what they've learned so far. It selects the right changes in the right order. It focuses efforts on the highest return-on-investment activities.

You might notice that I've described each of these processes in terms of what an optimal process will achieve. If the process does not achieve those standards of capability—and more—then it is not optimal and has room for improvement.

How do we use CTP for assessment and improvement? Test process improvements using CTP begin with an assessment of the existing test process. This assessment will identify which of the 12 test processes are currently done properly and which need improvement. The assessment results in a set of prioritized recommendations for improvements. The assessor bases those recommendations on organizational needs, not some maturity model.

Since the assessments are tailorabile, they can vary depending on the specific organization and its needs. We have done narrowly focused CTP assessments that looked only at one test team, we have done CTP assessments that looked only at a one or two test processes like the test system, and we have done broad CTP assessments that looked at everything that affects quality. So, while CTP assessments vary, we tend to examine the following metrics during a CTP assessment:

- Defect detection percentage
- Return on the testing investment
- Requirements coverage and risk coverage
- Test release overhead
- Defect report rejection rate

In addition, we also tend to evaluate the following qualitative factors,

among others, during a CTP assessment:

- Test team role and effectiveness
- Usefulness of the test plan utility
- Test team skills in testing, domain knowledge, and technology
- Value of the defect reports
- Usefulness of test result reports
- Change management utility and balance

Once an assessment has identified opportunities for improvement, the assessor will develop plans to implement this improvement. While the model includes generic guidelines for process improvement for each of the critical testing processes, the assessment team is expected to tailor those heavily.

I designed the CTP model to be very flexible. It does assume the primary use of an analytical testing strategy, balanced with a dynamic testing strategy. You can adapt CTP to use other test strategies primarily, such as checklist-based or model-based, but that would have a systemic impact on the model. Should I get a chance to write a second edition of Critical Testing Processes, I will probably build that additional element of flexibility into the model.

In [figure 8-4](#), you see the summary results of a CTP assessment for a client. You might want to take a moment to read this summary.

As you can see, our evaluation is fine grained, much like TPI's, in that we don't rate the process areas fulfilled or unfulfilled but rather given them a qualitative evaluation. We specifically identify opportunities to improve, sometimes in fine-grained, specific areas of the process. The assessment is not only a snapshot in time, but it captures the momentum of the organization in a specific process area.

In [figure 8-5](#), you see a specific quantitative metric used to measure the test system development process. The metric looks at test case detail. You can substitute "test procedure" if that's the term used for the actual unit of work assigned to a tester executing a manual test. We can assume that the more words per unit of execution time in the written test case or procedure,

the more explicitly specified the test is and thus the more instruction and guidance we are giving the tester.

Figure 8-4. CTP assessment summary

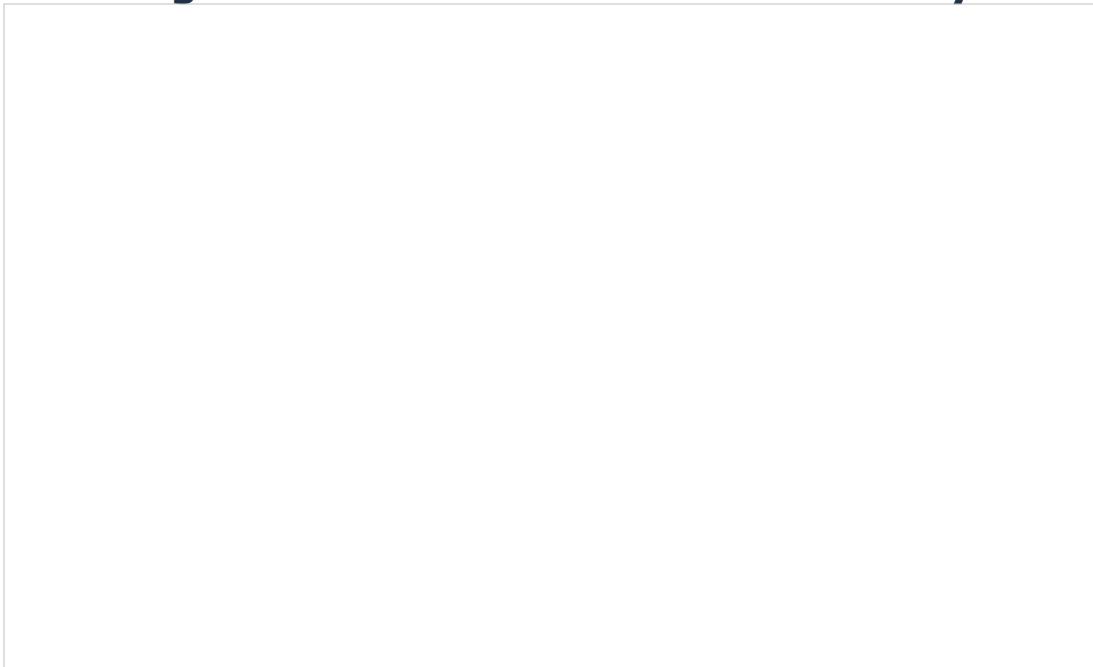
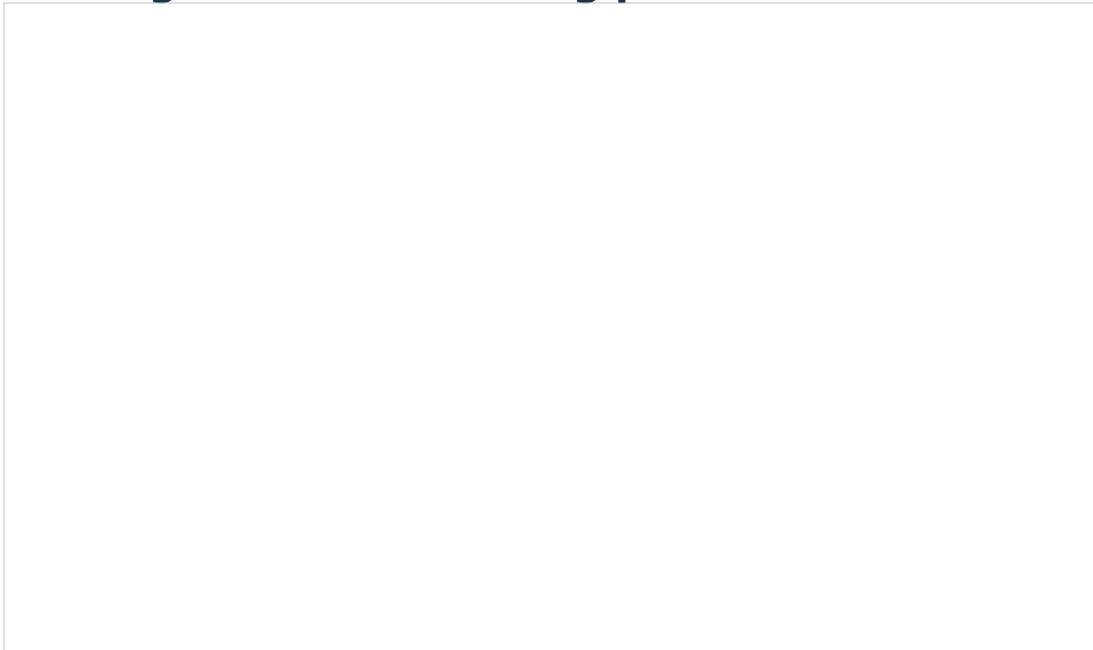


Figure 8-5. Measuring processes in CTP



The point is to evaluate whether the level of detail in the test cases is suitable for the business needs. You see, CTP prescribes neither a lightweight documentation regime nor a heavyweight one but instead measures to see the actual current state of documentation and then evaluates that against the business needs. For this specific metric, we are

looking at test cases (or test procedures, depending on the naming convention at the assessed organization).

To calculate this metric, we take a sample of at least 10 test cases. We count the number of words in each test case and the number of minutes required to execute it. We also calculate the average and standard deviation. What we often see during assessments is an order-of-magnitude variation between tests, or sometimes even a two- or three-orders-of-magnitude variation. That's fine if the variation is carefully considered and managed, but generally it arises from the lack of clear management direction on this topic as well as the lack of any understanding by the individual contributors as to what drives documentation density needs in one direction or another.

Let me raise a couple of additional points about this metric. For one thing, we want to measure across multiple testers and applications, and the larger the sample the better. For another, this metric applies to manual tests only, as we use other metrics for automated test cases.

There are some 5 or 10 metrics for each critical testing process, along with some qualitative evaluations. Now, we don't typically look at every metric for every process on every single assessment because the selected metrics, like the model itself, are tunable. This results in a customer-centric assessment.^[1]

[1] My book *Critical Testing Processes* describes the critical testing processes model.

8.8. Improving the Test Process with STEP

Learning objectives

(K2) Summarize the testing improvement process as defined by STEP.

(K2) Explain the evaluation criteria of the test improvement model STEP.

The Systematic Test and Evaluation Process (STEP) model, like CTP, is non-prescriptive in terms of improvements. That is, you can make the improvements in the order of business importance, not based on arbitrary maturity models. That said, STEP does make some assumptions about the overall testing approach, either as currently practiced or as the organization would like to practice it:

- It assumes an analytical requirements-based testing strategy.
- It assumes that testing starts at the beginning of the lifecycle, which is also embedded in other test process models.
- It assumes that tests are used as requirements and usage models, which is to say that tests validate the requirements and use cases while those are being developed, thus requiring overlap of the system requirements specification and the test design specification processes.
- It assumes that testware design will not only overlap system design but also in fact will lead and precede system design.
- It assumes that the test process is focused on early defect detection or even prevention.
- It assumes the use of root cause analysis and other sophisticated techniques, such as those discussed in [chapter 3](#), whereby defects are systematically analyzed with an eye toward earlier detection and prevention.
- It also assumes that testers and developers work together to accomplish all this.

As you can see, many of these assumptions and requirements are consistent with principles and concepts from the ISTQB syllabi. These assumptions and requirements can make the model a good fit for some organizations. If the organization currently satisfies some of these

assumptions, and agrees with the others, it is a good assessment framework. Even if none of these assumptions are satisfied, if the organization agrees they *should* be satisfied, then a STEP assessment will guide the way toward doing so. However, for those test teams that do not satisfy these assumptions and that work in organizations that would not support these assumptions, the STEP framework is likely to align poorly with the organization's needs.

So, how do we use STEP for assessment and improvement? The assessment involves evaluating three major phases of testing:

- Planning
- Acquisition, which consists of two activities, test analysis, and test design and implementation
- Measurement, which consists of two activities, test execution and test reporting

While this is similar to the ISTQB fundamental test process in the way it is decomposed, it is different in some details.

STEP documentation for these phases is based on IEEE 829. In fact, David Gelperin, the initial creator of the STEP framework, helped to create the IEEE 829 standard. Therefore, you can consider STEP a proprietary process developed by the U.S.-based company Software Quality Engineering as a wrapper around the IEEE 829 documentation standard.

A STEP assessment involves both quantitative metrics and qualitative interviews. The following metrics are quantitative:

- Test status over time
- Test requirements or risk coverage
- Defect trends, including detection, severity, and clustering
- Defect density
- Defect removal effectiveness
- Defect detection percentage
- Defect introduction, detection, and removal phases

- Cost of testing in terms of time, effort, and money

The following factors are qualitative:

- Defined test process utilization
- Customer satisfaction

The STEP model itself is nonprescriptive. My associates and I have worked with clients that have implemented this framework in a way that they tailored to their organizational needs.

As noted earlier, the model assumes the primary use of an analytical testing strategy, specifically a requirements-based testing strategy. However, STEP augments that with risk assessment of requirements elements.

Like CTP, STEP allows for balancing with a dynamic testing strategy. And like CTP, it can be adapted to use other test strategies primarily, such as checklist-based or model-based, but that would have a systemic impact on the model.

The STEP model is nonprescriptive. A team can choose to improve any given process, and any aspect of any given process. However, the model is often blended with TPI to derive a maturity rating, which transforms the model into a prescriptive one.

8.9. Capability Maturity Model Integration, CMMI

Learning objectives

(K2) Summarize the key process areas of verification and validation in CMMI.

(K2) Explain the key process areas of verification and validation in CMMI.

The Software Engineering Institute, a division of Carnegie Mellon University in Pittsburgh, Pennsylvania, was chartered by the United States Department of Defense to develop the Capability Maturity Model in the 1980s. The Capability Maturity Model (or CMM) drew heavily on the concepts of Total Quality Management (or TQM). Total Quality Management itself grew out of the ideas of statistical quality control advanced by J. M. Juran, W. E. Deming, and a number of Japanese practitioners. The core philosophy of Total Quality Management is that the quality of the process by which a product is developed and created will largely determine the quality of the product that is ultimately produced. This philosophy is embedded in the Capability Maturity Model, its variants, and its successors.

The Software Engineering Institute expanded the model to include a number of variants, including software development, systems development, software acquisition, and more. In the late 1990s and early 2000s, the Software Engineering Institute produced the Capability Maturity Model Integration (or CMMI), which integrated these variants.

Capability Maturity Model Integration is a five-level model of maturity. The five maturity levels are as follows:

- Initial: The process is unpredictable, poorly controlled, and reactive to events.
- Managed. The process is established at project level but is still often reactive to events.
- Defined. The process is established across the organization. The process is usually proactive in attempt to anticipate and control events.
- Quantitatively managed. The process is measured and controlled at the organizational level. We have data and can compare that data across projects.

- Optimizing. Not only is the process measured and controlled at the organizational level, with comparable data across projects, but also the focus is on using that data and other information to continuously improve the process.

At this point, you can see the alignment between CMMI and TMM.

So, why do we need TMM and other test process models if we have CMMI? Well, because CMM and CMMI focus on the process as the primary determinant of product quality, these models underemphasize testing. Philosophically, the Software Engineering Institute believes that good process is primary and therefore testing should be secondary. This philosophy works for assembly-line production of physical items but has proven unsatisfactory in practice for software and systems.

CMM was focused on a staged implementation, like TMM. However, CMMI allows continuous or staged implementation. In a continuous approach, CMMI allows for various levels of maturity for each process area. However, I have to say that I don't see many organizations adopting CMMI in a continuous fashion because many organizations appear to use the CMMI maturity level as a sales tool.

As with TMM, in the staged representation you have five levels of maturity. Each level builds on the process areas achieved in the previous level or levels. However, with a continuous representation, the organization can concentrate its improvement efforts on the process areas that need the most improvement, based on its unique considerations. The organization is allowed to disregard the predecessor process areas required in the staged model.

The staged representation of CMMI ensures commonality with CMM. The continuous representation is more flexible and less prescriptive. However, as I noted in the discussion on outsourced testing in [chapter 3](#), many organizations use CMM or CMMI accreditation for marketing purposes. For these organizations, the staged representation is important because it allows them to claim a particular level of maturity.

At level three, the Defined level, CMMI deals with two specific process areas that are of particular interest to us as testers. The first is Verification. As discussed in the Foundation syllabus and the ISTQB glossary, the purpose of verification is to ensure that selected work products meet their specified requirements. Verification can include checking to see that each

requirements specification element is present in the design specification and that each design specification element is present in the implementation.

CMMI calls for four practices in the development process for verification:

- Select Work Products for Verification
- Establish the Verification Environment
- Establish Verification Procedures and Criteria
- Perform Verification

These are not one-time practices but rather activities embedded in various places in the development lifecycle.

The second process area is Validation. Again, consistent with the Foundation syllabus and the ISTQB glossary, the purpose of validation is to demonstrate that a product fulfills its intended use in its intended environment and likewise for product components.

CMMI calls for four practices in the development process for validation:

- Select Products for Validation
- Establish the Validation Environment
- Establish Validation Procedures and Criteria
- Perform Validation

Again, these are not one-time practices but rather activities embedded in various places in the development lifecycle.

The practices related to these process areas of Validation and Verification involve both static and dynamic test processes, which is also consistent with the ISTQB Foundation syllabus and Advanced syllabus. In other words, there is no inconsistency between the ISTQB concepts of software testing and the CMMI process areas of Verification and Validation. However, CMMI's process areas are not sufficiently developed to require the use of testing best practices. I discussed this in [chapter 3](#) when I covered outsourcing. So, augmenting CMMI with the test process models discussed in this chapter is necessary to ensure testing best practices.

As an example of CMMI compliance, let's examine the IVR project that I've used as a case study many times in this course. During that project, a consultant evaluated the IVR project for CMMI compliance. One area he looked at was validation. How did we stack up? Let's look at each practice:

- In terms of Select Products for Validation, we performed multiple levels of testing of the IVR systems and the system of systems. We had formal system test of each major system. We had preintegration testing of the systems together. We also had formal system integration test of all system together.
- In terms of Establish the Validation Environment, we tested in increasing production-like environments. Each system test level aimed to test each system in a production environment. Preintegration test was done in increasingly production-like environments. System integration testing was done in a complete production environment.
- In terms of Establish Validation Procedures and Criteria, we had defined test cases and exit criteria for each level. The realism of each test and the rigor of the planned entry and exit criteria increased with each test level.
- In terms of Perform Validation, we carried out each level of testing against the plans and cases. However, we did not always manage to adhere to the planned extent of rigor for entry and exit criteria.

ISTQB Glossary

Capability Maturity Model (CMM): A five-level staged framework that describes the key elements of an effective software process. The Capability Maturity Model covers best practices for planning, engineering, and managing software development and maintenance.

Capability Maturity Model Integration (CMMI): A framework that describes the key elements of an effective product development and maintenance process. The Capability Maturity Model Integration covers best practices for planning, engineering, and managing product development and maintenance. CMMI is the designated successor of the CMM.

So, while the project was not CMMI level 3 compliant, our testing was consistent with CMMI level 3, Defined.

8.10. Test Improvement Process Exercise

As HELLOCARMS's test manager, you want to improve your test process during this project. Refer to the exercises completed so far for data, and reflect on what you have learned about the HELLOCARMS project. Assume you are at the end of iteration two and the project will last about five more months.

Outline test process improvements you would like to implement related to one or more process areas. You may use any of the four process models as a framework or make up your own.

If you are working in a classroom, break into groups of three to five. Once each group has finished, discuss the results.

I suggest 20 minutes to outline the improvements, followed by 10 minutes of discussion.

8.11. Test Improvement Process Exercise Debrief

I will use the Critical Testing Processes framework to outline what I would do in terms of improvement.

1. *Testing*

Because one concern toward the end of iteration two related to the number of bugs, I would want to measure the defect detection percentage for testing done in iteration one. I would select a sample of 100 bugs found in iteration two and determine the percentage of bugs that were present during testing of iteration one. I would use that to estimate the total number of bugs that escaped detection in iteration one and thus the defect detection percentage for iteration one.

Based on that analysis, I would then take appropriate steps if warranted to improve the defect detection effectiveness of testing for subsequent iterations.

2. *Establishing Context*

To confirm the business values I identified in a previous exercise, I would survey the test stakeholders to confirm their perception of the values of testing. If I identified any perceptions of a lack of value delivered by my test team, I would address those promptly.

I would use cost of quality to attempt to quantify the value of testing so far based on a predicted or actual historical cost of a field failure. I would then look at ways to reduce the cost per defect found—e.g., introducing some easy test automation or using test outsourcing—to further increase the quantified value of testing.

I would also check to see if we had high regression of features from iteration one during testing of iteration two. If so, I would discuss the possibility of an automated unit test harness, say using JUnit or a similar facility, with the development manager to try to reduce regression risk earlier in the lifecycle.

In general, I would check for insufficient upstream quality control activities—such as requirements and design reviews, code reviews and static analysis, and unit testing—resulting in a large number of defects delivered into system test for each iteration. Such a large number of

defects makes it difficult to complete testing during an iteration of fixed length and reduces the likelihood of having a deployable system at the end of that iteration.

3. *Quality Risk Analysis*

To check correctness of the quality risk analysis, I would evaluate the defects found so far. I would look especially for the following:

- High number of defects in areas we had assessed as unlikely to contain defects
- Low number of defects in areas we had assessed as likely to contain defects
- Disproportionately large number of high-impact defects compared to low-impact defects in areas we had assessed as low impact
- Disproportionately large number of low-impact defects compared to high-impact defects in areas we had assessed as high impact

Based on this evaluation, I would correct the quality risk analysis, reallocate test effort, and reprioritize the test cases.

4. *Test Estimation*

I would analyze the actual effort and completion dates for all tasks against the estimated effort and completion dates in the Gantt chart. If the variance indicated an unachievable plan for the remaining three iterations, I would adjust the estimate and reobtain management approval.

5. *Test Planning*

I would analyze variances from the test plan to date, particularly in the area of any unanticipated or unmanageable test project risks. If the variance indicated an unachievable plan or other major risks for completing testing for the remaining three iterations, I would adjust the plan and reobtain management approval.

6. *Test Team Development*

I would perform a skills inventory for my test team (as discussed in the next chapter). If this assessment revealed gaps in test team skills, I

would look for ways to make immediate improvements in the most urgent areas during the remaining three iterations.

7. *Test System Development*

I would check to see what proportion of test cases were blocked during iterations one or two. I would then investigate whether those blockages were due to test case, test tool, or test environment design decisions. If so, I would correct those problematic decisions and their consequences to improve the ability to run tests without delay when needed.

I would also reassess test coverage on three dimensions: risks, requirements, and code. I would check to see if we had a large expansion of test scope from iteration one to iteration two. If so, I would then try to identify the cause of that expansion and the project risks posed by it and put a plan in place to manage the growth in test scope over the next three iterations.

8. *Test Release Management*

I would check to see if the large number of bugs found in iteration two had anything to do with build or installation problems. If so, I would recommend changes in those processes to the release engineering team based on a business analysis of the cost of those problems during iteration two.

9. *Test Execution*

The test completion and test hours charts indicated efficient, effective execution of test cases during iteration two. I would check to see if the defects were being found in priority order. Ideally, a test team should find defects in exact order of priority. If that is approximately true, then the quality risk analysis prioritization is correct and the test team is running the tests without encountering excessive blockage. However, if the test team is finding defects in an order that is inconsistent with their priority, root cause analysis as to why is in order, with improvements needed to correct that. Finding bugs in the proper order is particularly urgent when the defect find rate remains very high through a test execution period, as it has on this project.

10. *Bug Reporting*

I would check for an excessively high rate of the following bug reporting

dysfunctions:

- Bug reports rejected by the bug triage committee as describing not failures but rather correct behaviors
- Bug reports returned to the test team for further clarifying information
- Bug reports closed as duplicates by the development team
- Bug reports deferred by the bug triage committee

I would target a rate of less than 5% for each of these types of noise or inefficiency. I would investigate causes for any of these dysfunctions that exceeded that rate.

11. *Results Reporting*

I would survey the various project stakeholders as to whether they found the information coming from the test team complete, pertinent, timely, and concise. If they had any suggestions for improvement of test results reporting, I would implement those.

12. *Change Management*

I would check for a large rate of feature addition during past and future iterations. I would then assess whether that exceeded the capacity of my test team to handle the testing of such a volume of features. If so, I would adjust my resource plan and discuss with management.

8.12. Sample Exam Questions

To end each chapter, you can try one or more sample exam questions to reinforce your knowledge and understanding of the material and to prepare for the ISTQB Advanced Level Test Manager exam.

Q: Which of the following standards is useful as a source of templates for a test plan?

1. IEEE 1044
2. ISO 9126
3. IEEE 829
4. DO 178B

Q: Which of the following is a best practice for retrospective meetings that will lead to process improvement?

1. Ensuring management commitment to implement improvements
2. Allowing retrospective participants to rely exclusively on subjective assessment
3. Requiring that every project include a retrospective meeting in its closure activities
4. Prohibiting any management staff from attending the retrospective meeting

Q: 3. Assume you are a test manager and you have just concluded a project to create a programmable thermostat for home use to control central heating, ventilation, and air conditioning (HVAC) systems. After release, you find that quality risk analysis failed to identify certain significant risk items, resulting in defects escaping to customers.

Assume you are using TPI to improve the test process. Which of the following is a TPI process area that you would look at to address this problem?

1. Lifecycle model

2. Test policy and goals

3. Moment of involvement

4. Estimating and planning

Q: Which of the following statements accurately captures the distinction between evaluation criteria for the four test improvement models discussed in the Advanced syllabus?

- 1.** TMM and TPI use a maturity model to guide the order of improvement, while CTP and STEP provide a means for assessing the effects of certain improvements
- 2.** CTP and STEP use a maturity model to guide the order of improvement, while TMM and TPI provide a means for assessing the effects of certain improvements
- 3.** CTP and TMM use a maturity model to guide the order of improvement, while STEP and TPI provide a means for assessing the effects of certain improvements
- 4.** STEP and TMM use a maturity model to guide the order of improvement, while CTP and TPI provide a means for assessing the effects of certain improvements

Chapter 9. Test Tools and Automation

"What is the business case for automation?"

—A question posed by the author to a team of consultants who had been working for a client for four years without making significant progress in the percentage of test cases automated. They were unable to answer the question in any meaningful fashion.

The ninth chapter of the Advanced syllabus is concerned with test tools and automation. While the Foundation syllabus covers this topic as well, the Advanced syllabus goes beyond the Foundation material to provide a solid conceptual background for test tools and automation. In addition, the Advanced syllabus elaborates on the categorization of tools introduced in the Foundation syllabus. [Chapter 9](#) of the Advanced syllabus has three sections:

- 1.** Introduction
- 2.** Test Tool Concepts
- 3.** Test Tool Categories

Let's look at each section and how it relates to test management.

9.1. Introduction

Learning objectives

Recall of content only

In this chapter, we'll expand on the basic tool ideas laid out in the Foundation syllabus. We first address general tool concepts and then specific tools.

All testers need a basic grasp of the test tools available and what tools can—and can't—do. Too often, organizations and individuals bring bad expectations to the use of test tools, usually the expectation that the choice of the right tool will solve all testing problems. That's not realistic.

In the Advanced syllabus, we've grouped the tools according to role; i.e., those for test managers, those for test analysts, and those for technical test analysts. Of course, some tools have broader use, across multiple roles.

9.2. Test Tool Concepts

Learning objectives

(K2) Compare the elements and aspects of each of the following test tool concepts: benefits and risks, test tool strategies, tool integration, automation languages, test oracles, tool deployment, open source tools, tool development, and tool.

(K2) Understand the different phases in test tool implementation classification.

(K2) Describe why and when it is important to create a test tool strategy or road map for your test tool.

Test tools can be very useful. Indeed, some are essential. It's hard to imagine a test project that involves more than two or three people getting along without some incident tracking system. Generally, test tools can improve efficiency and accuracy of testing. However, you have to carefully select and implement tools to receive the benefits.

While we often think of test automation as meaning automation of test execution, we can automate other parts of the test process as well. However, it's probably true that most of the test automation that happens involves attempts to automate tasks that are tedious, such as regression testing, or difficult to do manually, such as performance testing.

Getting the full benefit from test tools involves not only careful selection and implementation, but also careful ongoing management. You should plan to use configuration management for test tools, test scripts, test data, and other test tool items and remember to link version numbers of tests and test tools with the version numbers of the items tested with them.

You should plan to create a proper architecture for your test automation system. Too often as a consultant and practitioner, I've seen test teams saddle themselves with constraints due to poor design decisions made at the outset of automation.

A good architecture supports another important aspect of good automation, which is creating and maintaining libraries of tests. With consistent decisions in place about size of test cases, naming conventions for test cases and test data, interactions with the test environment, and such, you can now create a set of reusable test building blocks with your tools. You'll

need a library in which to store them.

An automated test is a program for testing other programs. So, like any program, the level of complexity and the time required to learn it often means that you'll want to have some documentation in place about how it works, why it is like it is, and so forth. This documentation doesn't have to be fancy, but most automated test systems of any complexity need it.

Remember to plan for expansion and maintenance. Failure to think ahead, particularly in terms of how the tests can be maintained, is probably the biggest single cause of test automation failure.

Finally, if you are considering test execution tools, remember that test execution requires a test oracle. The test oracle is the means by which you determine the expected results. For the automated tests to provide useful results, you will need some accessible, usable test oracle, other than human judgment.

Let me illustrate by examples. A program that does calculations requires some other calculator as an oracle. In this case, you might be able to use Excel to generate the correct answers for predefined tests and store them for reference when the tests execute. For a program that implements complex business logic, if you wanted a fully automated oracle, you'd need one that does the same thing as the software under test, which is not likely to exist unless you are updating a legacy system. In this situation, you typically create output from the software under test and then revise it where needed to make sure it is correct. This kind of oracle is often called a baseline, and you can use the kinds of comparators discussed in the Foundation syllabus to compare actual results to the baselines. In the case where an oracle is available but it is slow compared to the software under test, this concept of stored baselines becomes useful again.

ISTQB Glossary

test oracle: A source to determine expected results to compare with the actual result of the software under test. An oracle may be the existing system (for a benchmark), a user manual, or an individual's specialized knowledge, but it should not be the code.

Now, if you wanted to run usability tests, keep in mind that human judgment is required to interpret the results. So, no useful oracle is available for automated usability tests, making automating such tests typically a waste of money.

That's a quick survey of various issues affecting test automation. Let's examine some of these issues in more depth, starting with the business case for automated testing. Remember, test automation should occur only when there's a strong business case for it, usually one that involves shrinking the test execution period, reducing the overall test effort, or covering additional quality risks.

When we talk about these benefits, notice that they are compared to the duration, effort, or coverage we would have with manual testing. You can only evaluate a return on investment in comparison to other alternatives.

9.2.1. Test Automation Costs

In any business case, we have to consider costs, risks, and benefits. Let's start with the costs. We can think of costs in terms of initial costs and recurring costs. Initial costs include the following:

- Evaluating and selecting the right tool. Many companies try to shortcut this and they pay the price later, so don't succumb to the temptation.
- Purchasing the tool, or adapting an open source tool, or developing your own tool.
- Learning the tool and how to use it properly. This includes all costs of intra-organizational knowledge transfer and knowledge building, including designing and documenting the test automation architecture.

- Integrating the tool with your existing test process, other test tools, and your team.

Recurring costs include the following:

- Maintaining the tool and the test scripts. This issue of test script durability—how long a script lasts before it has to be updated—is huge. Make sure you design your test system architecture to minimize this cost or, to put it the other way, to maximize test durability.
- Ongoing license fees.
- Support fees for the tool.
- Ongoing training costs; e.g., for new staff that come onboard or tool upgrades.
- Porting the tests to a new platform.
- Extending the coverage to new features and applications.
- Dealing with issues that arise in terms of tool availability, constraints, and dependencies.
- Instituting continuous quality improvement for your test scripts. Again, it's a natural temptation to skip this, but with a disparate team of people doing test automation, to do so means that the tool usage and scripts will evolve in incompatible ways and your reuse opportunities will plummet. I saw a client waste well over \$250, 000 and miss a project deadline because two automation people were creating what was substantially the same tool.

In the Foundation syllabus, there was a recommendation to use pilot projects to introduce automation. That's a great idea. However, keep in mind that business cases based on pilot projects will often miss important recurring costs, especially maintenance.

We can also think of costs in terms of fixed costs and variable costs. Fixed costs are those that we incur no matter how many test cases we want to automate. Tool purchase, training, and licenses are primarily fixed costs. Variable costs are those that vary depending on the number of tests we have. Test script development, test data development, and the like are primarily variable costs.

9.2.2. Test Automation Risks

You should also consider risks. The Foundation syllabus listed a number of such risks. Let's review them:

- Unrealistic expectations for the tool. This is extremely common, especially in organizations with limited previous experience with test automation. Because software engineering is so hard—including the testing part of the software process—people are always looking for silver bullets, magic solutions to make it simple. This desire for some simple answer makes people easily deluded—or self-deluded—into thinking that some new process methodology, new programming language, new management technique, or, yes, new tool will magically transform the hard problem of testing into one that is easy. Test tools do not do that. So make sure that people have proper expectations, including functionality and ease of use.
- Underestimating the time, cost, and effort required to introduce the tool. Some people assume that, because they saw a tool vendor give a 30-minute demonstration where a sales engineer tested dozens of screens or millions of lines of code, that's what will happen when they get the tool back to their office. A prepared, canned demonstration is not the same as scaling the tool up to an entire organization. In addition to the sizeable effort required to build test frameworks, integrate with other tools, set standards and guide-lines for use of the tool, and the like, you need to plan to train those who will use the tool. This can involve outside training and, often, external consulting.
- Underestimating the time and effort needed to achieve a positive return on investment. For most of our clients, positive return on the testing investment is something that takes years. And that's for the clients lucky enough to get a positive return. We've seen plenty who have not. A positive return on the test automation investment implies some mix of the following benefits: reduced overall test effort, reduced test execution periods, and coverage of additional, previously uncovered quality risks. A positive return on the testing automation investment implies significant and continuing benefits from the tool. A positive return on the testing automation investment usually involves changes in the testing process and requires continuous improvement in tool usage.
- Underestimating the effort required to maintain the tests. This is

actually a twofold risk. In many cases, organizations shortchange the initial test development and introduction effort, creating brittle, hard-to-maintain testware. I'll return to that topic in a moment when I address the risks listed in the Advanced syllabus. However, even organizations that escape this risk can find that, nonetheless, they fail to budget realistically for test maintenance. This leads to running tests that result in lots of false positives and false negatives, which seriously erodes the return on the test automation investment and again can lead to a death spiral, this time driven by the loss of credibility of the test results.

- Overreliance on the tool. A common mistake that people make with automation is summed up by the aphorism that, if you give a child a hammer, the whole world becomes a nail. Actually, the situation is a bit more nuanced than that, though it is certainly the case that people try to auto-mate unautomatable tests. What can also happen is that people fail to test areas that are not automatable, resulting in tool-driven test coverage gaps. That's an absurd outcome, of course, because test tools and their capabilities should not drive test coverage decisions.

In addition to the risks listed in the Foundation syllabus, there are these risks to consider:

- Your existing manual testing could be incomplete or incorrect. If you use that as a basis for your automated tests, you're just doing the wrong thing faster! You need to double-check manual test cases, data, and scripts before automating because it's more expensive to fix them later.
- You could produce brittle, hard-to-maintain test scripts, test frameworks, and test data that frequently needs updates when the software under test changes. This is the classic test automation bugaboo. Careful design of maintainable, robust, modular test automation frameworks, design for test script and data reuse, and other techniques can reduce the likelihood of this happening. If carelessness during the introduction period occurs, resulting in maintainability problems, those are test automation project killers, guaranteed. The test maintenance work will soon consume all resources available for test automation, bringing progress in automation coverage to a standstill. We have seen organizations discover, often years after starting test automation projects, that they have designed maintainability problems

into their test frameworks, test standards and guidelines, and test automation processes. These bad design decisions result in enormous, often escalating costs of test maintenance, usually leading to a death spiral where the return on the test automation investment becomes entirely negative and the effort is cancelled.

- You could fail to monitor the return on the test automation investment closely and continue to fund a vampire test automation project. In fact, this is the worst outcome for you as a test manager. Sooner or later someone will see the futility of it. At that point, other managers, including your managers, will call into question your management acumen. So, your failure to deal with a wasteful, resource-sucking test automation project can go beyond a risk to the test automation effort itself to become a risk to your job and your career.
- An overall drop in defect detection effectiveness could occur because everyone is fixated with running the scripted, invariable, no-human-in-the-loop automated tests. Automated tests are great at building confidence, managing regression risks, and repeating tests the same way, every time. However, the natural exploration that occurs when people run test cases doesn't happen with scripts. You need to ensure that an adequate mix of human testing is included.

As you can see, all of these risks can—and should—be managed. There is no reason not to use test automation where it makes sense.

9.2.3. Test Automation Benefits

Of course, the reason we incur the costs and accept the risks is to receive benefits. What are the benefits of test automation?

First, I must emphasize that smart test teams invest—and invest heavily—in developing automated test cases, test data, test frameworks, and other automation support items with an aim of reaping the rewards on repeatable, low-maintenance automated test execution over months and years. When I say, "invest heavily", what I mean is that smart test teams do not take shortcuts during initial test automation development and rollout because they know that will reduce the benefits down the road.

Smart test teams are also judicious about which test cases they automate, picking each test case based on the benefit they expect to receive from automating it. Brain-dead approaches like trying to automate every existing manual test case usually end in well-deserved tears.

Once they're in place, we can expect well-designed, carefully chosen automated tests to run efficiently and with little effort. Because the cost and duration are low, we can run them at will, pushing up overall coverage and thus confidence upon release. In other words, with the same total effort, we get better quality risk mitigation. To put it another way, less effort is required to reach the current level of risk mitigation, freeing extra effort to expand coverage (without affecting the schedule).

If the time required for test execution is primarily driven by the time required to execute the test cases—as opposed to the time required to fix the bugs found by the test cases—then automation can accelerate the test execution period and result in earlier delivery of software. Be careful with this benefit before you start touting it, though. Check to see if perhaps the turnaround time inherent in bug find/fix/confirmation test cycles is actually the main determinant of test execution time.

In addition, there are certain test types that cannot be covered manually in any meaningful way, such as performance and reliability. With automation in place, we can cover those, which again reduces risk.

Let me reiterate a point I made a moment ago, before we get giddy thinking about these benefits. Given the size of the initial investment, you have to remember that the benefits will take months if not years to equal the initial costs. Understand: In most cases, there is no shortcut. If you try to reduce the initial costs of introducing automation, you will create a situation where the benefits of automated test execution are zero or less than zero. Do the math yourself on how long it takes to reach breakeven in that situation.

So, above and beyond the benefits of saved time, reduced effort, and better coverage (and thus lower risk), what else do we get from test automation done well? For one thing, we have better predictability of test execution time. If we can start the automated test set, leave for the night, come back in the morning, and find the tests have all run, that's a very nice feeling, and management loves that kind of thing. For another thing, notice that the ability to quickly run regression and confirmation tests creates a byproduct benefit. Since we can manage the risk associated with changes to the product better and faster, we can allow changes later in a project than we otherwise would. Now that's a two-edge sword, for sure, because it can lead to recklessness, but used carefully, it's a nice capability to have for emergencies.

Because test automation is seen as—and really is—more challenging and

more esteemed than manual testing, testers and test teams find the chance to work on automated testing rewarding.

Finally, because of the late and frequent change inherent in certain lifecycle models, especially in agile and iterative lifecycles, the ability to manage regression risk without ever increasing effort is a key benefit of automated testing.

9.2.4. Test Automation Strategies

As I have said to clients on a number of occasions, a test tool by itself is not a test automation strategy. Let me give you a nonsoftware example of this. Where I live, in Central Texas, we have these huge, unsightly juniper bushes that people call cedars for some reason. They are not cedars. Cedars are noble, upright, useful trees. These juniper bushes are nothing but overgrown gin factories, and when they pollinate in the winter, three to four million noses and twice as many eyes in Central Texas start to water.

My family lives on 11 acres—about 4 hectares—of property. We are surrounded by junipers. My wife and I want to get rid of them. However, going down to Home Depot and buying a chain saw is not a strategy for clearing 11 acres of junipers, some of which are 10 to 15 feet tall. A chain saw is a tactic that will be useful in clearing 11 acres of junipers, but a strategy involves picking where to start, planning for keeping them down, figuring out disposal of the cut junipers, and the like.

So, here are some strategies for test automation. First and foremost, automate for the long term. Build a maintainable automated test system. Only automate those tests that are automatable. An automatable test can run unattended, and human judgment is not required during test execution to interpret the results.

Automate those tests and tasks that would be error prone if done by a person. This includes not only regression testing—which is certainly a high-value target for automation—but also creating and loading test data.

Only automate those test suites and even test cases within test suites for which there's a business case. That means you have to have some idea of how many times you'll repeat the test case between now and the retirement of the application under test.

Even though most automated tests involve deliberate, careful, significant effort, be ready to take advantage of easy automation wins where you find

them. Pick the low-hanging fruit. For example, if you find that you can use a freeware scripting language to exercise your application in a loop in a way that's likely to reveal reliability problems, do it.

That said, be careful with your test tool portfolio, both freeware and commercial. It's easy to find yourself in a situation where the set of tools used by your team is unmanageable and not well understood, especially if everyone downloads their own favorite freeware test tool. Have a careful process for evaluating and selecting test tools and don't deviate from that process.

To enable reuse and consistency of automation, make sure to provide guidelines for how to get tools, how to select tests to automate, how to write maintainable scripts, and the like.

When you're doing test automation for safety-critical applications, different rules can apply. For example, during consulting with one major defense vendor in a U.S.-allied country, my clients and I identified the following requirements that would affect their use of an automated test tool for unit testing:

- The tool would have to support—or at least be compatible with—the DO-178B standard and compliant with it.
- The tool would have to be able to handle the fact that the target system, an embedded system running on the aircraft itself, was not the same as the host system, a PC on which the programmers developed the code. The differences between the target system and the host system created various related test and development issues.
- The tool would have to work in an environment with lots of interrupt handling and be able to test interrupt handling.
- The tool would need to help us manage—or at least not exacerbate—the difficulty of creating realistic test environments.

We have other clients in safety-critical areas, some of whom are FDA regulated. Our FDA-regulated clients have different but similarly tight constraints on their testing and thus their test tools.

9.2.5. Case Study in Test Automation Strategy

A few years ago, we did an assessment for a client whose test team was

having trouble with automation. They weren't really aware of how bad it was, but they had a large, negative return on their test automation investment. In my report, I made two basic points:

- First, I told them that the current test automation approaches had achieved very limited success. Load testing delivered some value but was constrained by test environment issues. Functional regression testing was stuck in an endless maintenance mode and making no progress in automating any of the 99 percent of other data center applications that had big regression risks but zero automation coverage.
- Second, I told them that, in spite of a large investment in test automation over the previous four years—an investment in the millions of dollars—they had no demonstrable business value or return on investment. The only company to get a return on test automation at their shop, I told them, was the outsource company that had done a horrible job designing the unmaintainable automated test framework and were now charging them by the hour to maintain tests.

Due to the high regression risks I mentioned, abandoning automation was not an option. So, I proposed the road map for revitalizing test automation that you see in [figure 9-1](#). Let me walk you through it, starting at the upper left of the chart.

Figure 9-1. Test implementation road map



In the first week, define the benefits desired from automation, select key process indicators for automation, and create a model for calculating return

on investment.

In the second week, define the processes for deciding which tests to automate, along with a tool evaluation process and a dashboard for measuring the automation benefits.

In the third week, develop a short list of a few candidate tools using the needed characteristics for each tool enumerated in the tool evaluation process.

In the fourth week, using a live, on-site, realistic tool evaluation process, select the appropriate tools.

At this point, I told them, they could consider the test automation effort revitalized. A clear strategy for delivering real, measurable business value exists.

Now, in the fifth week, launch the revitalized test automation effort. Develop a detailed project plan. (Note that I showed on the diagram that they could accelerate this activity and begin as early as the second week.) Put test automation metrics gathering mechanisms in place. Define the processes by which automation will take place and by which automation will fit into the larger test process.

Also in the fifth week, launch the tools. Train the automated test engineers in the use of the tools. (I made clear in my report that they should select only outsource staff involved in test automation and with solid knowledge of and experience with the selected tools, as that had been part of their mistake before.) Configure the automated test execution environment. Create any other infrastructure needed for automation.

Shortly after the fifth week, the revitalized test automation effort would begin to produce automated tests. At that point, the work to develop, execute, and maintain tests starts.

9.2.6. Test Tool Integration and Scripting

Test tools can and should be made to work together to solve complex test automation projects. I'll give you an example of what I mean by that a little later.

In many organizations, multiple test and development tools are used. We could have a static analysis and unit test tool, a test results reporting tool,

a test data tool, a configuration management tool, an incident management tool, and a graphic user interface test execution tool. In such a case, it would be nice to integrate all the test results into our test management tool and add traceability from our tests to the requirements or risks they cover. In such situations, try to integrate tools and get them to exchange information.

Just because you bought a single vendor's test tool suite doesn't necessarily mean the tools included in the suite will integrate. They should, though, and you shouldn't buy suites with tools that don't.

If you can't get a fully integrated set of tools, you might have to integrate them yourselves. The extent of effort you put into doing this should be balanced against the costs and risks associated with moving the information around manually.

Lately, there have been many advances in integrated development environments. We testers can hope that this presages similar integration for test tools in the future.

Most test automation tools—at least those for execution—have the ability to use scripting languages. Typically, we are going to write the testing framework in these scripts and then use data or keywords in flat files, XML files, or databases to drive the tests. This separation of the scripts and the data used by the scripts supports maintainability.

In the companion volumes on test analysis and technical test analysis, we spend a significant amount of time on the issue of combinatorial explosions in testing. Manual testing doesn't deal well with combinatorial tests because we are overwhelmed by the effort. However, using scripting languages, we can often create tests that can cover more combinations than we could manually.

Some tools also provide the ability to go directly to an application's API. For example, some test tools can talk directly to the web server at the HTTP and HTTPS interfaces rather than pushing test input through the browser.

Scripting languages and their capabilities vary widely. Some scripting languages are like general-purpose programming languages. Others are domain specific, like TTCN-3. Some are not domain specific but have features that have made them popular in certain domains, like TCL in the telephony and embedded systems worlds.

Not all tools cost money—at least to buy. Some you download off the Internet and some you build yourself. In terms of open source test tools, there are lots of them. As with commercial software, the quality varies considerably. I've used some very solid open source test tools, and I've heard some stories about real garbage.

Even if an open source tool costs nothing to buy, it will cost time and effort to learn, use, and maintain. So evaluate open source tools just as you would commercial tools—rigorously and against your systems, not by running a canned demo. Remember, the canned demo will almost always work and establishes nothing more than basic platform compatibility.

In addition to quality considerations, with open source tools that have certain types of licenses, such as the Creative Commons and GNU Public License, you might be forced to share enhancements you create. Your company's management will want to know about that if it's going to happen. As a test manager, watch very carefully for use of open-sourced tools in your team, and insist that senior management or the company attorneys have a chance to vet licenses before putting such tools into service.

If you can't find an open source or commercial tool, you can always build your own. Plenty of people do that. However, it's a very expensive way to go. Also, since one or two people often develop these as a side activity, there's a high risk that when the tool developer leaves, the tool is dead. Make sure that custom tools are documented.

Finally, a caution: When testing safety-critical systems, there can be regulatory requirements about the certification of the tools used to test them. These requirements could preclude the use of custom and open source tools for testing such systems.

9.2.7. Case Study of Integrated Test Tools

Let's look at an example of an integrated test system architecture in [figure 9-2](#). This is an automated test system for an insurance company. The system under test—or, more properly, the system of systems under test—is shown in the middle.

Figure 9-2. Integrated test architecture



On the front end are three main interface types: browsers, legacy UNIX-based green screen applications, and a newer Windows-based Consolidated View. The front-end applications communicate through the insurance company's network infrastructure, and through the Internet, to the iSeries server at the back end. The iSeries, as you might imagine for a well-established regional insurance company, manages a very large repository of customers, policies, claims histories, accounts payable, accounts receivable, and the like.

On the right side of the figure, you see the main elements of the test automation system. For each of the three interface types, we need a driver that will allow us to submit inputs and observe responses. The terminal driver is shown in a dotted line because there was some question initially about whether that would be needed. The controller/logger piece uses the drivers to make tests happen, based on a repository of scripts, and it logs results of the tests. The test data and scripts are created using tools as well, and the test log analysis is performed with a tool.

Notice that all of these elements on the right side of the figure could be present in a single, integrated tool. However, this is a test system design, so we leave out the question of implementation details now. It is a good practice to design what you need first and then find tools that can support it rather than letting the tools dictate how you design your tests. Trust me on this one; I have the scars to prove it! If you let the tools drive the testing, you can end up not testing important things.

Let's move on now to the left side and bottom of the figure. In many complex applications, the action on the screens is just a small piece of what goes on. What really matters is data transformations, data storage, data deletion, and other data operations. So, to know whether a test passed or failed, we need to check the data. The data probe allows us to do this.

The pipe is a construct for passing requests to the data probe from the controller and for the data probe to return the results. For example, if starting a particular transaction should add 100 records to a table, then the controller uses one of the applications to start the transaction—through a Windows interface via the Windows driver, say—and then has the data probe watch for 100 records being added. See, it could be that the screen messages report success, but only 90 records are added. So we need a way to catch those kinds of bugs, and this design does that for us.

In all likelihood, the tool or tools used to implement the right-hand side of this figure would be one or two commercial or freeware tools, integrated together. The data probe and pipe would probably be custom developed.

9.2.8. Test Tool Classification

Let's look at various ways we could classify tools:

- In the Foundation, we grouped tools by the test activity they supported.
- We can also group tools by the level of testing they support; e.g., unit test tools, integration test tools, system test tools. This strikes me as quite a weak way, given the potential for reuse.
- We can group tools by the types of defects we are looking for with them.
- We can group tools by the type of test techniques they support; e.g., you might have seen pairwise and classification tree tools in the companion volume on test analysis.
- We can group tools by the purpose of tool.
- We can group tools by the application domain they are used to test, which is most useful for domain-specific rather than general-purpose test tools.
- We can group tools based on how they are applied.

- Finally, we can group tools based on the user, which is how we'll do it in the next section, "Test Tool Categories".^[5]

^[5] In that discussion about tools and their classifications, keep in mind that the Advanced syllabus augments the tools categories already introduced in the Foundation syllabus, along with introducing new tool categories. So, you'll need to refer to the Foundation syllabus as well as the Advanced syllabus for general information concerning the other tools categories not included in the discussion.

During initial deployment of automated test tools, it's easy to make some dumb mistakes that will come back to haunt you later. Let's review some important considerations during this critical launch period for automation.

It's important to remember that any automated tool you might use for testing is a form of software, typically a complex and sophisticated form of software. You will probably use it to solve a complex problem, and the solution will need to persist and even expand for years to come. Plan to document the architecture of the test system you build with the tool, and plan to test the tests you create with the tool.

Tools typically have hardware and software dependencies. Some integrate easily with other tools, while some are more stand-alone, which you should consider during your evaluation.

With commercial tools, support is often limited for nonstandard platforms. So if you need to support platforms other than Windows PCs or browser-based applications, you'll want to consider this issue carefully. You might find that an open source or custom-developed tool is the only path forward. Trying to adapt commercial tools for environments other than their intended ones doesn't often work well, and vendors tend to provide limited (if any) support.

When putting a tool in place, you might be tempted to automate every manual test case you have in the organization; that's usually not a good idea. If testers created the manual test cases specifically to be automated, that might work, but usually some modifications are needed, such as formatting the test cases, considering reuse patterns, expanding input and facilitating maintainability by using variables instead of using hard-coded values, and utilizing the benefits of the test tool. Test tools can often be used to navigate the test set, repeat tests, and vary the order of tests,

which might open new possibilities for the way the tests are implemented. Of course, the tool should provide better analysis and reporting facilities, and you'll want to exploit those.

You should remember that automated test engineering involves programming skills, among other types of skills. That doesn't mean that you can't build automated test frameworks that allow nonprogrammers to create tests—you can and you should—but building such frameworks is a programming task. To create a large, complex test framework is a large, complex task. If done improperly, you'll end up in the zero-ROI hell I mentioned earlier. People involved in test automation, especially those building the frameworks, need appropriate training in test tools, programming and design techniques.

Test automation is not a one-way street because the system under test is likely to evolve. So, just because your automated tests are running fine, that doesn't mean you are getting meaningful results. Be sure to audit the automated tests. This typically involves executing some rotating set of test cases manually to retain the knowledge of how the test works and to verify correct operation.

Another easy mistake to make is to think that the test automation problem you are solving today is the same test automation problem you'll have forever. Your needs will change, in part because the system under test changes and also because your awareness of the use of automation will change. Consider extensibility of tools. A tool with open application programming interfaces (APIs), standard scripting languages, and interoperability with other automated tools will extend more easily.

Finally, when doing your tool selection, whether going with commercial, open source, or home-built tools, carefully evaluate tool capabilities. Here are some areas to consider:

- What assistance can the tool provide with test analysis?
- What assistance can the tool provide with test design and implementation? Must you create the tests manually or are they automatically generated?
- What assistance can the tool provide with test selection? Again, is the process of selecting the tests to run purely manual or can the selection be done automatically according to a range of criteria?

- What assistance can the tool provide with test execution? Is the test execution purely manual or does the tool execute tests automatically? (While most test execution tools will run tests automatically, some test design tools cannot do that themselves and must interface with an automated test execution tool.) If the tests are run automatically, how much work is required to start them? Is intervention needed, and if so, why and when? Can the tests restart themselves under common error conditions?
- What assistance can the tool provide with test evaluation? Certainly, for test execution tools you need to have a test oracle. How about with the evaluation, analysis, and presentation of results?

Remember that a careful tool selection process will pay you back by saving you untold grief in the months and years ahead on a typical automation endeavor.

9.3. Test Tool Categories

Learning objectives

(K2) Summarize the test tool categories by objectives, intended use, strengths, risks, and examples

(K2) Summarize specific requirements for commercial test tools and open source test tools used for testing safety-critical systems

(K2) Describe important aspects and consequences of different test tools and their implementation, usage, and effects on the test process.

(K2) Describe when and why implementing your own tool is an option and its benefits, risks, and consequences.

Let's review the various types of test tools that are available to us, starting with test management tools.

9.3.1. Test Management Tools

Test management tools are, naturally, used to manage testing. Because testing generates information, there's a lot of information to manage, including the following:

- Traceability of test artifacts to the test basis, such as requirements, risks, etc.
- The capture of test environment data for complicated environments

ISTQB Glossary

test management tool: A tool that provides support to the test management and control part of a test process. It often has several capabilities, such as testware management, scheduling of tests, logging of results, progress tracking, incident management, and test reporting.

- Tracking of concurrent test execution, including when tests are running in different test environments at multiple sites
- Various test-related metrics such as these:
 - Test conditions
 - Test cases
 - The time required to execute test cases, test suites, regression test sets, and other test-process-describing metrics
 - The number of test cases, test scripts, test environments, and so forth
 - The pass/fail rates for tests
 - The number of blocked test cases (along with the blocking conditions)
 - Trends in various metrics like bug find/fix rates
 - Number and status of requirements
 - The relationships and traceability between test scripts, test designs, and so forth

Test management tools often have certain organizing concepts embedded in them, such as the following:

- Serving as central repository and driver of test cases and other test artifacts

- Organizing test conditions and test environments
- Storing and grouping tests cases for regression suites and test sessions
- Handling logging and failure information
- Instructions for environment restart and reinitialization

Test management tools are used by test managers, test analysts, and technical test analysts.

9.3.2. Test Execution Tools

Used properly, test execution tools should reduce costs, increase coverage, and/or make tests more repeatable. Because of the large amount of effort and tedium, test execution tools are often used to automate regression tests.

Most test execution tools work by executing a set of instructions written in a scripting language, which is just a programming language, customized for the tool. The tool usually gives precise ability to drive key presses and mouse actions and to inspect the graphical user interface or some other interface. That can lead to brittleness of the test scripts and expected results.

The scripts can be recorded using capture playback facilities or constructed like a real application. You can use capture playback for tracing the path of exploratory or other nonscripted testing, but the resulting scripts and expected results are very difficult to maintain.

Test execution tools use a comparator to compare the expected results—which were usually captured during some previous test run—with the actual results. Usually, you can instruct these comparators to avoid comparing fields that will vary, like dates and times.

Test automation scripts that are constructed like real programs tend to be built of libraries of functions or actions, which are then driven by tables of keywords or data. The keyword approach is called keyword-driven or action-word-driven test automation. The separation of the data from the scripts can overcome the maintainability problems created by capture playback.

Poor skills in programming and bad design of automation architecture can cause failure of test automation. There's also a need for careful

management. And, since the test execution scripts are programs, we need to remember to test them.

Test execution tools are mostly used by test analysts and technical test analysts.

9.3.3. Keyword-Driven Automated Test Execution

Let me describe this idea of keyword-driven or action-word-driven test automation. Keywords represent some business interaction with the system. For example, we could have a keyword for "cancel order", which would result in a whole workflow that made that happen. We can then string together keywords to create test procedures, which can correspond to a use case or a whole end-to-end scenario.

ISTQB Glossary

keyword-driven testing: A scripting technique that uses data files to contain not only test data and expected results, but also keywords related to the application being tested. The keywords are interpreted by special supporting scripts that are called by the control script for the test.

Inside the framework, the keyword is implemented as an executable test script using whatever tool was chosen. The scripts are modular. Sophisticated programming skills are required to implement such a framework. While expensive up front due to the cost of creating the framework, this approach offers a number of advantages.

First, domain experts, based on their understanding of the system, can define keywords. So, the tests can be written based on a user's view of the system, not the tool's or the test automator's view.

Once the framework is in place, domain experts, using the keywords along with data that they define, can write the test cases themselves. Once the test cases are done, anyone can run them. No real expertise is required, unless a test fails. In that case, the domain expert might need to isolate the failure to see if the problem is the system under test or—as is always possible—the test framework.

The modular scripts and test cases are easier to maintain than other, more linear approaches to automation. If one screen changes, there is usually just a single, reused script that must be updated.

The test specifications are independent of implementation, so you could port them to another implementation of the framework. We actually had a system like this years ago, where we tested various versions of a query tool that ran on about a dozen operating systems with three frameworks. It took a while to learn how to program the framework so it looked the same to the tests, but once we figured it out, we had no trouble moving the tests around.

Keyword test development is mostly done by domain experts and test analysts, though technical test analysts create the frameworks.

9.3.4. Case Study of Test Execution Objectives

Let's look at an example, shown in [figure 9-3](#). We helped one client implement test execution tools for unit testing. At the start of the project, we identified the objectives for the four main stakeholders in the engagement. As you can see, management has a number of major objectives:

- Reduce regression, both during system testing and after release.
- Increase confidence in changes at release.
- Deliver on time, with acceptable quality.
- Increase test and development efficiency.
- Reduce delays and costs of fixing regressions and other side effects of changes, especially by reducing regression issues during system test.
- Schedule acceleration, especially by reducing the number of bugs in system test.

The development team, being the primary users of the unit test tool, had a number of objectives too:

- Reduce regression via automatic regression testing for individual developers.
- Achieve a uniform standard of unit testing to make sure everyone agreed on what "done" meant for this task.
- Reduce regression test lab resource needs because the regression testing lab consumed a lot of resources and required a lot of developer support.

Finally, the test team, being downstream of the development team, had a few objectives:

- Increase percentage of stable test releases, as bad test releases ate much of their test time.
- Reduce regression bugs.
- Increase tester efficiency.

Finally, our objective was to enable client unit test project success. We wanted a referenceable client. Of course, we also wanted to get paid!

It's always good to start a test automation project—of any sort—with clearly defined objectives like these. A good next step is to define one or more metrics (often referred to as key process indicators in process improvement efforts) for each of these objectives.

ISTQB Glossary

test execution tool: A type of test tool that is able to execute other software using an automated test script, e.g., capture/playback.

Figure 9-3. Objectives for unit test tool



In addition to these objectives, a number of important aspects or constraints applied to this project. We needed a tool and a process that were both lightweight and easy to use. Creating and running tests had to be a minor, incremental task compared to the programmers' existing workload. Any sort of paperbound process would not be acceptable.

We needed a tool that would integrate with existing tools and environments. Specifically, this meant the development environment, Visual C++; the developer Windows desktops; and the TestDirector database.

Finally, given the nature of the project, we had to support regression testing at unit level. This meant high availability to developers as well as multiple regression tests running at the same time, on different "private" versions but with shared customization of the tests.

ISTQB Glossary

debugging tool: A tool used by programmers to reproduce failures, investigate the state of programs, and find the corresponding defect. Debuggers enable programmers to execute programs step-by-step, to halt a program at any program statement, and to set and examine program variables.

9.3.5. Debugging and Troubleshooting Tools

Debugging and troubleshooting tools can help us narrow down the area where a bug lives. In some cases, as with user interface bugs, the location of the bug is obvious, but in other cases the bug can be a long way from the symptom. Debugging tools can include logs, traces, and simulated environments. They can include debugging and tracing tools.

Debuggers allow a programmer to execute programs line by line, watching for unexpected control or data flows. Debuggers can halt the program at any program statement if the programmer has a hunch about where the bug lives or wants to check some variables at the point. Debuggers can set flags on and examine program variables.

Debugging is related to testing but is not testing. Similarly, debugging tools are related to testing but are not testing tools.

Debugging and troubleshooting tools are used mostly by technical test analysts.

9.3.6. Fault Seeding and Fault Injection

Fault seeding and fault injection are different but related techniques. Fault seeding uses a compiler-like tool to put bugs into the program. This technique is typically used to check the ability of a set of tests to find such bugs. Of course, the modified version of the program with the bugs is not retained as production code! This technique is also sometimes called mutation testing.^[1]

[1] While I've not used this technique, there are those who have. Voas and McGraw's book, *Software Fault Injection*, describes the technique in great detail and is a good place to start if you're curious.

ISTQB Glossary

fault seeding tool: A tool for seeding (i.e., intentionally inserting) faults in a component or system.

static analyzer: A tool that carries out static analysis.

static analysis: Analysis of software artifacts, e.g., requirements or code, carried out without execution of these software artifacts.

Fault injection is usually about injecting bad data or events at an interface. For example, we have a tool that allows RBCS consultants to randomly corrupt file contents.

Fault seeding and fault injection are used mainly used by technical test analysts.

9.3.7. Static Analysis Tools

You can use static analysis tools, which automate some parts of the static testing process, throughout the lifecycle. They provide warnings about potential problems with code, requirements, etc. For example, a code analysis tool will flag dangerous or insecure constructs. Running a spelling and grammar checker on a requirements specification can reveal too high a reading difficulty level.

The usual problem we've had when using these tools for clients is the number of false positives. In this case, a false positive is a potential problem that does not actually cause any damage. The number of false positives on an existing code base can be huge, as many as one for every 5 or 10 lines of code.

There are various strategies for working around this, like only using the tool on new and changed modules of code. Fortunately, vendors recognize this problem and are working to fix it.

Static analysis tools are mainly used by technical test analysts.

9.3.8. Dynamic Analysis Tools

Dynamic analysis tools provide runtime information on the state of the

executing software. They can be used to pinpoint a number of problems that are hard to find in static analysis and hard to isolate in dynamic testing. These tools include evaluating pointer use, but perhaps memory leak detection is the most common example. Memory leaks are particularly likely in programming languages like C and C++ where the programmer manages memory directly—because they sometimes mismanage it!

ISTQB Glossary

dynamic analysis tool: A tool that provides runtime information on the state of the software code. These tools are most commonly used to identify unassigned pointers, check pointer arithmetic, and monitor the allocation, use and de-allocation of memory, and to flag memory leaks.

Dynamic analysis tools are mostly used by technical test analysts.

9.3.9. Performance Testing Tools

Performance test tools typically consist of two major elements. One is a load generator. The other is a measurement and analysis component.

The load generator executes a script, which implements an operational profile. Sometimes these scripts are captured, though my experience is that they more typically are created. The script needs to be able to throw at the system under test whatever kind of data the system needs to accept.

When a mix of scripts is run under most performance testing tools, a complex mixture of simulated or virtual users can be pounding on the system simultaneously. In many cases, the tools are not pounding directly on the user interface but rather on a communication interface such as HTTP or HTTPS. While this is happening, the measurement component gathers metrics such as these:

- Numbers of simulated users
- Number and type of transactions generated by the simulated users
- Response times to particular transaction requests made by the users

Based on these metrics, you can produce various reports, including graphs of load against response times.

Performance testing is a complicated activity. There are a number of important factors to consider:

ISTQB Glossary

performance testing tool: A tool to support performance testing and that usually has two main facilities: load generation and test transaction measurement. Load generation can simulate either multiple users or high volumes of input data. During execution, response time measurements are taken from selected transactions and these are logged. Performance testing tools normally provide reports based on test logs and graphs of load against response times.

- First, do you have sufficient hardware and network bandwidth on the load generator host required to generate the loads? I have seen load generators saturate before the system under test did, which defeats the purpose.
- Second, is the tool you intend to use compatible with the communications protocol used by the system under test? Can the tool simulate everything you need to simulate?
- Third, does the tool have sufficient flexibility and capability to allow you to create and run the different operational profiles?
- Finally, are the monitoring, analysis, and reporting facilities that you need available?

While it's common for organizations to build simple load generators for reliability testing, they typically purchase performance test tools or use open source versions. The real tricky part—and where most of the work will be should you decide to build your own performance testing tool—is in the measurement and analysis piece.

Let me mention something at this point that you should keep in mind. Many performance-related defects are design problems. I have seen late discovery of serious performance problems doom a project. So, when performance is a key quality risk, be sure to use modeling and unit testing to performance test critical components rather than waiting for system tests.

Performance test tools are typically used by technical test analysts.

9.3.10. Web Tools

Web tools are another common type of test tool.

ISTQB Glossary

hyperlink test tools: *Not defined in the ISTQB glossary.*

One common use of these tools is to scan a website for broken or missing hyperlinks. Some tools will also provide a graph of the link tree, the size and speed of downloads, hits, and other metrics. Some tools will do a form of static analysis on the HTML to check for conformance to standards.

Web tools are used mostly by test analysts and technical test analysts.

9.3.11. Simulator and Emulators

Simulators, as those of you who have watched any movies about space flight know, provide a way to test in an artificial environment. We might want to do this type of test because some of the code or some other part of the system is unavailable, because the real system is too expensive to use in testing, or because testing in the real system is unsafe. For example, aircraft, spacecraft, and nuclear control software are usually tested in simulators before being deployed. I suppose you could say that the deployment constitutes the first test in the real environment.

Some simulators can be sophisticated, able to inject faults, produce reproducible pseudo-random data streams, and the like. My experience with testing in simulators has been, no matter how good they were, there were always problems we found when we went on to the real hardware. Timing problems and resource constraints and dependencies in particular are tricky to simulate.

An emulator is a type of simulator where software mimics hardware. My first encounter with an emulator was in my assembly language course in my first year of computer science at UCLA. Rather than allow us to write real assembly language that ran on their multiuser VAX machines—they were smart enough not to let us do that, as we would have killed those machines for sure!—they provided use with a emulator that looked like a dedicated assemble-and-test environment for our programs.

The potential advantage of an emulator is the kinds of elaborate testing that can be done—at least if the emulator includes it. The one UCLA gave us didn't have a lot of fancy support for testing as it was mainly designed to

prevent us from getting direct access to the VAX hardware. A well-designed emulator, created to support testing, could allow for tracing, debugging, recording, and other kinds of "watching the program run from underneath" activities that would be impossible in a real system.

ISTQB Glossary

emulator: A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system.

simulator: A device, computer program or system used during testing, which behaves or operates like a given system when provided with a set of controlled inputs.

Test analysts and technical test analysts, depending on the type of emulation required, use these tools.

9.3.12. Case Study of Custom Tool Development

[figure 9-4](#) is a graphic that shows the architecture of the end-to-end, system integration testing harness we built for the IVR system of systems project. I recommend that you study this figure for a moment. We used a commercial tool, SilkTest, to interface with the Windows GUI on the Customer Service Application host PCs. However, we also had two custom-created tools.

One was the IVR interface test tool. It could send phone signals (called DTMF tones) into the phone lines, simulating a user interacting with the IVR input interface. By use of special branded voice messages, it could also recognize whether the correct message was played. We had to build this tool from scratch, using some open source components like the scripting language TCL. No commercial tool of the appropriate scale (or within our budget) existed.

Another custom-created tool was the middleware that tied together the IVR interface tool with the GUI tool. We called this a "fake pipe". It allowed each side of the overall test system to send messages to the other. In that way, they could coordinate tests, including end-to-end functional and performance tests.

Figure 9-4. IVR testing tool

9.4. Sample Exam Questions

To end each chapter, you can try one or more sample exam questions to reinforce your knowledge and understanding of the material and to prepare for the ISTQB Advanced Level Test Manager exam.

Q: Which of the following is a risk associated with test management tools?

- 1.** Untrained staff misuse complex classification fields.
- 2.** False positives result when inappropriate tests are automated.
- 3.** Test managers gain insight into trends that span multiple projects.
- 4.** Programmers introduce new defects when modifying code to comply with coding standards.

Q: Identify all of the following topics that you could address in a test tool strategy document.

- 1.** Return on investment projections
- 2.** Selection of pilot projects for particular tools
- 3.** Best practices gleaned from other organizations using similar tools
- 4.** List of the programmers developing the applications under test
- 5.** Specific quality risk items for each application under test

Q: Which of the following applications is most likely to require a custom-built automated test execution tool?

- 1.** An accounting application running on Windows PCs
- 2.** A spreadsheet application running on a browser
- 3.** An email application running on Linux PCs
- 4.** An embedded application running on an iPod

Chapter 10. People Skills and Team Composition

"Most of us as managers are prone to one particular failing: a tendency to manage people as though they were modular components. It's obvious enough where this tendency comes from. Consider the preparation we had for the task of management: We were judged to be good management material because we performed well as doers, technicians, and developers. That often involved organizing our resources into modular pieces, such as software routines, circuits, or other units of work. The modules we constructed were made to exhibit a black-box characteristic so that their internal idiosyncrasies could be safely ignored. They were desired to be used with a standard interface.

After years of reliance on these modular methods, small wonder that as newly promoted managers, we try to manage our human resources the same way. Unfortunately, it doesn't work very well."

—Tim Lister and Tom DeMarco, from the introduction to Part I of *Peopleware*

The 10th chapter of the Advanced syllabus is concerned with people skills and test team composition. The chapter starts with the skills of the individual tester, then moves to internal and external test team dynamics. It concludes with discussions of motivating testers and test teams and with communicating testing results. [Chapter 10](#) of the Advanced syllabus has six sections:

- 1.** Introduction
- 2.** Individual Skills
- 3.** Test Team Dynamics
- 4.** Fitting Testing Within an Organization
- 5.** Motivation
- 6.** Communication

Let's look at each section and how it relates to test management.

10.1. Introduction

Learning objectives

Recall of content only

This chapter is focused primarily on test management topics related to managing a test team. Thus, it is mainly the purview of test managers. As test managers, there are a number of issues we need to address to wind down this course:

- Individual skills needed by test professionals
- Test team dynamics
- Test team organization
- Test team motivation
- Communication issues

In your test teams, it is important for all testers to be mindful of their relationships not just with their fellow testers but also with all the members of the project team while doing test work.

10.2. Individual Skills

Learning objectives

(K3) Use a given questionnaire in order to determine strengths and weaknesses of team members related to use of software systems, domain and business knowledge, areas of systems development, software testing and interpersonal skills.

As test managers, we need to have a team with the appropriate balance of skills. That might sound obvious, but it's amazing how many completely unqualified test practitioners there are.

In fact, while writing this very material, we had to deal with a dissatisfied customer, a test practitioner who was returning books because he didn't find the templates he wanted in them. There were templates in those books for test plans, test cases, and test procedures, but apparently these templates still required too much thought and skill for this person. He wrote in an email, "I will be returning both books. I was hoping to see test plans, test cases and test procedures in template [form] so an individual could just plug in there [sic] own company's test items in to a preformatted template that would just be like 'fill in the blank fields' and you have a test plan or test case or test procedure. These books did not provide that. I'm a defense contractor ..."

Think about that. The safety of the soldier, sailor, or marine depends on military gear that actually works, and most of it has software in it. This fellow apparently couldn't even be bothered to read the books to try to learn how to use the extensive templates we provided. Oh, no, too much thought involved.

As test managers, we can achieve the right balance of skills in a couple of ways. One is by trying to ensure even distribution of skills across the entire team, which is referred to as the *team of generalists* approach. We can also achieve that balance by allowing individuals to have particular strengths in some needed skills, with the team collectively strong in all necessary areas, which is referred to as the *team of specialists* approach. There are innumerable variations on these approaches, as you can imagine. Whichever way you choose, a technique called skills inventories can help you manage this process.

Tester skills grow through a combination of experience and training. Using experience alone to grow your team is inefficient and leads to subtle weaknesses in your individuals and your team as a whole. Using training alone to grow your team is ineffective and leads to a team with strong theoretical knowledge but little ability to apply the theory to real testing problems. The ideal approach is to use targeted training followed by immediate application of the training concepts in real-world experience.

When thinking about the skills to grow in your individual testers and across your team as a whole, keep in mind these three main areas of tester skills:

- Use of software systems, including knowledge of the domain or business. In other words, does the tester understand the problem that the system must solve?
- Understanding of software development, including analysis, development, technical support, and technical documentation. In other words, does the tester understand the practical issues related to creating, deploying, and supporting systems?
- Understanding of software testing. In other words, does the tester know how to apply testing best practices to the problems they will face?

Let's review each area in the next three sections.

10.2.1. Testing Skills

All testers, whatever their role, need good interpersonal skills. They should be able to effectively give and receive criticism. The nature of the test job requires a level of emotional toughness due to the pressures of the job and the fact that we are often the bearers of bad news, but that toughness must not cross the line into sadism, cynicism, or callousness.

A tester, especially a test manager, plays a role within a project team. To get work done and to have your findings accepted and used appropriately, a degree of skill in influencing and negotiating is typically required.

Many testers in the business are technically competent—indeed, some are quite technically brilliant—but find themselves hampered on their projects and in their professional success by their poor interpersonal skills. If you find yourself often ignored, involved in heated arguments with other project team members, viewed as an adversary, kept in the dark about important project details, put on people's email blacklists, or stymied in your career

advancement, look at your interpersonal skills critically and honestly assess your role in your career problems.

Testing is about generating information, and it tends to generate a lot of it. Testing often involves generating, editing, reviewing, absorbing, and analyzing a lot of technical documents and metrics. Therefore, testers need good organizational skills; this is especially true for disorganized and chaotic projects.

Testers must see the broader aspects of the project but at the same time focus on important details. I've known a number of "big picture" types who were unsuccessful in their work because of poor attention to detail. I've also known a number of "detail oriented" types who were unsuccessful in their work because they picked the wrong details to focus on. Don't fall into the stereotype of the tester as the beside-the-point, nitpicky distracter. These focus skills are especially important for test managers.

Again, because testing generates information, your ability to be effective as a tester is only as good as your ability to convey that information. You need strong written and verbal communication skills. These skills become especially important for test managers. Above and beyond the essential people skills, for test analysts and technical test analysts, a number of specific testing skills apply:

- They should be able to read and analyze a requirements specification, a design specification, a use case, and other information about the system.
- They should be able to participate in product and project risk analysis activities. In fact, I often find that testers are the ones organizing their activities.
- They should be able to apply the concepts discussed in the Foundation and Advanced syllabi to design an effective and efficient set of tests. These skills include creating test data, selecting test oracles, defining expected results, and, in some cases, participating in test automation.
- They should be able to run tests, evaluate the results, and record those results in a meaningful way. These skills include logging of test status, test incidents, and test-impacting events. It also includes capturing metrics.

For test managers, a number of specific testing skills apply:

- They should be able to manage a project, because testing is often a sub-project within a larger project context. Test managers should have the skills required for all the usual project management tasks.
- They should be able to carry out the various activities related to test planning discussed in [chapter 3](#). These skills include risk analysis, test estimation, and writing test plans. It also includes formulating effective test strategies and helping to set the organization's test policy.
- They should be able to track test progress. These skills include capturing the appropriate metrics and logs. It also includes the ability to analyze those metrics, evaluate variance from the project plan and the test plan, and suggest and implement appropriate test control activities.
- They should be able to report test results. These skills include creating, presenting, and explaining the kinds of charts and reports discussed in [chapter 3](#). The ability to effectively communicate the results to project stakeholders is critical. Without this ability, all the work done in planning, creating, and running the tests can be wasted when the project team rejects or mis-understands the findings.

For test analysts and technical test analysts on a path toward test management roles, it's important to start cultivating these skills early, prior to the promotion into such a role. Otherwise, the individual contributor promoted into manager is at risk of the kind of failure DeMarco and Lister mention in the opening quote of this chapter.

10.2.2. Technical and Software Skills

There is a myth among some testers that we should not know how to program and should not have strong technical skills. The theory is that, when testers start to understand the mechanisms by which bugs become failures, they will become more tolerant of failures.

Now, it is true that testers over time, with repeated exposure to a system, will become more tolerant of its failures. I refer to this as acclimatization. It happens even with the most nontechnical testers. Your job, as a test manager, is to watch for and prevent acclimatization. In my experience, technical knowledge does not cause acclimatization. What it does cause is a greater insight, a greater degree of effectiveness, and a greater ability to communicate with programmers about failures and defects.[\[2\]](#)

□ My view is not shared universally. While reviewing this material, senior RBCS associate Judy McKay commented, "This is interesting because I have found that the more technical people tend to find creative workarounds to bugs that would completely baffle the user. I like to have some pure black-box folks to make sure I don't get into the situation where my team is too technical and forgets to identify with the user."

Testers who understand requirements analysis, design, and coding have insight into the bug lifecycle. In other words, they know how programmers introduce defects, where they can detect defects, and, better yet, how to prevent the introduction of defects in the first place.

Testers who understand technical support issues have insight into the user experience, user and customer expectations, and usability issues. In fact, we have clients that populate their test teams almost exclusively with former technical support staff. That staffing technique works well when the erstwhile technical support people are given the skills needed in other areas. When they are not, they tend to retain the firefighting, solve-the-problem-immediately attitude of technical support staff, which can create real problems for the test team.

Finally, remember that, as discussed in [Chapter 9](#), many test automation tasks require software development experience. So make sure to bring people into your test automation team that have the appropriate knowledge and experience here.

10.2.3. User, Business, and Domain Skills

At the risk of stating the obvious, people who spend their time getting real work done with the system have the essential perspective of the user. They know how to use and operate the system. They have a good understanding of failure priority, which helps during both quality risk analysis and incident management. They have a clear grasp of what "correct behavior" looks like, or at least what they want it to look like, which makes them and their opinions useful inputs for test oracles.

Business and domain experts, whether users or not, also have valuable insight. They know the importance of various features to the business. They know how certain behaviors would affect the business. Again, this makes them valuable participants in both quality risk analysis and incident management.

In addition to quality risk analysis and incident management, both users and domain experts can help with various other test activities. They can help to prioritize testing activities, particularly if various project constraints make it difficult to run tests in purely risk order. They can provide or help to create realistic test data and test cases. They can verify or, better yet, supply use cases.

Of course, as a test manager, you want to draw on the abilities of users and domain experts sparingly. For one thing, these people usually have other work assignments, and participating in testing tasks distracts them from that real work and might prove irritating to them. I made some political trouble for myself early in my career as a test manager by relying too heavily on domain experts who worked for another manager, leading to complaints that I was practicing "vampire management" and also neglecting the skills deficits in my team.

For another thing, users and domain experts sometimes feel they have the right to dictate to members of the project team. In that case, without careful management of the involvement of these folks, you can easily find yourself in a situation where outside staff and managers are second-guessing, micromanaging, and commanding your test effort. As a manager, you must know how to tactfully solicit help without abdicating ownership of your team.

10.2.4. Skills Inventory and Management

Okay, so it's fine to agree in general about the individual skills required for testers. However, how can you as a test manager measure, manage, and grow these skills? We will look at multiple figures providing examples of how to do it. These figures show a skills inventory. It is an example skills inventory based on a hypothetical project to develop a browser-based word processor.

Figure 10-1. Skills inventory and management—general qualifications

In this spreadsheet, you see a list of skills down the left column. [figure 10-1](#) shows those skills associated with general professionalism. The next three figures show testing skills, domain skills, and technical skills. Again, this list is an example for a hypothetical project. You'd need to customize the critical skills list based on a task analysis of what your team actually does and what they need to be able to do.

The legend across the top helps explain the entries in the body of the spreadsheet. We rate the skill levels required for various positions, along with the skills of the current team, on a four-point scale:

- 0 indicates someone has no knowledge or experience in the skill area.
- 1 indicates that a person has some knowledge and experience in the skill area but needs to be able to ask people questions to do work that requires this skill.
- 2 indicates that a person has significant knowledge and experience in the skill area and typically does not need to ask questions to do work that requires this skill.
- 3 indicates that a person has expert knowledge and experience in the skill area and would be the kind of person people would come to in order to ask questions to do work that requires this skill.

There are also two other types of notations on the spreadsheet:

- R indicates that a skill (at a particular rating, typically) is required to

hold a given position. D indicates that the skill is desirable but not required. A lot of confusion happens in the hiring process when people don't distinguish between required and desired skills, so be smart and keep those distinct in your mind.

- TM indicates test manager, TA indicates test analyst, and TTA indicates technical test analyst. I am using those position titles in the sense in which they are meant in the Advanced syllabus.

Okay, so now look down at the skills list again, on the leftmost side. In the two columns immediately to the right of the skills list, we have the headings Test Analyst (TA) Minimum Ratings and Technical Test Analyst (TTA) Minimum Ratings. In these columns, for each critical skill, we establish the minimum skill rating required. For most skills, we use the four-point scale from 0 to 3 mentioned above, but the "years of experience" ratings are, of course, in years.

To use these columns, you review your list of critical skills one by one and ask yourself, "What rating is required to hold this position in this skill area?" The temptation will be to enter the ideal value. That's not realistic. Set the minimum value, the level of skill at which you would not hire someone if they didn't reach it. Again, remember to distinguish between required and desired skills.

In addition, during this part of the skills inventory process, be prepared to suspect that your existing team does not meet the minimum requirements. Don't worry; that doesn't mean you have to fire a bunch of people and start over. What it means is that you'll need to undertake some serious skills growth work with your team.

Now, in the three columns in the middle of the figure, you'll have exactly one column for each person in your team. Rate each person in your team, again using the scale discussed earlier. Be fair, but be honest. All parents rate their children as above average in some way or another, but this is not the time for wishful thinking.

You'll notice that I have included the test manager in this example. I would suggest that you include yourself as the test manager if you might be a source of internal team knowledge transfer. In other words, if you have in the recent past done a lot of testing work—actual individual practitioner stuff—and you have strong expertise in that area, you might be able to spend some time teaching people what you know, even if you don't do much

hands-on work personally anymore. However, don't make a fool of yourself by pretending to know something you don't or by touting obsolete knowledge; you might end up attempting to "train" people in skills that they know or in skills that are no longer pertinent to the job.

With these columns populated, you will now see the two columns on the right side meaningfully populated. The team minimum indicates areas where at least one member of your team is weak. This indicator is interesting if you intend to have a team of test generalists because weaknesses mean that there are one or more tasks that require a particular person, or at least that can't be done by one or more particularly weak people. If you want a team of specialists, you should look at the team maximum because you want at least one person at the expert knowledge (3) rating for each skill.

The team average again is a generalist-focused metric that tells you, on average, where your team stands. You'd like an average rating of knowledgeable (2) in a generalist team across all skills. For a specialist team, you'd probably like a metric such as a count of the number of people with an expert knowledge (3) rating for each skill. In other words, how many experts do you have in your team?

Having done this assessment, review your team and identify weaknesses. Then, put a plan in place to address those weaknesses via training and job assignments. Periodically reassess your team skills, update this inventory, and revise your plan. This is an ongoing job for a test manager, not a one-off project. Plan to do this at least every six months.

[figure 10-2](#) shows the skills list for the testing skills. You might notice a couple of errors in this sheet. For example, the technical test analyst is the one who should have test automation skills, not the test analyst. These kinds of mistakes are easy to make in a worksheet like this, but they also tend to be self-correcting because they will jump out at you when you inspect the worksheet. You'll notice them when you do the skills analysis with the team.

Figure 10-2. Skills inventory and management-testing skill

figure 10-3 shows the skills list for the domain skills. Again, you might find you need to make corrections in this worksheet. For example, the domain knowledge for a test analyst should usually be higher than that of a technical test analyst. You will find and correct these problems when you do the actual skills inventory.

Figure 10-3. Skills inventory and management–domain knowledge

Finally, figure 10-4 shows the skills list for the technical skills.

Figure 10-4. Skills inventory and management–technical expertise

10.2.5. Individual Skills Exercise

As the HELLOCARMS test manager, you want to assess your test team. How would you update the skills list shown in the previous example to assess the team for this banking project? List the skills that would remain on the list and the ones that would change.

If you are working in a classroom, break into groups of three to five. If you are working alone, you'll need to do this exercise yourself. If you are working in a classroom, once each group has finished, discuss the results.

I suggest allotting 30 minutes for this exercise, including the discussion.

10.2.6. Individual Skills Exercise Debrief

In [Table 10-1](#), I have shown the list of skills from the figures, in the four major areas.

Skills that I would delete are shown in strikethrough text (like this). Skills that I would add are shown in bold text (like this). Skills neither underlined nor struck are those that remain unchanged across the teams. I have added explanatory comments where I felt they were needed.

You'll notice that, outside of the domain expertise area, the changes are minor or nonexistent. This happens because the testing skills required to test an application are relatively constant across all applications and because of the common technology behind all browser-based applications, be they word processors or banking applications.

Table 10-1. Skills list modified for HELLOCARMS project

General Qualifications	
Skill	Explanatory Comment
<i>Education</i>	
Bachelor Science Degree (or Advanced)	
Test Training or Certification	
<i>Work Experience (Years)</i>	
Test Roles	
Non-Test, Computer	
Non-Computer, Domain	
Non-Computer, Non-Domain	
<i>Professionalism</i>	
Oral Communication	
Written Informal Communication	
Written Formal Communication	
Continuing Education	
Test Team Building/Cross-training	
Cross-functional Relationship Building	
Reading (Retention, Reasoning, and Analysis)	
Business/Technical Trends	
<i>Testing Skills</i>	
Skill	Explanatory Comment
<i>General</i>	
Testing Standards and Maturity	
Version Control and Config Management	
<i>Planning</i>	
Estimation and Cost of Quality	
Documentation	

Quality Risk Analysis/Management	
<i>Design and Development</i>	
Behavioral (Black-box)	
Structural (White-box)	
Static (Reviews and Analysis)	
Performance (Modeling/Simulation/Testing)	
<i>Test Automation</i>	
COTS Execution Tools	
COTS Test Management	
Test Data Generators	
<i>Execution</i>	
Manual (Scripted and Dynamic)	
Automated	
Test Status Reporting and Metrics	
<i>Domain Knowledge</i>	
<i>Skill</i>	<i>Explanatory Comment</i>
<i>Word Processing</i>	Replace this section with the one immediately below
Windows Applications	
Linux/UNIX Applications	
Macintosh Applications	
Graphics, Figures, and Tables	
<i>Banking</i>	
<i>Home equity loans</i>	
<i>Reverse mortgages</i>	
<i>Lines of credit</i>	
<i>Mortgages</i>	
<i>General branch applications</i>	
<i>Document Management</i>	Replace this section with the one immediately below

Windows Applications	
Linux/UNIX Applications	
Macintosh Applications	
<i>Credit</i>	
Decisioning systems	
Credit bureau	
<i>Document Interchange</i>	Replace this section with the one immediately below
Windows Applications	
Linux/UNIX Applications	
Macintosh Applications	
<i>Regulations</i>	
Sarbanes-Oxley	
Data privacy	
State regulations	
Printing	Delete this section
Color vs. BW	
Laser, Inkjet, Other	
Publishing/Binding	
Web Publishing	Delete this section
HTML	
PDF	
<i>Technical Expertise</i>	
<i>Skill</i>	<i>Explanatory Comment</i>
<i>Programming</i>	
C/VB (3GL)	Pick one depending on HELLOCARMS
Java/C++ (OO)	
Shell Scripting	
Code Complexity and Metrics	
<i>Operating Systems</i>	
Windows	
Linux/UNIX	
Mac OS	Not used in the call center

<i>Networking/Internetworking</i> TCP/IP, FTP, RCP (Internet Architecture)	Might focus on the Globobank standards and architecture for these skills
Browsers (Firefox, Internet Explorer, etc.)	
Network Application Architecture	
Network Hardware	
<i>Systems and Servers</i>	
Web/Application Servers	
Database Servers	

10.3. Test Team Dynamics

Learning objectives

(K3) Perform a gap analysis in order to determine the required technical and soft skills for open positions in an organization

As a test manager, test team development is probably the most critical testing process of all. If you develop your team properly, you can often survive other classic test manager mistakes because your strong team will cover for you. For example, if you fail to engage with key project stakeholders to improve the quality of your team's bug reports—which is a classic test manager mistake—with a strong test team the individuals will attend to that detail for you.

Conversely, if you fail to properly grow and develop your test team, even if you do many other things well, the odds are good that your team will suffer serious problems. Hiring just one person you should not have hired can do major damage to the effectiveness of your test team.

New test team members are often hired, of course, but they can also be transferred from within the organization. However, I've come to distrust this avenue of team growth. If someone is being sent to your team from some other team, there's a reason. It could be that it's truly a better fit in terms of skills, and that could better for everyone; that's the ideal case. More likely, in my experience, though, is that someone is being sent to another team because some other manager wants to rid themselves of a problem employee without going through the rigmarole associated with firing someone. You should insist on applying the same rigorous standards to accepting inbound employee transfers as you would to hiring a new person.

The issue of the right team members is not only one of individual skills. You also must consider team dynamics. Using the skills inventory discussed earlier, it's easy to determine how to select a person with complementary skills, someone who can help fill weak spots in your team and cross-train others.

However, you also need to consider personality type. For example, do you need someone more tenacious on your team to chase down questions on requirements, borderline bugs, and so forth? In that case, don't hire a

person who is clearly introverted and shy.

The best test teams have a variety of personality types. As I mentioned earlier, you can also build a team that has a variety of specialized skills, as an alternative to a team of generalists.

As a test manager, you should see the whole test team development process as an ongoing effort to build the best possible test team. The team should be suitable for various projects. In addition, they should work well with the various project team members with whom they will interact.

When you add people to the team—whether by hiring them or by transferring them from within the organization—you want to ensure that the experience of joining the team is positive from the beginning. When I say "positive", I mean for the new team member, for you as the manager, and for the rest of the test team.

This requires a period of close interaction with you, the test manager, so they get good direction and adequate supervision. All too often, managers take a "throw the new, warm body at the fire" approach to new team members, giving them no clear direction and no attention. Don't make this mistake.

In addition, you need to define the new hire's role. Give them assignments they can handle, not ones that they will fail at. There's a time and a place for "stretch goals", particularly in very dynamic workplaces where part of carrying your weight on the team involves taking on challenging assignments, but I would suggest that the first 90 days, give or take, should be a period where you give people assignments that are clearly within their current abilities. You can determine this based on the skills inventory.

In this day of hardheaded, tough-minded management, some might call this approach coddling and soft. I disagree. My experience is that individual success is synergistic with team success. Conversely, widespread individual failure can't help but affect the team.

I have noticed a trend toward callousness and management by fear over the last decade. I have heard managers—ranging from executives on down to line managers—make comments that indicate that everyone is expendable and if they don't like someone's work they'll just get rid of that person. Sometimes it seems as if they don't take into account that the individual contributors have families to feed, aspirations to achieve, and lives to lead. And then those same managers are shocked to find that no one has any

loyalty anymore and that people don't put forth their best efforts on behalf of the team.

At RBCS, we have had good luck with most of our employees and associates. We have had test teams that would—and did—walk across a lake of fire for our clients and us. I like to think that this comes from a supportive style of management.

So, in my opinion, when you're a test manager, being a good leader is part of the job. In addition to being a supportive manager, though, you have to put people on the proper career path. This individual success arises from a proper match of personality and skills to the roles, the work assignments, you give.

Of course, you can't just focus on building your team's skills as a test manager. You have real work to do with your team. So be sure to enable team success by building the overall set of individual skills to match the test team's mission. Remember, the test policy—discussed in [chapter 3](#)—should define the test team's mission. If you don't have a policy—either written or verbal—that is supported by management, you can hardly grow the right team because there's no definition of what the right team is or of what the right team could, or should, do.

Sometimes, when RBCS helps clients with staff augmentation and employee search work, we find that people are looking for the perfect employee. Ten years of Java programming experience, 15 years of test automation experience, 5 years of domain experience, and, oh yeah, by the way, willing to take an entry-level position and work 65 hours per week.

The problem lies in part in something I mentioned earlier, which is the failure to distinguish between the required and the desired when defining the skills and traits needed for the job. Now, some people say, "Hey, why not set the bar really high, as that will discourage the clearly unqualified and make those who are qualified feel they have to stretch to meet my expectations?"

Wrong. That's not what happens when you have unrealistic job requirements. What happens is that plenty of qualified people take your job requirements seriously and choose not to try for the job, thus depriving you of the option of considering them. In addition, plenty of people who are unqualified look at the unrealistic job requirements and say, "That's silly, no one meets those requirements, so I'm just as qualified as anyone else, I

might as well apply," and now you get to waste your time weeding out these applicants.

So, set realistic standards and hire for those. Notice that I'm not arguing that you settle for incompetence or, worse yet, troublemakers. What I am saying is that you will have to grow some skills in your team members that, in an ideal world, all incoming candidates would have.

Of course, every now and then the perfect individual is available. My business partner and I recently hired someone at RBCS who exactly fit our dream idea of what they would be. However, that happens just often enough to remind us that "seldom" is not "never".

You are going to build a test team out of people who are a good fit, not a perfect fit. Over time, though, you are going to perfect your test team. In part, this involves good hiring decisions, building a strong team through balance.

What skills, experience, and attitudes does each new hire bring to your team that was missing before? You should know the answer to this question for each person you hire. How can this person cross-train the rest of the team to transfer those skills and experiences to their colleagues? You should know the answer to this question for each person you hire. Where is this person weak? How do you intend to address those weaknesses? You should know the answer to these questions for each person you hire.

In [chapter 8](#), we discussed continuous test process improvement. Notice that now we're discussing continuous test *team* improvement, which is at least equally important, and in some cases will be more important.

In the next four figures we'll continue the example from the previous section, this time using the skills inventory to do a gap analysis for a hypothetical test team. The team consists of just three people: Irit, Charlotte, and James. However, it's sufficient to demonstrate the technique.

Remember that the skills listed on the left side of this worksheet are those required to carry out important testing tasks in this team. They are grouped into four categories: general qualifications, testing skills, domain skills, and technical skills. We are going to rate each team member on our four-point, 0 to 3 scale and use those ratings to identify where we need to improve the team's skills. Remember that the rating of 2 indicates basic, autonomous competence with the skill in question.

Let's assume we're trying to build a team of generalists. In that case, we would want to evaluate closely two specific situations:

- One or more individual scores less than 2 on the skills rating. The "minimum" column at the right side of the worksheet will reveal this problem if it exists.
- The team as a whole scores only at or slightly above 2 on the skills rating. The "average" column at the rightmost side of the worksheet will reveal this problem if it exists.

Let's evaluate those situations on the worksheet shown in [figure 10-5](#).

In [figure 10-5](#), the test analysts, Charlotte and James, are weak on business and technical trends. Some increased time spent studying these trends would help. Perhaps Irit, the test manager, could assign each of them to research and report on a trend affecting this work? The other areas, where the team average is 2, do not seem troublesome because of the even distribution of skills. The next three figures will show similar comparisons for the testing skills, domain skills, and technical skills.

Now, what can we do with this skills inventory and gap analysis? For one thing, you can generate job descriptions for hiring directly from these worksheets. You can list the required skills from the worksheets and in particular emphasize the skills where the team is currently the weakest. For another thing, you can generate a training and cross-training plan from this worksheet, starting with the weakest areas.

Let's see where we have gaps in terms of test skills. In [figure 10-6](#), we can see that the team is quite weak compared to where we would like it to be. Before we start having Charlotte and James do research on industry trends, Irit should sort the list of skills on this worksheet by importance and have a plan in place to

Figure 10-5. Gap analysis using inventory-general qualifications

Figure 10-6. Gap analysis using inventory-testing skills

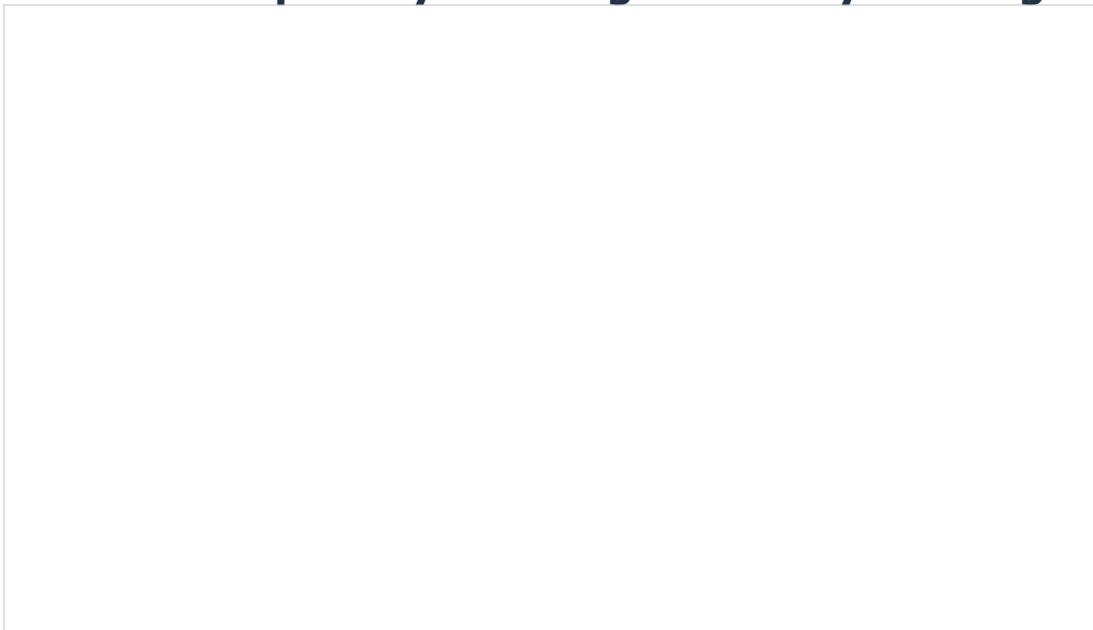


Figure 10-7. Gap analysis using inventory-domain knowledge



bring these ratings up, systematically and across the board, over the next six months.

Let's see where we have gaps in terms of domain skills. Here in [figure 10-7](#), we see what appears at first to be an equally bleak picture. However, notice that many of the skills are ranked as "desirable" rather than "required". So in terms of domain skills, Irit should plan to address the areas of serious ignorance—team minimums or worse yet averages of 0—or systemic ignorance—team averages around 1—in the coming year and then gradually, over the next couple years, increase the other ratings.

Let's see where we have gaps in terms of technical skills, shown in [figure 10-8](#). The evaluation here is less bleak than for domain knowledge. In fact, it is considerably less so when you see that these requirements fall mostly on James, the technical test analyst. In addition, it's likely the case that Irit, the test manager, can accept basic familiarity (signified by the 1 rating) as opposed to autonomous competence (signified by the 2 rating) because project team stakeholders probably expect a lower level of skill in this area for testers.

Figure 10-8. Gap analysis using inventory–technical expertise

So, perhaps a plan to gradually infuse better knowledge of the Mac OS into the team, along with giving Charlotte basic familiarity with various programming concepts, will suffice.

10.3.1. Test Team Dynamics Exercise

If you are working in a classroom, break into groups of three to five. If you are working alone, you'll need to do this yourself.

Self-assess yourself and others in the group against the skills list from the last exercise. Use that self-assessment to identify gaps in the team. Outline an approach to fill the gaps.

If you are working in a classroom, once each group has finished, discuss the results.

I suggest allotting 30 minutes for this exercise, including the discussion.

10.3.2. Test Team Dynamics Exercise Debrief

The specific results of this exercise depend on the team. However, in general, a manager has a number of options to fill a skills gap in a team:

- Hire someone with the skill. Going forward, have that person serve as the specialist, handling those tasks that require the skill.
- Hire someone with the skill. In the short term, have that person serve as the specialist, handling those tasks that require the skill. In the long term, arrange for that person to cross-train other team members.

- Bring in a consultant with the skill. Use that person to handle those tasks that require the skill. (This makes sense only when the need is temporary and nonrecurring.)
- Bring in a consultant with the skill. Use that person to handle those urgent tasks that require the skill. Before releasing the consultant from the engagement, arrange for the consultant to cross-train other team members. (This might not strike the consultant as being in his best financial interests, so be clear about it up front and make the knowledge transfer a material part of the contract.)
- Outsource the work to a company with the skill. Use that company to handle those tasks that require the skill. (Outsourcing often makes sense when there are various economies of scale or when the skill has a particularly long learning curve or is difficult to master.)
- Send someone to be trained in an off-site course. Going forward, have that person serve as the specialist, handling those tasks that require the skill.
- Send someone to be trained in an off-site course. In the short term, have that person serve as the specialist, handling those tasks that require the skill. In the long term, arrange for that person to cross-train other team members.
- Bring in on-site training for some of your team. (This option is usually only cost effective when multiple people need the training.) Going forward, have those people serve as specialists, handling those tasks that require the skill.
- Bring in on-site training for all of your team. In the short term, have some of the attendees who showed the most aptitude serve as the specialists, handling those tasks that require the skill. In the long term, arrange for those people to cross-train other team members.
- Rotate staff with the skill in from other areas of your company while rotating out staff to areas of the company that need certain skills from your team. Of course, you need to make sure you don't create a new skills gap by rotating out members of your staff.
- Have the development team provide a transfer of information to the entire test team prior to the start of testing.

As you can see, some of these have implications in terms of test team organization and perhaps even company-wide human resource management decisions. So not all of these options are available to all test managers in all situations.

10.4. Fitting Testing within an Organization

Learning objectives

(K2) Characterize the various organizational options and compare them with insourcing and outsourcing options, whether onshore or offshore.

Back in the early days of computing, there was no such specific, separately identifiable activity known as "testing". Developers debugged their code, and that was usually intertwined with some unit testing tasks. That didn't work. My first job was as a programmer, and where I worked, this was exactly how we approached quality assurance. It was an ongoing, kaleidoscopic disaster.

In spite of technological changes and agile lifecycles, this approach, as the sole quality control technique, still doesn't work. It does not work for those throwback, Neanderthal organizations that rely on it entirely. It does not work for those cutting-edge companies that think some fancy language or process or tool has solved the software quality problem at last. The software quality problem is a hard problem—of the kind Fred Brooks referred to as having "essential complexity" rather than "accidental complexity"—and no magic solutions are available for problems of essential complexity.^[1]

[1] Fred Brooks originally wrote about the difference between essential complexity and accidental complexity in his article "No Silver Bullets", which is now included as a chapter in *The Mythical Man-Month*, 2e. In short, accidental complexity is complexity we create by mistake or by the wrong choice of tools, such as forgetting to bring a map when traveling on a complex road trip or selecting a snowmobile to traverse a desert. Essential complexity is complexity that exists, inherent in the problem. Attempting to detect and remove a significant percentage of the potentially infinite number of defects that could be in a software product is an essentially hard problem. To think that we can solve this problem with simple changes like new software development lifecycles or new software testing tools is an example of another problem Brooks referred to in his book, that of excessive optimism.

With the widespread advent of independent test teams in the late 1980s and early 1990s, we saw improvements. I was working in an independent test lab in the late 1980s and as a test manager in the early 1990s, and we made great strides. However, we also saw the emergence of a new

dysfunction, the "hurl it over the wall to the test guys and hold them responsible for delivered quality" mistake. Every now and then, we work with organizations that still suffer from this.

When quality matters—and it always does always—everyone must play a role. We looked at the idea of a series of quality assurance filters in earlier chapters as well as the Foundation syllabus. Each team in the organization typically participates in and owns one or more of these filters.

Some of these filters, especially high-level testing like system test and acceptance test, work best with a high level of independence. Some of these filters, especially low-level testing like unit testing, work best with less independence. Let's survey the degrees of independence so we understand our options.

Self-testing occurs when the developer tests their own code. There is no independence here, of course. The author bias problem is significant, and the developer—even if given enough time to do unit testing—often misses the important bugs because they determine that the code works as they intended. Of course, that might not match the actual requirements. The advantages are that the developer can fix any defects they find quite quickly and, being quite technical, understands the software being tested.

Buddy testing occurs when developers test each other's code but not their own code. Pair programming, which is a practice in some agile techniques, is a special form of this, where development of code, continuous code review, and development and execution of unit tests by a team of two programmers evolves the code. While the author bias problem is not so acute here, when two people work closely together, it's hard to say there is much independence between the developer and the tester. In addition, there tend to be few if any usable defect metrics captured in this situation without careful cultivation of the proper mindset since peers testing each other's code might not want to report defects. Finally, because the average programmer has little training or formal experience with testing, the mindset is usually focused on positive tests. Once again, the advantages include a quick repair of defects and good tester understanding of the software being tested.

Having a tester or some testers inside of a development team occurs when a development team includes one or more testers. This is quite popular these days, as many proponents of agile methodologies advocate this. There is nothing wrong with it as part of a larger quality assurance process, but by

itself it can be quite dangerous. The main problem is editing and self-editing.

Self-editing means that the tester does not report—or reports only informally to the developer—those problems found, leaving no official trail in a bug tracking system. Self-editing is the equivalent of an organization tearing out its eyes and flying blind with respect to quality. Defect metrics, while insufficient by themselves for all the reasons discussed in [chapter 3](#), are certainly necessary to any balanced, meaningful picture of quality.

Even if the tester does report bugs, editing can happen. The development manager or project manager does not allow the tester to release a clear, balanced, complete set of test results to the broader set of stakeholders. Furthermore, because the development manager or project manager is often focused on short-term goals like getting the product released on time and on budget, the entire mission for the testers is likely to be verifying adherence to requirements. Finally, it is often the case in these arrangements that the tester tasks are assigned to junior developers or factotums of some sort or another, along with a number of other responsibilities. So, the testing is often done hurriedly and without any particular professionalism.

All those disadvantages enumerated, I do see value in having one or more testers—whose permanent positions are inside an independent test team and who are true professional testers—assigned to act as consultants to development teams. In this role they can help them create good test cases, they can build automated test harnesses, create continuous integration-build-and-smoke-test facilities, and the like. We have played this role for clients in the past, with great success; however, this is not sufficient by itself.

Testing by business, users, and technical support occurs, often in the context of acceptance testing and beta testing. This approach has the advantage of a truly independent outlook, motivated to report findings truthfully to the stakeholders. What these folks typically care about is the ability to get their job done, and if quality's not there, they'll suffer. This is a great approach for the final levels of testing.

Unfortunately, what we tend to see with organizations that rely on this approach for system testing is that the test teams skills are one-dimensional. They are focused entirely on domain knowledge. Technical skills, if present, are limited. Management disdains professional testing,

with the usual refrain being, "Oh, any user can test." In addition, the testers tend to bring a firefighting, patch-it-until-it-works attitude to the testing work.

Test specialists in an independent test group occur in many thoughtful organizations, with the independent test team responsible for system test, system integration test, and, in some case, component integration test. In this case, we have all the advantages of true, professional testers testing against specific test targets. Unlike the approaches mentioned earlier, we often see test targets beyond functionality, including usability, security, and performance. For all the advantages of an independent test team, it should be kept in mind that the formality usually associated with such teams does tend to slow down the process. It's also possible that reporting structures or poor management can lead to perverse incentives and a lack of focus on quality.

Finally, testing by an external test organization occurs in a number of settings. For example, in certain military contracts, independent verification and validation by a team not in any way associated with the prime contractor is required. As another example, you might hire a test lab to do compatibility testing for an e-commerce website to save the expense of having all the configurations in-house. Here, the maximum level of independence is achieved. Of course, the separation of test and development duties might mean that the knowledge transfer necessary for thorough testing might not occur. To make up for these disconnections, the organizations must put in place very clear requirements and well-defined communication structures.

Furthermore, there's a potential "who guards the guards" problem. Remember that I mentioned earlier in this chapter the fellow doing testing for a defense contractor that wanted turn-off-the-brain, fill-in-the-blanks templates for test plans, test cases, and test procedures? Surely even author-biased testing by the programmer is better than some lazy, inept tester doing a haphazard, path-of-least-resistance job of it. Any company engaging an external test organization should plan to audit the quality of that organization regularly, including the skills and professionalism of the team.

Now, notice that I listed advantages and disadvantages for each option. This means that you can use each option for one or more of the quality filters I mentioned earlier. The advantages and disadvantages tend to be mutually correcting, so a mix of degrees of independence, with different degrees for

different tasks, is often appropriate.

10.4.1. On Using Different Mixes of Independence

I mentioned in [chapter 8](#), in the context of maturity models, that reducing matters to black-or-white, either/or dualities, while simplifying the concept, often loses some of the important nuances. This question of independence is one where too many people have fallen into this same trap. People ask, "Should we have developers testing, or users testing, or independent test teams, or buddy testing, or outside test labs?"

The answer is not one of these options to the exclusion of all others, but rather each of these options, in some mix with some or all of the other options, to the degree appropriate for the particular project, product, and software development and testing process. Independence is a matter of degree, not an either/or state.

In addition, independence is an attribute of the relationship between those developing and those testing. For any two entities—whether at the level of individual people, teams, or organizations—we can ask, "What is the relationship between them and to what extent are they independent?" The more an entity is free to act as it sees fit, without having to accept direction from the other entity, the more independent that entity is.

Notice that the distinction here is not one of disregarding how one's actions affect another's but rather not having to get approval for those actions. This is important to remember because some independent test teams make the mistake of thinking that they can and should—as a sign of their independence—do whatever they think is right, and to the devil with the consequences to the project and the organization. Independent test teams that fall into this adversarial, quality cop mindset often end up disbanded. Successful independent test teams act in consultation with other project stakeholders, preserving their independence but with the goal of serving the stakeholders and the best interests of the project and organization in mind.

Increasing independence of testing is not without risks, as I discussed earlier. More independence can result in more isolation. It can reduce the level of insight and understanding of what is going on in the project. It can also lead to a loss of ownership and responsibility for quality on the part of those developing the code. These are not necessary outcomes of independent testing, as some of those who argue a dualistic view of this

question suggest, but simply project risks that the manager of an independent test team must mitigate.

Decreasing independence of testing is not without risk either. It can increase insight and understanding of the project—and this is the outcome touted by many of the dualists—but it can and often does introduce conflicting goals. Decreasing independence can lead to blinds spots as to what the requirements really are. Decreasing independence also decreases the degree to which testing involves people who possess focus and specialization on testing, and thus teams have an imperfect, skewed skills mix.

In some cases, the choice of software development models and other project realities can influence the choices here. For example, if you follow an agile lifecycle model, then pair programming and testing within the development team might be part of the mix. As another example, if you are seeking to have your product certified as Microsoft compatible, then having Microsoft's compatibility test lab—obviously completely independent from your organization—will be required.

Again, remember that you can mix all of these independence options that we've discussed. In a few moments, you'll see an example of a very successful organization that does just that.

When you do split up the testing across various entities with various degrees of independence, the usual rules of pervasive testing apply. Make sure that you define the responsibilities and expectations for each test level and entity doing testing. A concise, clear test policy document, developed with the participation of all the entities and approved by senior management, can accomplish this. By doing this, you'll be setting up a mix of different filters deployed at the ideal spot in the lifecycle, which can maximize quality within the schedule and budget constraints of the project.

independence of testing: Separation of responsibilities, which encourages the accomplishment of objective testing.

10.4.2. Outsourcing as Independent Testing

Outsourcing of testing is one form of external, independent testing. This can take a number of forms. One is hiring an outside testing company to provide collocated testing services (which is sometimes called insourcing). Another is to have the testing done at an external facility located close to the development team. Yet another is to have the testing done at an external facility that is some distance away, perhaps even in different time zones.

Many outsource services companies, RBCS included, provide a mix of all three options. We have testers who work on-site for our clients. We have the ability to provide testing services through local or in-country facilities. We also have the ability to have the testing done at our facility in New Delhi.

I discussed this topic at some length in earlier in this book, so let's review some of the challenges here:

- The outsource testing team might have cultural differences with your test team, your development team, or both.
- The project team and local test team might have difficulty providing timely, adequate supervision and direction, particularly on chaotic, constantly changing projects.
- Due to a lack of foresight, significant communication problems can exist between the local project team and the outsource test entity. Poor communication can compound the supervision problem.
- Without careful contracting, you can have problems with protection of intellectual property. Even with good contracts, in some countries your legal recourse might be quite limited.

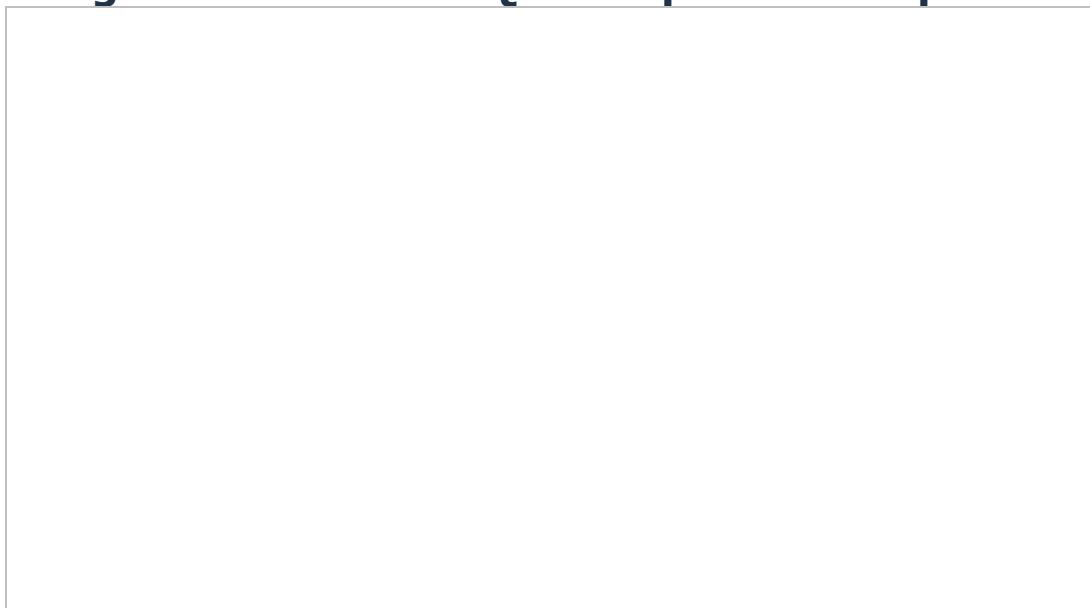
- Again, if insufficient care is taken during the contracting of the testing, including especially the selection of the outsource test vendor, the skills of the testers can be questionable.
- Exacerbating this skills problem can be the problem of employee turnover. Again, proper contracting and vendor selection can help reduce this.
- Because companies pursuing outsourcing often forget to include their own costs of managing the relationship in the overall budget, outsourcing does not always involve accurate cost estimation.
- Finally, quality of the work can suffer.

Some of these challenges can increase with distance. Again, I'm not mentioning these as outcomes that necessarily will occur and dog your project, but rather as risks that you can and should mitigate through the kinds of techniques for managing outsourcing testing that I covered in [chapter 3](#). A lot of it comes down to careful outsource test vendor selection and careful contracting.

10.4.3. Case Study of Mixed QA Independence Options

[figure 10-9](#) shows how one of our clients approaches integrating quality assurance tasks, including testing, into the entire development lifecycle with various degrees of independence.

Figure 10-9. Mixed QA independence options



First, let me review the lifecycle. Project teams take on projects for various business stakeholders who authorize them. To be deployed, projects will be included in a bundled release that will be tested with other projects and delivered into production. The bundled releases into production occur about every two months. This approach minimizes regression and interoperability risks.

For each project, as it moves through this lifecycle, a variety of quality assurance activities take place. Each has different levels of independence, different responsible parties, and a different focus.

Each project goes through a requirements and design specification period at the beginning. The independent test team leads mandatory reviews of these specification documents. The participants vary, with business stakeholders and business analysts heavily involved in the requirements reviews and senior programmers, system architects, network administrators, and database administrators heavily involved in the design reviews.

Each project then goes into a development phase. Each programmer unit tests their own code. Programmers review each other's code. Unit tests are subject to approval.

Each project then moves into a pre-bundle testing period. In other words, the code developed will go through separate testing prior to integration with the other projects' code. This period can—but need not necessarily—contain a component integration test level, depending on the number of modules involved. The programmers themselves handle this component integration test, assisted by a test team internal to the development team working on the project. This internal development test team is a transient team assembled by the project manager from members of the project development team. Thus, it is not independent but does have excellent insight into the project, especially technically.

The pre-bundle test period contains a mandatory system test level. The development test team handles this test level.

At this point, all those projects that have qualified to enter the bundle will do so. The projects qualified to enter the bundle have project metrics indicating sufficient quality to be in the bundle.

The first quality filter for the bundled software is system integration testing by the independent test team. This is a mandatory activity. The independent test team includes insourced testers, including from RBCS. One

of the test types during system integration testing, security testing, is done by the security team, another separate group.

The final quality filter is user acceptance testing, done by actual users, ideally expert users who know the most about the business problem to be solved by the system.

So, we have eight quality filters:

- Requirements reviews
- Design reviews
- Code reviews
- Unit test
- Component integration test
- System test
- System integration test
- User acceptance test

These filters have different levels of independence.

Does this approach work? Well, the defect detection percentage for system integration test and user acceptance test has been consistently over 99 percent for a few years now. This high level of defect detection effectiveness is in part because the independent test team and users find very few bugs in those final filters, only those bugs that couldn't have been found earlier.

So, yes, this approach works. In fact, if what you care about is the highest quality, every time, without compromise, this approach not only works, but indeed it's the way it is done.

10.5. Motivation

Learning objectives

(K2) Provide example of motivating and demotivating factors for testers.

As test managers, we need to motivate our team members. How can we motivate testers? There are a number ways.

Recognition. When someone does good work, tell them so. Public praise for a job well done is a major motivator for many people. The converse, by the way, is not true: Public criticism for a job poorly done will not motivate someone to do better. You should give criticism privately, and make sure that criticism is constructive and guides the tester toward the desired improvement.

Management approval. This doesn't just mean your approval. As a test manager, promote your team and get upper-management approval for your work. Then, share that with your team.

Respect. As a test manager, you want to build a culture of mutual respect in your team. You must also ensure that the project team treats your testers with respect. Of course, that respect must be earned.

Adequate rewards. This includes financial elements, such as salary, merit increases, and bonuses. It also includes nonfinancial elements such as training and career growth.

Of course, it happens that project realities constrain the motivational tools available to you. If the team is subject to impossible deadlines, that makes the job hopeless. In addition, tight project deadlines usually mean poor quality when testing starts because everyone is taking shortcuts.

Testers should not take bug deferral personally, if it is done thoughtfully and with an eye toward what matters to customers. However, wholesale deferral of bugs without any consideration of the impact of them is demotivating.

Crunch time happens on projects, and testers should be ready to put in extra hours to accomplish important tasks. However, mandatory overtime that clearly accomplishes nothing is demotivating.

Finally, being held responsible for quality when everyone else on the project is taking steps that actively undermine quality—being the quality scapegoat—is demotivating.

As a test manager, you can't necessarily stop these things from happening, but you should plan to expend serious political capital to try to resolve them. At the least, your team will see that you are serious about trying to maintain their motivation. Few things are more demotivational than working for a manager who doesn't care about how you feel.

10.5.1. Metrics and Motivation

Unfortunately, metrics are often misused, so many people think of metrics as demotivating. Properly used, though, product, process, and project metrics can demonstrate the value of testing.

You should remember these metrics from [chapter 3](#). You can measure your team's effectiveness, for example, using the defect detection percentage to measure bug finding. You can measure your team's efficiency, for example, by looking at cost of quality. You can measure your team's risk mitigation contribution. You can measure your team's cross-functional interactions, for example, by looking at bug report reject rates and surveying stakeholders about the quality of your test results reports.

Define an assessment program for your test team. You can use the models discussed in [chapter 8](#). In consultation with the test team and other project stakeholders, and based on what you want to see, set goals and publish progress toward those goals. This will help you demonstrate value and progress to management and will also help show testers that, in spite of some daily frustrations, you are collectively, as a team, making progress.

Recognition takes various forms. People want respect from their peers and managers and peer and manager approval of their work. They want promotional opportunities. They want fair pay, relative to their peers and the industry. They want a career path.

A test team that is not respected will not be recognized. One that is respected can make meaningful contributions and will be recognized; that is a virtuous cycle, and your job as a test manager is to create and sustain that cycle.

To create and sustain this cycle of recognition and respect, you must be able to demonstrate value. Metrics are essential to demonstrate value. Test

managers who insist on respect and recognition for their test teams based purely on subjective reports of good work done—or worse yet, based on an assertion that the test team deserves respect—tend to be disappointed in the organization's reaction.

10.5.2. Case Studies of Motivating and Demotivating Comments

One thing that makes a project motivating—or demotivating—is the feedback you get from your peers. So here are some example quotes from project team members that illustrate either motivating or demotivating experiences for test teams that I've managed.

Let's start with the demotivating comments:

- "We all have pain in our lives right now." This comment came from a developer in a conference call. His implication was that, because developers were suffering from all the bugs found, the test team needed to have some pain in its life too. However, that's an odd idea of justice, since the test team suffered for bugs it didn't create.
- "Some developers are returning bugs as 'irreproducible,' even though our bug reports document them as intermittent ... or reproducible only in the test environment." This is from an email I sent to the development manager to report a problem with development handling of bug reports. It was frustrating for our test team to see these responses because it was clear that the developers weren't paying attention to the reports.

Now, here are some motivating ones:

- "If you never made anyone upset, you wouldn't be doing your job." This comment came from a developer who was remarking on another developer's defensive reaction to a bug report. Seeing that understanding within the development team was motivating to us.
- "As of [next week], test goes on a five-day schedule." This was a comment I sent to the test team, ending a long period of six-day weeks. They were motivated to start working normal hours again.
- "I'm there, and I really like expensive tequila!" This was from a manager, agreeing to come to a test team dinner after a tough project. Getting some attention in a social setting from management sends a

message to the test team that they matter.

An important point here is that simple comments—often spontaneous comments—can have a serious impact on motivation.

10.6. Communication

Learning objectives

(K2) Describe by example professional, objective, and effective communication in a project from the tester perspective, considering risks and opportunities

There are three levels of communication for most test teams:

- First, we communicate, mostly internally but also with others, about the documentation of test products. These communications include discussions of test strategies, test plans, test cases, test summary reports, and defect reports.
- Second, we communicate feedback on reviewed documents, typically on a peer level both inside and outside the test group. These communications include discussions about requirements, functional specifications, use cases, and unit test documentation.
- Third, we communicate as part of information gathering and dissemination. These communications include not just peer level communications but communications to managers, users, and other project stakeholders. It can be sensitive, as when test results are not encouraging for project success.

It's important to remember that both internal and external communications are important to the professionalism of the testers that work for you, and for you as a test manager.

Effective communication assists you in achieving your objectives as a test manager, while ineffective communication will hinder you. It's important to be professional, objective, and effective. You want each communication you have, both inside and outside the test team, to build and maintain respect for the test team. When communicating about test results, giving feedback on issues with documents, or delivering any other potentially touchy news, make sure to use diplomacy.

It's easy to become caught up in emotions at work, especially during test execution when things are often stressful. Remember to focus on achieving test objectives. Remember also that you want to see the quality of products and processes improved. Don't engage in communication that is contrary to

those goals.

It's also easy to communicate as if you were communicating with yourself or someone like you. In other words, we testers often speak in a sort of shorthand about very fine-grained details of our work and findings, and with a certain degree of skepticism. When talking to fellow testers, this is fine. However, you have to remember to tailor communication for the target audience. When talking to users, project team members, management, external test groups, and customers, you need to think carefully about how you are communicating, what you are communicating, and whether your communications support your goals.

As a test manager, I have seen a single thoughtless email, bug report, or hallway conversation do a great deal of damage to my test team's reputation and credibility. So, even with all the other work you have to do, remember to think about your communications.

Figure 10-10 is an example of test communication. This is an excerpt of an email to a vendor about the results of acceptance testing our new website. The first paragraph communicates that this is a carefully thought-out analysis, not just one of the dozens of ill-considered emails a manager is likely to get in a day. The message is, "Pay attention to this email, please, because I did." This paragraph also refers the reader to further details in the attached document.

Figure 10-10. Acceptance test status email

I have spent a couple hours reviewing the current status of the site and the acceptance test. Please see attached [documents] with deferred bug reports and [test status] ...for pass 2.

The following issues are must-fix to move forward with deployment:

- Consistency of messages and UI (for examples, see bug 85,91, 92, 95, 97)
- Newsletter link...not in place as agreed (see bug 98)
- Identification and resolution of internal dead links (see bug 103)

While some of these issues might strike the casual reader as picayune, please understand that our target customers... are sensitive to any errors...

In the interests of moving forward and having this critical marketing collateral in place.. I have agreed to defer a number of bugs from pass 1 that either failed verification testing or which related to advertised product features that [the vendor] retroactively and unilaterally withdrew... Please note that deferral of these bugs does not indicate acceptance by RBCS of the disposition of those bugs for all time.

Finally, please note that there were eighteen (18) new bug reports filed during the second pass.

The second paragraph—including the bullet list and closing sentence—

summarizes what needs to be done to complete the acceptance testing and move into deployment.

The third paragraph clarifies the meaning of the deferral of certain bugs. I wanted to make sure that I was not waiving any legal rights RBCS had to insist on these problems being fixed later.

The final paragraph is a subtle—I hope—hint that we were disappointed to be still finding problems.

Now, this type of email is appropriate for me to send as a customer to a vendor, explaining test results. Would you send it to your development colleagues? Probably not. The important point here is that every word and every sentence of that email had a communication objective.

10.7. Sample Exam Questions

To end each chapter, you can try one or more sample exam questions to reinforce your knowledge and understanding of the material and to prepare for the ISTQB Advanced Level Test Manager exam.

Q: Assume you have just been hired as a test manager in charge of integration testing, system testing, and acceptance testing for a bank. Most of the bank's systems involve a mix of mainframe central data repositories and services, UNIX-based client-server applications in the branches and back office, and browser-based banking access for customers.

You have inherited a testing team from your predecessor as test manager. His team was effective at finding defects and at automating most of the regression tests. However, he left under pressure due to a lack of overall business confidence in the systems after release and due to relationship problems between himself and other managers in the organization. He staffed the test team primarily with people who had extensive experience testing systems that were implemented using one or more of the same types of technologies used by your bank. He tended to hire people who were self-taught or had learned on the job.

You are now performing a skills inventory on your test team, identifying strengths and weaknesses in five major areas: technical knowledge of software systems, business knowledge related to banking applications, software testing knowledge, interpersonal skills, and education and training. Based only on the information given in the scenario here, identify the results that you are most confident you will find in this skills inventory?

1. The team has broad knowledge of the banking domain.
2. The team has focused knowledge of pertinent technologies.
3. The team has targeted knowledge of testing.
4. The team has strong interpersonal skills.
5. The team has sound knowledge of testing fundamentals.

Q: Continue the previous scenario.

Assume that you are in a position to hire two additional testers for your test team. In addition, you have been given a budget to provide training for the team. Your manager has asked you to submit a team development plan for the coming year. She has told you that you may provide two training courses for your team. She has also told you that you may hire two people, and you may use those new hires to cross-train your team. Select the best possible choices from the following list.

- 1.** Provide test training focused on automation skills.
- 2.** Provide test training covering broad testing topics.
- 3.** Provide business skills training focused on banking applications.
- 4.** Hire a tester with experience in testing banking applications.
- 5.** Hire a tester with experience in the underlying technologies.
- 6.** Hire a tester with strong interpersonal skills.

Q: 3. Assume you are a test manager working on a project to create a programmable thermostat for home use to control central heating, ventilation, and air conditioning (HVAC) systems. In addition to the normal HVAC control functions, the thermostat also has the ability to download data to a browser-based application that runs on PCs for further analysis.

You have decided to outsource browser compatibility testing for this system to a large company that provides testing, development, and technical support services offshore. By checking this company's references, you have determined that it has done compatibility testing for e-commerce applications in the past. Which of the following is most likely to be true about the outsource organization?

- 1.** This organization will have the browsers and PCs required.
- 2.** This organization will have people who specialize in testing.
- 3.** This organization will have a stable test team.

4. This organization will do a better job of testing than an in-house team.

Q: You are a test manager in charge of system testing on a project to update a cruise-control module for a new model of a car. The goal of the cruise-control software update is to make the car more fuel efficient. Assume that management has granted you the time, people, and resources required for your test effort, based on your estimate. Which of the following is an example of a motivational technique for testers that will work properly and is based on the concept of adequate rewards as discussed in the Advanced syllabus?

1. Bonuses for the test team based on improving fuel efficiency by 20 percent or more
2. Bonuses for the test team based on detecting 90 percent of defects prior to release
3. Bonuses for individual testers based on finding the largest number of defects
4. Criticism of individual testers at team meetings when someone makes a mistake

Q: 5. Assume you are a test manager for a banking project to upgrade an existing automated teller machine system to allow customers to obtain cash advances from supported credit cards. You have not been able to obtain a list of the supported credit cards, which are not included in the requirements specification. Which of the following is an example of a good way to communicate that problem in an email?

1. "Until we receive a complete list of supported credit card types, I have directed the test analysts not to do any work on test design."
2. "When will it be possible for us to know which specific credit card types will be supported? Test design is impeded by a lack of clarity here."
3. "Here we go again. The business analysts gave us incomplete and ambiguous requirements specifications. Typical."

4. Do not communicate the problem, just log the delaying effect of the information problem and be ready to explain the delays to the project manager later.

Chapter 11. Preparing for the Exam

"Not everyone who doesn't study for the exam will fail.

But everyone who does fail didn't study enough."

—A frequent warning from the author when teaching ISTQB courses.

The 11th chapter of this book is concerned with topics that you need to know in order to prepare for the ISTQB Advanced Test Manager exam. The chapter starts with a discussion of the ISTQB Advanced Test Manager learning objectives, which are the basis of the exams.

[chapter 11](#) of this book has two sections:

1. Learning Objectives
2. ISTQB Advanced Exams

If you are not interested in taking the ISTQB Advanced Test Manager exam, this chapter might not be pertinent for you.

11.1. Learning Objectives

National boards and exam boards develop the Advanced syllabus exams based on learning objectives. A learning objective states what you should be able to do prior to taking an Advanced exam. Each Advanced exam has its own set of learning objectives. I listed the learning objectives for the Advanced Test Manager exam at the beginning of each section in each chapter.

The learning objectives are at four levels of increasing difficulty. Question writers will structure exam questions so that you must have achieved these learning objectives to determine the correct answer for the question. The exams will cover the more basic levels of remembrance and understanding implicitly as part of the more sophisticated levels of application and analysis. For example, to answer a question about how to create a test plan, you will have to remember and understand the IEEE 829 test plan template. So, unlike the Foundation exam, where simple remembrance and understanding often suffice to determine the correct answer, an Advanced exam requires you to apply or analyze the facts that you remember and understand in order to determine the correct answer.

Let's take a closer look at the four levels of learning objectives you will encounter on the Advanced exams. The tags K1, K2, K3, and K4 are used to indicate these four levels, so remember those tags as you review the Advanced syllabus.

11.1.1. Level 1: Remember (K1)

At this lowest level of learning, you will be expected to recognize, remember, and recall a term or concept. Watch for keywords such as remember, recall, recognize, or know. Again, this level of learning is likely to be implicit within a higher-level question.

For example, you should be able to recognize the definition of failure as:

"nondelivery of service to an end user or any other stakeholder," and

"actual deviation of the component or system from its expected delivery, service or result."

Such level 1 learning objectives mean that you should be able to remember the ISTQB glossary definitions of terms used in the ISTQB Advanced

syllabus and also standards like ISO 9126 and IEEE 829 that are referenced in the Advanced syllabus. Expect this level of learning to be required for questions focused on higher levels of learning like K3 and K4.

11.1.2. Level 2: Understand (K2)

At this second level of learning, you will be expected to be able to select the reasons or explanations for statements related to the topic and summarize, differentiate, classify, and give examples. This learning objective applies to facts, so you should be able to compare the meanings of terms. You should also be able to understand testing concepts. In addition, you should be able to understand test procedure, such as explaining the sequence of tasks. Watch for keywords such as *summarize, classify, compare, map, contrast, exemplify, interpret, translate, represent, infer, conclude, and categorize*.

For example, you should be able to explain the reason why tests should be designed as early as possible:

- To find defects when they are cheaper to remove.
- To find the most important defects first.

You should also be able to explain the similarities and differences between integration and system testing:

- Similarities: Testing more than one component and testing nonfunctional aspects.
- Differences: Integration testing concentrates on interfaces and interactions, while system testing concentrates on whole-system aspects, such as end-to-end processing.

Such level 2 learning objectives mean that you should be able to understand the ISTQB glossary terms used in the ISTQB Advanced syllabus and the proper use of standards like ISO 9126 and IEEE 829 that are referenced in the Advanced syllabus. Expect this level of learning to be required for questions focused on higher levels of learning like K3 and K4.

11.1.3. Level 3: Apply (K3)

At this third level of learning, you should be able to select the correct application of a concept or technique and apply it to a given context. This level is normally applicable to procedural knowledge. At K3, you don't need

to evaluate a software application or create a testing model for a given software application. If the syllabus gives a model, the coverage requirements for that model, and the procedural steps to create test cases from a model in the Advanced syllabus, then you are dealing with a K3 learning objective. Watch for keywords such as *implement, execute, use, follow a procedure, and apply a procedure*.

For example, you should be able to do the following:

- Identify boundary values for valid and invalid equivalence partitions.
- Use the generic procedure for test case creation to select the test cases from a given state transition diagram (and a set of test cases) in order to cover all transitions.

Such level 3 learning objectives mean that you should be able to apply the techniques described in the ISTQB Advanced syllabus to specific exam questions. Expect this level of learning to include lower levels of learning like K1 and K2.

11.1.4. Level 4: Analyze (K4)

At this fourth level of learning, you should be able to separate information related to a procedure or technique into its constituent parts for better understanding and distinguish between facts and inferences. A typical exam question at this level will require you to analyze a document, software, or project situation and propose appropriate actions to solve a problem or complete a task. Watch for keywords such as *analyze, differentiate, select, structure, focus, attribute, deconstruct, evaluate, judge, monitor, coordinate, create, synthesize, generate, hypothesize, plan, design, construct, and produce*.

For example, you should be able to do the following:

- Analyze product risks and propose preventive and corrective mitigation activities.
- Describe which portions of an incident report are factual and which are inferred from results.

Such level 4 learning objectives mean that you should be able to analyze the techniques and concepts described in the ISTQB Advanced syllabus in order to answer specific exam questions. Expect this level of learning to

include lower levels of learning like K1, K2, and perhaps even K3.

11.1.5. Where Did These Levels of Learning Objective Come From?

If you are curious about how this taxonomy and these levels of learning objectives came to be in the Foundation and Advanced syllabi, then you'll want to refer to Bloom's taxonomy of learning objectives, defined in the 1950s. It's standard educational fare, though you probably haven't encountered it unless you've been involved in teaching training courses.

You might find it simpler to think about the levels this way:

- K1 requires the ability to remember basic facts, techniques, and standards, though you might not understand what they mean.
- K2 requires the ability to understand the facts, techniques, and standards and how they interrelate, though you might not be able to apply them to your projects.
- K3 requires the ability to apply facts, techniques, and standards to your projects, though you might not be able to adapt them or select the most appropriate ones for your project.
- K4 requires the ability to analyze facts, techniques, and standards as they might apply to your projects and adapt them or select the most appropriate ones for your project.

As you can see, there is an upward progression of ability that adheres to each increasing level of learning. Much of the focus at the Advanced level is on application and analysis.

11.2. ISTQB Advanced Exams

Like the Foundation exam, the Advanced exams are multiple choice exams. Multiple choice questions consist of three main parts. The first part is the stem, which is the body of the question. The stem may include a figure or table as well as text. The second part is the distractors, the choices that are wrong. If you don't have a full understanding of the learning objectives that the question covers, you might find the distractors to be reasonable choices. The third part is the answer or answers, the choice or choices that are correct.

If you sailed through the Foundation exam, you might think that you'll manage to do the same with the Advanced exams. That's unlikely. Unlike the Foundation exam, the Advanced exams are heavily focused on questions derived from K3 and K4 level learning objectives. In other words, the ability to apply and to analyze ideas dominates the exams. K1 and K2 level learning objectives, which make up the bulk of the Foundation exam, are mostly covered implicitly within the higher-level questions.

For example, the Foundation exam might typically include a question like this:

Which of the following is a major section of an IEEE 829 compliant test plan?

- 1.** Test items
- 2.** Probe effect
- 3.** Purpose
- 4.** Expected results

The answer is A, while B, C, and D are distractors. All that is required here is to recall the major sections of the IEEE 829 templates. Only A is found in the test plan, while C and D are in the test procedure specification and the test case specification, respectively. B is an ISTQB glossary term. As you can see, it's all simple recall.

Recall is useful, especially when you're first learning a subject. However, the ability to recall facts does not make you an expert, any more than my ability to recall song lyrics from the 1970s qualifies me to work as the lead

singer for the band AC/DC.

On the Advanced exam, you might find a question like this:

Consider the following excerpt from the Test Items section of a test plan.

During System Test execution, the configuration management team shall deliver test releases to the test team every Monday morning by 9:00 a.m. Each test release shall include a test item transmittal report. The test item transmittal report will describe the results of the automated build and smoke test associated with the release. Upon receipt of the test release, if the smoke test was successful, the test manager will install it in the test lab. Testing will commence on Monday morning once the new weekly release is installed.

Should the test team not receive a test release, or if the smoke test results are negative, or if the release will not install, or should the release arrive without a transmittal report, the test manager shall immediately contact the configuration management team manager. If the problem is not resolved within one hour, the test manager shall notify the project manager and continue testing against the previous week's release, if possible. If the test release fails installation, additionally the test analyst who attempted the installation shall file an incident report.

Assume that you are working as the test manager on this project. Suppose that you have received two working, installable, testable releases so far. On Monday of the third week, you do not receive the test release.

Which of the following courses of action is consistent with the test plan?

1. Filing an incident report describing the time and date at which you first noticed the missing test release
2. Creating a test procedure specification that describes how to install a test release
3. Sending an SMS text to the configuration management team manager
4. Sending an email to the project manager and the configuration management team manager

The answer is C. A, B, and D are distracters. A is wrong because it is not that the release didn't install, it's that it didn't even arrive. B is wrong

because, while such a test procedure might be useful for installation testing, it has nothing to do with the escalation process described in the test plan. C is consistent with the test plan. D is not consistent with the test plan because the spirit of the one-hour delay described in the test plan excerpt is that the configuration management team manager should have a chance to resolve the problem before the project manager is engaged. In addition, when time is of the essence, email is not a good escalation technique.

As you can see, this kind of question requires analysis of a situation. Yes, it helps to know what the IEEE 829 templates such as the test plan, incident report, test item transmittal report, and test procedure specification contain. In fact, you'll probably get lost in the terminology if you don't know the standard. However, simply knowing the IEEE 829 standard will not allow you to get the right answer on this question except by chance.

11.2.1. Scenario-Based Questions

Further complicating this situation is the fact that many exam questions will actually consider a scenario. In scenario-based questions, the exam will describe a set of circumstances. It will then present you with a sequence of two, three, or even more questions based on that scenario.

For example, the questions about the scenario of the test plan excerpt and the missing test release might continue with another pair of questions:

Assume that on Monday afternoon you finally receive a test release. When your lead test analyst attempts to install it, the database configuration scripts included in the installation terminate in midstream. An error message is presented on the database server in Cyrillic script, though the chosen language is U.S. English. At that point, the database tables are corrupted and any attempt to use the application under test results in various database connection error messages (which are at least presented in U.S. English).

Consider the following possible actions:

- 1. Notifying the configuration management team manager**
- 2. Notifying the project manager**
- 3. Filing an incident report**
- 4. Attempting to repeat the installation**

5. Suspending testing

6. Continuing testing

Which of the following sequence of actions is in the correct order, is the most reasonable, and is most consistent with the intent of the test plan?

1. I, II, V
2. V, I, IV, III, I
3. VI, II, I, III, IV
4. II, I, V

The answer is B, while A, C, and D are distractors. A is wrong because there is no incident report filed, which is required by the test plan when the installation fails. C is wrong because meaningful testing cannot continue against the corrupted database, because the project manager is notified before the configuration management team manager, and because the incident report is filed before an attempt to reproduce the failure has occurred. D is wrong because the project manager is notified before the configuration management team manager and because no incident report is filed.

As you can see, with a scenario-based question it's very important that you study the scenario carefully before trying to answer the questions that relate to it. If you misunderstand the scenario—perhaps due to a rushed reading of it—you can anticipate missing most if not all of the questions related to it.

Let me go back to this question of learning objectives for a moment. I said that the exam covers K1 and K2 learning objectives—those requiring recall and understanding, respectively—as part of a higher level K3 or K4 question. There's an added complication with K1 learning objectives: They are not explicitly defined. The entire syllabus, including glossary terms used and standards referenced, is implicitly covered by K1 learning objectives. As quoted from the Advanced Syllabus:

This syllabus' content, terms and the major elements (purposes) of all standards listed shall at least be remembered (K1), even if not explicitly mentioned in the learning objectives.

So, you'll want to read the Advanced syllabus carefully, a number of times, as you are studying for the Advanced exam.

Not only should you read the Advanced syllabus, but you'll need to go back and refresh yourself on the Foundation. From the Advanced syllabus:

All Advanced Certificate examinations must be based on this syllabus and on the Foundation Level Syllabus. Answers to examination questions may require the use of material based on more than one section of this and the Foundation Level Syllabus. All sections of this and the Foundation Level Syllabus are examinable.

Notice that the second sentence in the above paragraph means that a question can conceivably cross-reference two or three sections of the Advanced syllabus or cross-reference a section of the Advanced syllabus with the Foundation syllabus. It would be smart to take a sample Foundation exam and reread the Foundation syllabus as part of studying for the Advanced exam.

11.2.2. On the Evolution of the Exams

The structure of the Advanced exams is in flux and might not be completely and universally determined until the end of 2009. Even once the ISTQB Advanced Exam Guidelines are finalized, note that there is a somewhat insidious paragraph tucked away in the Advanced Syllabus:

The format of the examination is defined by the Advanced Exam Guidelines of the ISTQB. Individual Member Boards may adopt other examination schemes if desired.

I have written this chapter in large part based on the draft ISTQB Advanced Exam Guidelines. I assume that most ISTQB national boards and exam boards will choose to follow those guidelines. However, it is permitted, based on this paragraph in the Advanced syllabus, for some boards to create exams that differ from the ISTQB Exam Guidelines and thus what is described in this chapter. You'll want to check with the national board or exam board that is providing your exam to be sure. They should post on their website the version of the ISTQB Advanced Exam Guidelines they used to create their exam.

Okay, having read this, you might be panicking. Don't panic! Remember, the exam is meant to test your achievement of the learning objectives in the Advanced syllabus. This book contains solid features to help you do

that. Ask yourself the following questions:

- Did you work through all the exercises in the book? If so, then you have a solid grasp of the most difficult learning objectives, the K3 and K4 objectives. If not, then go back and do so now.
- Did you work through all the sample exam questions in the book? If so, then you have tried a sample exam question for most of the learning objectives in the syllabus. If not, then go back and do so now.
- Did you read the ISTQB glossary term definitions where they occurred in the chapters? If so, then you are familiar with these terms. If not, then return to the ISTQB glossary now and review those terms.
- Did you read every chapter of this book and the entire ISTQB Advanced syllabus? If so, then you know the material in the ISTQB Advanced syllabus. If not, then review the ISTQB Advanced syllabus and reread those sections of this book that correspond to the parts of the syllabus you find most confusing.

I can't guarantee that you will pass the exam. However, if you have taken advantage of the learning opportunities created by this book, by the ISTQB glossary, and by the ISTQB Advanced syllabus, you will be in good shape for the exam.

Good luck to you when you take the exam, and the best of success when you apply the ideas in the Advanced syllabus to your next testing project.

Appendix A. Bibliography

Advanced Syllabus Referenced Standards

Advanced Syllabus Referenced Books

Other Referenced Books

Other References

Advanced Syllabus Referenced Standards

British Computer Society. BS 7925-2 (1998), Software Component Testing.

Institute of Electrical and Electronics Engineers. IEEE Standard 829 (1998/2007), IEEE Standard for Software Test Documentation.

Institute of Electrical and Electronics Engineers. IEEE Standard 1028 (1997), IEEE Standard for Software Reviews.

Institute of Electrical and Electronics Engineers. IEEE Standard 1044 (1993), IEEE Standard Classification for Software Anomalies.

International Standards Organization. ISO/IEC 9126-1:2001, Software Engineering - Software Product Quality.

International Software Testing Qualifications Board. ISTQB Glossary (2007), ISTQB Glossary of terms used in Software Testing, Version 2.0.

US Federal Aviation Administration. DO-178B/ED-12B, Software Considerations in Airborne Systems and Equipment Certification.

Advanced Syllabus Referenced Books

Beizer, Boris. *Black-Box Testing*. Wiley, 1995.

Black, Rex. *Managing the Testing Process (Second Edition)*. Wiley, 2002.

Black, Rex. *Critical Testing Processes*. Addison-Wesley, 2003.

Black, Rex. *Pragmatic Software Testing*. Wiley, 2007.

Buwalda, Hans. *Integrated Test Design and Automation*. Addison-Wesley, 2001.

Burnstein, Ilene. *Practical Software Testing*. Springer, 2003.

Copeland, Lee. *A Practitioner's Guide to Software Test Design*. Artech House, 2003.

Craig, Rick, and Stefan Jaskiel. *Systematic Software Testing*. Artech House, 2002.

Gerrard, Paul, and Neil Thompson. *Risk-Based e-Business Testing*. Artech House, 2002.

Gilb, Tom, and Dorothy Graham. *Software Inspection*. Addison-Wesley, 1993.

Graham, Dorothy, Erik van Veenendaal, Isabel Evans, and Rex Black. *Foundations of Software Testing*. Thomson Learning, 2007.

Grochmann, M. "Test Case Design Using Classification Trees." Conference Proceedings of STAR 1994.

Jorgensen, Paul. *Software Testing: A Craftsman's Approach (Second Edition)*. CRC Press, 2002.

Kaner, Cem, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing*. Wiley, 2002.

Koomen, Tim, and Martin Pol. *Test Process Improvement*. Addison-Wesley, 1999.

Myers, Glenford. *The Art of Software Testing*. Wiley, 1979.

Pol, Martin, Ruud Teunissen, and Erik van Veenendaal. *Software Testing: A Guide to the TMap Approach*. Addison-Wesley, 2002.

Splaine, Steven, and Stefan Jaskiel. *The Web-Testing Handbook*. STQE Publishing, 2001.

Stamatis, D. H. *Failure Mode and Effect Analysis*. ASQ Press, 1995.

van Veenendaal, Erik, ed. *The Testing Practitioner*. UTN Publishing, 2002.

Whittaker, James. *How to Break Software*. Addison-Wesley, 2003.

Whittaker, James, and Herbert Thompson. *How to Break Software Security*. Addison-Wesley, 2004.

Wiegers, Karl. *Software Requirements (Second Edition)*. Microsoft Press, 2003.^[5]

[5] In an omission, this book is not included in the bibliography but it is referenced in the text.

Other Referenced Books

- Beizer, Boris. *Software Test Techniques*. 2e. Van Nostrand Reinhold, 1990.
- Bernstein, Peter. *Against the Gods*. Wiley, 1996.
- DeMarco, Tom, and Tim Lister. *Peopleware*, 2e. Dorset House, 1999.
- Dornich, Dieter. *The Logic of Failure*. Basic Books, 1997.
- Hetzl, William. *The Complete Guide to Software Testing*. Wiley, 1988.
- Jones, Capers. *Estimating Software Costs*. McGraw-Hill, 1995.
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley, 2000.
- Koomen, Tim. et al. *T-Map Next*. UTN Publishers, 2006.
- Levitt, Steven. *Freakonomics*. William Morrow, 2006.
- Tufte, Edward. *The Visual Display of Quantitative Information*, 2e. Graphics Press, 2001.
- Tufte, Edward. *Envisioning Information*. Graphics Press, 1990.
- Tufte, Edward. *Visual Explanations*. Graphics Press, 1997.
- Voas, Jeff, and Gary McGraw. *Software Fault Injection*. Wiley, 1998.
- Walsh, James. *True Odds*. Merritt Publishing, 1996.
- Walton, Mary. *The Deming Management Method*. Putnam Publishing Group, 1986.

Other References

Holmes, Jeff. "Identifying Code-Inspection Improvements Using Statistical Black Belt Techniques." *Software Quality Professional*, December 2003, Volume 6, Number 1.

www.risks.org, the Risks Digest, for up-to-date information on how software defects affect real-world systems and their users.

dictionary.com, for standard English words.

serc.carleton.edu/introgeo/socratic/, an explanation of the Socratic method of teaching (also applicable to writing test plans and test cases).

www.rbccs-us.com, to read various articles related to topics in this book.

Appendix B. HELLOCARMS The Next Generation of Home Equity Lending

System Requirements Document

Table of Contents

II Versioning

III Glossary

000 Introduction

001 Informal Use Case

003 Scope

004 System Business Benefits

010 Functional System Requirements

020 Reliability System Requirements

030 Usability System Requirements

040 Efficiency System Requirements

050 Maintainability System Requirements

060 Portability System Requirements

A Acknowledgement

System Requirements Document

This document contains proprietary and confidential material of RBCS, Inc. Any unauthorized reproduction, use, or disclosure of this material, or any part thereof, is strictly prohibited. This document is solely for the use of RBCS employees, authorized RBCS course attendees, and purchasers of Rex Black's book.

Table of Contents

I	Table of Contents	354
II	Versioning	355
III	Glossary	356
000	Introduction	357
001	Informal Use Case	359
003	Scope	361
004	System Business Benefits	362
010	Functional System Requirements	363
020	Reliability System Requirements	368
030	Usability System Requirements	369
040	Efficiency System Requirements	370
050	Maintainability System Requirements	371
060	Portability System Requirements	372
A	Acknowledgement	373

II Versioning

Ver.	Date	Author	Description	Approval By/On
0.1	Nov 1, 2007	Rex Black	First Draft	
0.2	Dec 15, 2007	Rex Black	Second Draft	
0.5	Jan 1, 2008	Rex Black	Third Draft	

III Glossary

Term ^[1]	Definition
Home Equity	<p>The difference between a home's fair market value and the unpaid balance of the mortgage and any other debt secured by the home.</p> <p>A homeowner can increase their home equity by reducing the unpaid balance of the mortgage and any other debt secured by the home. Home equity can also increase if the property appreciates in value.</p> <p>A homeowner can borrow against home equity using home equity loans, home equity lines of credit, and reverse mortgages (see below).</p>
Secured Loan	Any loan where the borrower uses an asset as collateral for the loan. The loan is secured by the collateral in that the borrower can make a legal claim on the collateral if the borrower fails to repay the loan.
Home Equity Loan	A lump sum of money, disbursed at the initiation of the loan and lent to the homeowner at interest. A home equity loan is a secured loan, secured by the equity in the borrower's home.
Home Equity Line of Credit	A variable amount of money with a prearranged maximum amount, available for withdrawal by the homeowner on an as-needed basis and lent to the homeowner at interest. A home equity line of credit allows the homeowner to take out, as needed, a secured loan, secured by the equity in the borrower's home.
Mortgage	A legal agreement by which a sum of money is lent for the purpose of buying property and against which property the loan is secured.
Reverse Mortgage	A mortgage in which a homeowner borrows money in the form of regular payments which are charged against the equity of the home, typically with the goal of using the equity in the home as a form of retirement fund. A reverse mortgage results in the homeowner taking out a regularly increasing secured loan, secured by the equity in the borrower's home.

[] *These definitions are adapted from www.dictionary.com.*

000 Introduction

The Home Equity Loan, Line-of-Credit, and Reverse Mortgage System - (HELLOCARMS), as to be deployed in the first release, allows Globobank Telephone Bankers in the Globobank Fairbanks call center to accept applications for home equity products (loans, lines of credit, and reverse mortgages) from customers. The second release will allow applications over the Internet, including from Globobank business partners as well as customers themselves.

At a high level, the system is configured as shown in [Figure 1](#).

The HELLOCARMS application itself is a group of Java programs and assorted interfacing glue that run on the Web server. The Database server provides storage as the Application is processed, while the Application server offloads gateway activities to the clients from the Web server.

Figure A1-1. HELLOCARMS System (First Release)



001 Informal Use Case

The following informal use case applies for typical transactions in the HELLOCARMS System:

- 1.** The Telephone Banker interviews the Customer, entering information into the HELLOCARMS System through a Web browser interface on their Desktop. If the Customer is requesting a large loan or borrowing against a high-value property, the Telephone Banker escalates the application to a Senior Telephone Banker who decides whether to proceed with the application.
- 2.** Once the Telephone Banker has gathered the information from the Customer, the HELLOCARMS System determines the credit-worthiness of the Customer using the Scoring Mainframe.
- 3.** Based on all of the Customer information, the HELLOCARMS System displays various Home Equity Products (if any) that the Telephone Banker can offer to the customer.
- 4.** If the Customer chooses one of these Products, the Telephone Banker will conditionally confirm the Product.
- 5.** The interview ends. The Telephone Banker directs the HELLOCARMS System to transmit the loan information to the Loan Document Printing -System (LoDoPS) in the Los Angeles Datacenter for origination.
- 6.** The HELLOCARMS System receives an update from the LoDoPS System when the -following events occur:
 - 1.** LoDoPS System sends documents to customer.
 - 2.** Globobank Loan Servicing Center receives signed documents from customer; and,
 - 3.** Globobank Loan Servicing Center sends a check or other materials as appropriate to the Customer's product selection.

Once the Globobank Loan Servicing Center has sent the funds or other materials to the Customer, HELLOCARMS processing on the application is complete, and the system will not track subsequent loan-related activities for this -Customer.

Once HELLOCARMS processing on an application is complete, HELLOCARMS shall archive the application and all information associated with it. This applies whether the application was declined by the bank, cancelled by the customer, or ultimately converted into an active loan/line of credit/reverse mortgage.

003 Scope

The scope of the HELLOCARMS project includes:

- Selecting a COTS solution from a field of five vendors.
- Working with the selected application vendor to modify the solution to meet Globobank's requirements.
- Providing a browser-based front end for loan processing access from the Internet, existing Globobank call centers, outsourced (non-Globobank) call centers, retail banking centers, and brokers. However, the HELLOCARMS first release will only provide access from a Globobank call center (specifically Fairbanks).
- Developing an interface to Globobank's existing Scoring Mainframe for scoring a customer based on their loan application and HELLOCARMS features.
- Developing an interface to use Globobank's existing underwriting and origination system, Loan Document Printing System (LoDoPS), for document preparation. This interface allows the HELLOCARMS system, after assisting the customer with product selection and providing preliminary approval to the customer, to forward the preapproved application (for a loan, line of credit, or reverse mortgage) to the LoDoPS and to subsequently track the application's movement through to the servicing system.
- Receiving customer-related data from the Globobank Rainmaker Borrower Qualification Winnow (GloboRainBQW) system to generate outbound offers to potential (but not current) Globobank customers via phone, email, and paper mail.

004 System Business Benefits

The business benefits associated with the HELLOCARMS include:

- Automating a currently manual process, and allowing loan inquiries and applications from the Internet and via call center personnel (both from the current call centers and potentially from outsourced call centers, retail banking centers, and loan brokers).
- Decreasing the time to process the front-end portion of a loan from approximately 30 minutes to 5 minutes. This will allow Globobank's Consumer Products Division to dramatically increase the volumes of loans processed to meet its business plan.
- Reducing the level of skill required for the Telephone Banker to process a loan application, because the HELLOCARMS will select the product, decide whether the applicant is qualified, suggest alternative loan products, and provide a script for the Telephone Banker to follow.
- Providing online application status and loan tracking through the origination and document preparation process. This will allow Telephone Banker to rapidly and accurately respond to customer inquiries during the processing of their application.
- Providing the capability to process all products in a single environment.
- Providing a consistent way to make decisions about whether to offer loan products to customers, and if so, what loan products to offer customers, reducing processing and sales errors.
- Allowing Internet-based customers (in subsequent releases) to access Globobank products, select the preferred product, and receive a tentative loan approval within seconds.

The goal of the HELLOCARMS System's business sponsors is to provide these benefits for approximately 85% of the customer inquiries, with 15% or fewer inquiries escalate to a Senior Telephone Banker for specialized processing.

010 Functional System Requirements

The capability of the system to provide functions which meet stated and implied needs when the software is used under specified conditions.

ID	Description	Priority
010-010	Suitability	
010-010	Allow Telephone Bankers to take applications for home equity loans, lines of credit, and reverse mortgages.	1
010-010-020	Provide screens and scripts to support Call Center personnel in completing loan applications.	1
010-010-030	If the customer does not provide a "How Did You Hear About Us" identifier code, collect the lead information during application processing via a drop-down menu, with well-defined lead source categories.	2
010-010-040	Provide data validation, including the use of appropriate user interface (field) controls as well as back-end data validation. Field validation details are described in a separate document.	1
010-010-050	Display existing debts to enable retirement of selected debts for debt consolidation. Pass selected debts to be retired to LoDoPS as stipulations.	1
010-010-060	Allow Telephone Bankers and other Globobank telemarketers and partners to access incomplete or interrupted applications.	2
010-010-070	Ask each applicant whether there is an existing relationship with Globobank; e.g., any checking or savings accounts. Send existing Globobank customer relationship information to the Globobank Loan Applications Data Store (GLADS).	2
010-010-080	Maintain application status from initiation through to rejection, decline, or acceptance (and, if accepted, to delivery of funds).	2
010-010	Allow user to abort an application. Provide an abort function on all screens.	3

090		
010	Allow user to indicate on a separate screen which, if any, are existing debts that the customer will retire using the funds for which the customer is applying. Allow user the option to exclude specific debts and to include specific debts. For debts to be retired, send a stipulation to LoDoPS that specifies which debts that the customer must pay with loan proceeds.	3
010	Exclude a debt's monthly payment from the debt ratio if the customer requests the debt to be paid off.	3
110		
010	Provide a means of requesting an existing application by customer identification number if a customer does not have their loan identifier.	4
010	Direct the Telephone Banker to transfer the call to a Senior Telephone Banker if an application has a loan amount greater than \$500, 000; such loans require additional management approval.	1
010	Direct the Telephone Banker to transfer the call to a Senior Telephone Banker if an application concerns a property with value greater than \$1, 000,000; such applications require additional management approval.	2
010	Provide inbound and outbound telemarketing support for all States, Provinces, and Countries in which Globobank operates.	2
010	Support brokers and other business partners by providing limited partner-specific screens, logos, interfaces, and branding.	2
010	Support the submission of applications via the Internet, which includes the capability of untrained users to properly enter applications.	3
010	Provide features and screens that support the operations of Globobank's retail branches.	4
180		
010	Support the marketing, sales, and processing of home equity applications.	1
190		
010	Support the marketing, sales, and processing of home	2

010-	equity line of credit applications.	
200		
010-	Support the marketing, sales, and processing of home equity reverse mortgage applications.	3
210		
010-	Support the marketing, sales, and processing of applications for combinations of financial products (e.g., home equity and credit cards).	4
220		
010-	Support the marketing, sales, and processing of applications for original mortgages.	5
230		
010-	Support the marketing, sales, and processing of preapproved applications.	4
240		
010-	Support flexible pricing schemes including introductory pricing, short-term pricing, and others.	5
250		
010-	Accuracy	
020		
010-	Determine the various loans, lines of credit, and/or reverse mortgages for which a customer qualifies, and present these options for the customer to evaluate, with calculated costs and terms. Make qualification decisions in accordance with Globobank credit policies.	1
020-		
010		
010-	Determine customer qualifications according to property risk, credit score, loan-to-property-value ratio, and debt-to-income ratio, based on information received from the Scoring Mainframe.	1
020-		
020		
010-	During the application process, estimate the monthly payments based on the application information provided by the customer, and include the estimated payment as a debt in the debt-to-income calculation for credit scoring.	2
020-		
030		
010-	Add a loan fee based on property type:	3
020-		
040	<ul style="list-style-type: none"> • 1.5% for rental properties (duplex, apartment, and vacation) • 2.5% for commercial properties. 	

- 3.5% for condominiums or cooperatives.
- 4.5% for undeveloped property.

	Do not add a loan fee for the other supported property type, residential single family dwelling	
010- 020- 050	Capture all government retirement fund income(s) (e.g., Social Security in United States) as net amounts, but convert those incomes to gross income(s) in the interface to LoDoPS. [Note: This is because most government retirement income is not subject to taxes, but gross income is used in debt-to-income calculations.]	1
010- 020- 060	Capture the length of time (rounded to the nearest month) that the customer has received additional income (other than salary, bonuses, and retirement), if any.	3
010- 030	Interoperability	
010- 030- 010	If the customer provides a "How Did You Hear About Us" identifier code during the application process, retrieve customer information from GloboRainBQW.	2
010- 030- 020	Accept joint applications (e.g., partners, spouses, relatives, etc.) and score all applicants using the Scoring Mainframe.	1
010- 030- 030	Direct Scoring Mainframe to remove duplicate credit information from joint applicant credit reports.	2
010- 030- 040	Allow user to indicate on a separate screen which, if any, are existing debts that the customer will retire using the funds for which the customer is applying. Allow user the option to exclude specific debts and to include specific debts. For debts to be retired, send a stipulation to LoDoPS that specifies which debts that the customer must pay with loan proceeds.	1
010- 030- 060	If the Scoring Mainframe does not show a foreclosure or bankruptcy discharge date and the Customer indicates that the foreclosure or bankruptcy is discharged, continue processing the application, and direct the Telephone Banker to ask the applicant to provide proof of discharge in paperwork sent to LoDoPS.	3

010-	Allow user to indicate on a separate screen which, if any, are existing debts that the customer will retire using the funds for which the customer is applying. Allow user the option to exclude specific debts and to include specific debts. For debts to be retired, send a stipulation to LoDoPS that specifies which debts that the customer must pay with loan proceeds.	3
010-	Capture all government retirement fund income(s) (e.g., Social Security in United States) as net amounts, but convert those incomes to gross income(s) in the interface to LoDoPS. [Note: This is because most government retirement income is not subject to taxes, but gross income is used in debt-to-income calculations.]	1
010-	Pass application information to the Scoring Mainframe.	1
030-		
090		
010-	Receive scoring and decision information back from the Scoring Mainframe.	1
030-		
100		
010-	If the Scoring Mainframe is down, queue application information requests.	2
030-		
110		
010-	Initiate the origination process by sending the approved loan to LoDoPS.	2
030-		
120		
010-	Pass all declined applications to LoDoPS.	2
030-		
130		
010-	Receive LoDoPS feedback on the status of applications.	2
030-		
140		
010-	Receive changes to loan information made in LoDoPS (e.g., loan amount, rate, etc.).	2
030-		
145		
010-	Support computer-telephony integration to provide customized marketing and sales support for inbound telemarketing campaigns and branded business partners.	4
030-		
150		
010-	Security	

010	Support agreed-upon security requirements (encryption, firewalls, etc.).	2
010	Track "Created By" and "Last Changed By" audit trail information for each application.	1
020		
010	Allow outsourced telemarketers to see the credit tier but disallow them from seeing the actual credit score of applicants.	2
040		
030		
010	Support the submission of applications via the Internet, providing security against unintentional and intentional security attacks.	2
040		
050		
010	Allow Internet users to browse potential loans without requiring such users to divulge personal information such as name, government identifying numbers, etc. until the latest feasible point in the application process.	4
040		
060		
010	Support fraud detection for processing of all financial applications.	1
040		
050		
	[To be determined in a subsequent revision]	
[]		

[] Priorities are:

1. Very high
2. High
3. Medium
4. Low
5. Very Low

020 Reliability System Requirements

The capability of the system to maintain a specified level of performance when used under specified conditions.

ID	Description	Priority
020-010	Maturity	
	[To be determined in a subsequent revision]	
020-020	Fault-tolerance	
	[To be determined in a subsequent revision]	
020-030	Recoverability	
	[To be determined in a subsequent revision]	
020-040	Compliance (reliability standards/laws/reg)	
	[To be determined in a subsequent revision]	

030 Usability System Requirements

The capability of the system to be understood, learned, used, and attractive to the user and the call center agents when used under specified conditions.

ID	Description	Priority
030-010	Understandability	
030-010-010	Support the submission of applications via the Internet, including the capability for untrained users to properly enter applications.	2
	[More to be determined in a subsequent revision]	
030-020	Learnability	
	[To be determined in a subsequent revision]	
030-030	Operability	
030-030-010	Provide for complete customization of the user interface and all user supplied documents for business partners, including private branding of the sales and marketing information and all closing documents.	3
	[More to be determined in a subsequent revision]	
030-040	Attractiveness	
	[To be determined in a subsequent revision]	
030-050	Compliance (usability standards)	
030-050-010	Comply with local handicap-access laws.	5

040 Efficiency System Requirements

The capability of the system to provide appropriate performance, relative to the amount of resources used under stated conditions.

ID	Description	Priority
040-010	Time behavior	
040-010-010	Provide the user with screen-to-screen response time of one second or less. This requirement should be measured from the time the screen request enters the application system until the screen response departs the application server; i.e., do not include network transmission delays.	2
040-010-020	Provide an approval or decline for applications within 5 minutes of application submittal.	2
040-010-030	Originate the loan, including the disbursal of funds, within one hour.	3
	[More to be determined in a subsequent revision]	
040-020	Resource utilization	
040-020-010	Handle up to 2, 000 applications per hour.	2
040-020-020	Handle up to 4, 000 applications per hour.	3
040-020-030	Support a peak of 4, 000 simultaneous (concurrent) application submissions.	4
040-020-040	Support a total volume of 1.2 million approved applications for the initial year of operation.	2
040-020-050	Support a total volume of 7.2 million applications during the initial year of operation.	2

040-	Support a total volume of 2.4 million conditionally approved applications for the initial year of operation.	2
------	--	---

 060 | [More to be determined in a subsequent revision]] |

040-	Compliance (performance standards)
------	------------------------------------

 030 | [To be determined in a subsequent revision] |

050 Maintainability System Requirements

The capability of the system to be modified. Modifications may include corrections, improvement, or adaptations of the software changes in environments and in requirements and functional specifications.

ID	Description	Priority
050-010	Analyzability	
	[To be determined in a subsequent revision]	
050-020	Changeability	
	[To be determined in a subsequent revision]	
040-030	Compliance (performance standards)	
	[To be determined in a subsequent revision]	

060 Portability System Requirements

The capability of the system to be transferred from one environment to another.

ID	Description	Priority
060-010	Adaptability	
	[To be determined in a subsequent revision]	
060-020	Installability	
	[To be determined in a subsequent revision]	
060-030	Co-existence	
060-030-010	Should not interact in any nonspecified way with any other applications in the Globobank call centers or data centers.	1
	[More to be determined in a subsequent revision]	
060-040	Replaceability	
	Not applicable	
060-050	Compliance	
	[To be determined in a subsequent revision]	

A Acknowledgement

This document is based on an actual project. RBCS would like to thank their client, who wishes to remain unnamed, for their permission to adapt and publish anonymous portions of various project documents.

Appendix C. Answers to Sample Questions

chapter 1

- 1.** A
- 2.** A, B, E
- 3.** D, E
- 4.** A
- 5.** C
- 6.** D

Chapter 2

- 1.** A, B, D
- 2.** A
- 3.** A
- 4.** B

chapter 3

- 1.** C
- 2.** B
- 3.** A
- 4.** D
- 5.** A
- 6.** C
- 7.** B
- 8.** A, C
- 9.** A

10. C

11. A, B, E

12. B

13. D

14. C

15. A, C, E

16. D

17. A, C, D

18. D

19. D

20. A

21. B

22. A

23. C

24. D

chapter 4

None

chapter 5

None

chapter 6

1. B

2. D

3. A, B, E

4. B

Chapter 7

1. C

2. A

chapter 8

1. C

2. A

3. D

4. A

Chapter 9

1. A

2. A, B, C

3. D

Chapter 10

1. B, C

2. B, C, D, F

3. A

4. B

5. B

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W]

[SYMBOL]

61508 standard

[top]

[A]

acceptance testing
accepting risk and consequence
accessibility testing, defined
accuracy testing, defined
accuracy, defined
action-word-drive test automation
actual vs. expected results
adapting standards
Advanced exams, about
aggregate risk priority, determining 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th
alignment of methodology in distributed testing
alignment with lifecycle activities 2nd 3rd 4th 5th 6th 7th 8th
analytical testing strategies
architecture for test automation
as factors in test estimation
as form of insurance
as independent testing
assessing risk 2nd
assessing skills
assessment of risk 2nd
assignments to team members
assumption for test estimation
attacks, defined

audits 2nd
independent evaluation (audits)
audits
authors of reviews
automated
automated testing of
automating 2nd 3rd
availability risks
averages, industry-level
avionics industry standards

[top]

[B]

baseline metrics, defining
basis measurements
benefits of 2nd
benefits, costs, and challenges 2nd
beta testing 2nd
black-box testing, defined
bottom-up test estimation
branch testing, defined
breakdown structure
BS 7925/2 standard 2nd 3rd 4th 5th 6th 7th 8th
buddy testing
budget negotiation
bug convergence charts 2nd
bug hunts, risk-based testing as
building teams
building test teams
bus detection, costs of
business culture, distributed testing and
business factors in risk identification
business skills
business, testing by
business-domain standards 2nd

[C]

CMM (Capability Maturity Model)

CMMI (Capability Maturity Model Integration) 2nd 3rd 4th 5th 6th 7th 8th
9th 10th 11th

capacity-related risk 2nd

case studies 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th
16th 17th 18th 19th

case studies of test tools 2nd

case study

preintegration testing (case study)

case study 2nd 3rd 4th

acceptance testing (case study) 2nd

case study

preconditions for starting (case study)

beta testing (case study)

preconditions for (case study)

readiness for test execution (case study)

preconditions for starting test execution (case study)

marketing component in beta testing (case study)

beta testing (case study)

case study 2nd 3rd 4th 5th 6th 7th 8th 9th 10th

custom (case study)

custom tool development (case study) 2nd

case study

categories of 2nd 3rd 4th 5th

categories of defects

categories of quality risk 2nd

categories of risk 2nd

categories of test tools

cause-effect graphing, defined

challenges of

change management, within critical test processes

changing based on metrics 2nd

characteristics and benefits of

checklists for reviews

classification of test tools
classification tree method, defined
classifying incidents 2nd
client interaction ethics
Capability Maturity Model (CMM) 2nd 3rd 4th
Defined maturity level (CMMI) 2nd
validation process (CMMI)
Quantitatively Managed maturity level (CMMI)
Optimizing maturity level (CMMI)
Managed maturity level (CMMI)
code coverage metrics
code reviews
colleague interaction ethics
combinatorial tests
combined with master test plan
comments, motivating
communicating metrics to
communication skills 2nd
comparing actual results with
complexity, systems of systems projects 2nd
complexity, test automation projects
component integration testing
component interface risk
component testing, risk identification
concepts
concepts of
condition determination testing, defined
condition testing, defined
confidence 2nd
configuration-related risk
configuring the test environment
consultative test strategies
contingency, for risk control 2nd
continuous process improvement models 2nd
control flow analysis, defined
controlling risk 2nd 3rd
"correct" results, identifying 2nd
cost of exposure 2nd
cost of quality 2nd
costs of 2nd

coverage metrics 2nd

coverage risks

creating test cases 2nd

CTP (Critical Testing Processes) 2nd

cultural differences in distributed testing

cutover-related risk

[top]

[D]

D-D path, defined

daily closure period value

data flow analysis, defined

data flow coverage

data quality risk 2nd

date and time handling, risk with

debriefs with session testing

debugging tools

decision table testing, defined

decision testing, defined

defect fields 2nd

defect lifecycle

defect metrics 2nd

DRE (defect removal effectiveness)

defect-based test techniques 2nd

defect-based test techniques, defined

defined 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th
16th 17th 18th 19th 20th 21st 22nd 23rd 24th 25th 26th 27th 28th 29th
30th 31st 32nd 33rd 34th 35th 36th 37th 38th 39th 40th 41st 42nd 43rd
44th 45th 46th 47th 48th 49th 50th 51st 52nd 53rd 54th 55th 56th 57th
58th 59th 60th 61st 62nd 63rd 64th 65th 66th 67th 68th 69th 70th 71st
72nd 73rd 74th 75th 76th 77th 78th 79th 80th 81st

defining metrics and measures

delaying factors 2nd

Deming's cycle

dependencies of test tools

design coverage metrics

design reviews
detail test plan documents 2nd
detailed design reviews
detectability of defects 2nd
detection, costs of
disaster handling and recovery 2nd
discovery of incidents 2nd
distributed testing
distributed testing and 2nd 3rd
distributed, outsourced, and insourced testing 2nd 3rd 4th 5th
DO-178B standard 2nd 3rd 4th 5th 6th 7th 8th 9th
Severe level (DO-178B standard) 2nd
criticality levels (DO-178B standard)
Minor level (DO-178B standard) 2nd
Major level (DO-178B standard) 2nd
Hazardous and Severe level (DO-178B standard) 2nd
criticality levels (DO-178B standard)
Catastrophic level (DO-178B standard) 2nd
documentation
documentation and 2nd 3rd
documentation and project timeline
documentation during
documentation of 2nd
documenting 2nd 3rd 4th 5th 6th
documenting in master test plan 2nd
documenting in test policy
domain skills
domain-specific standards 2nd
downstream risk identification processes
defect removal effectiveness (DRE) 2nd
duration, project
dynamic test points
dynamic testing

[top]

[E]

efficiency testing, defined
email communications
employer interaction ethics
emulators
environmental coverage metrics
environments for testing
error guessing, defined
error handling and recovery 2nd
error, defined
ethics
public interest (ethics)
evaluating people with defect metrics
evaluating with defect metrics
evaluation of exit criteria and reporting
backbone integration in system of systems (example)
building teams (example) 2nd
example with integrated testing
exams, about
exercise and debrief 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th
14th 15th 16th 17th 18th 19th 20th 21st 22nd 23rd 24th 25th 26th 27th
28th 29th 30th 31st 32nd 33rd
exercises and debriefs 2nd 3rd 4th 5th
exploratory testing
exposure cost
external failure, costs of
external test organizations

[top]

[F]

FAA DO-178B standard 2nd
factors that affect 2nd
failure metrics
FMEA (failure mode and effect analysis)
FMECA (failure mode, effect and criticality analysis)
failure of reviews
failure, costs of

failure, defined
fault injection tools
fault seeding tools
FDA Title 21 CFR Part 820
feedback, motivating
flexibility in automated testing
failure mode and effect analysis (FMEA)
FMEA and
failure mode, effect and criticality analysis (FMECA)
"following the sun"
for design 2nd
for reviews 2nd
for test estimation
for test execution
formal reviews 2nd 3rd
formality of testing, cost of
fulfillment charts
function points
functional design reviews
functionality-related risk 2nd
fundamental test process, risk-based testing in 2nd

[top]

[G]

gaps and overlaps in distributed testing
gaps in distributed testing
generalists, teams of
generality of test management techniques
generic framework for 2nd
graphical displays of metrics

[top]

[H]

handoffs, identifying
hardware dependencies of test tools
hardware emulators
hardware requirements for nonfunctional testing
hazard analysis
heuristic test strategies
high-level design reviews
hiring team members
history of
how to do
how to use 2nd 3rd
HSIA standard

[top]

[I]

identification of incidents
identification of risk 2nd 3rd 4th
identifying problems, during test planning
identifying touchpoints and handoffs
IEEE 1028 standard
IEEE 1044 classifications 2nd 3rd 4th
IEEE 1044 standard 2nd
IEEE 610 standard
IEEE 829 specifications for 2nd 3rd
IEEE 829 standard 2nd 3rd 4th 5th 6th
IEEE 829 standard on 2nd 3rd
IEEE 829 templates 2nd 3rd
IEEE 829 test plan templates
ignoring risk
improving testing 2nd
scenario-based questions (in exams)
in nonfunctional testing
in reviews 2nd
in risk analysis 2nd

in risk identification 2nd
in risk identification and analysis 2nd 3rd 4th
in risk prioritization
in risk-based testing strategy
in test planning
management overhead (in test point analysis)
perceived (inaccurately)
Resolution (incident classification)
Repeatability (incident classification)
Project Phase (incident classification)
Project Activity (incident classification)
Product Status (incident classification)
Disposition (incident classification)
Corrective Action (incident classification)
Actual Cause (incident classification)
Type (incident classification)
Symptom (incident classification)
Suspected Cause (incident classification)
Source (incident classification)
incident classifications 2nd
Project Schedule (incident impact classification)
Project Risk (incident impact classification)
Project Quality/Reliability (incident impact classification)
Project Cost (incident impact classification)
Priority (incident impact classification)
Mission Safety (incident impact classification)
Customer Value (incident impact classification)
Societal (incident impact classification)
Severity (incident impact classification)
incident logging, defined 2nd 3rd
incident management
disposition phase (incident management)
action phase (incident management)
recognition phase (incident management)
investigation phase (incident management)
recognition phase (incident management)
investigation phase (incident management)
impact classifications (incident management)
disposition phase (incident management)
action phase (incident management)
incident management

incident reports 2nd
incidents 2nd
anomalies (incidents)
incidents
anomalies (incidents)
incidents 2nd
incremental lifecycle models 2nd 3rd 4th
independence of testing 2nd
individual skills 2nd
industry averages
informal reviews 2nd 3rd
initial costs of test automation
insourced testing
inspection leader, defined
inspections
installation-related risk
IEEE (Institute of Electrical and Electronics Engineering)
integrating testing into
integrating testing into software lifecycle
integration testing
interface risk
internal failure, costs of
ISO (International Organization for Standardization)
international standards
interoperability testing, defined
intolerable risks, knowing
introducing
introducing reviews
inventorying and managing
inventorying skills
irrational decision-making
irrational decision-making about risk
irrationality of 2nd 3rd
ISO 12207 standard
ISO 15504 standard
ISO 9126 standard
ISO/IEC standard 61508 2nd 3rd 4th 5th 6th 7th 8th
issues 2nd 3rd 4th
iterative (incremental) lifecycle models 2nd

[J]

job requirements
judgment, ethical

[top]

[K]

keyword-driven testing

[top]

[L]

language differences in distributed testing
LCSAJ (Linear Code Sequence and Jump), defined
learning from past projects
learning objectives, ISTQB Advanced exams
length of project
level of detail
level test plan
levels of risk
lifecycle of
lifecycle, incident management
lifelong learning
likelihood of problem 2nd
lines of code, function points and
list of
logging deviations from
logging deviations from plan
logging test results 2nd

[M]

maintainability testing, defined
maintenance of test tools
management ethics
management reviews
managing 2nd 3rd 4th 5th
managing skills
master test plan 2nd 3rd 4th 5th 6th 7th
material factors in test estimation
measurement, defined
mechanical life risk
medical systems standards
memory leak, defined
methodical test strategies
methodologies in distributed testing
resolved vs. reported defects (metric)
reported vs. resolved defects (metric)
cumulative defects reported (metric)
closure period (metric) 2nd
status of (metric)
confirmation test status (metric)
metric, defined
metrics and
metrics and measurement
metrics and measurement and
metrics and measurements for 2nd
metrics for 2nd 3rd 4th 5th 6th 7th
MISRA standard
missing or poor specifications
missing specifications, working with
mitigating risk 2nd
mixed
model-based testing
models for
moderator, review

monitoring test progress with
motivating comments
motivation 2nd
motivation and
Motor Industry Software Reliability Association (MISRA) standard.
multiple condition testing, defined,

[top]

[N]

national standards 2nd
negotiation
negotiation and test scope 2nd
negotiation with
no business value in
nonfunctional testing
nonfunctional testing issues
nonprescriptive process improvement models 2nd

[top]

[O]

OATs (operational acceptance tests), defined
of automation
of exposure
of incidents
of lessons learned
of quality
of reviews 2nd 3rd
of test formality
of testing
on defects
on test summary reports

open/closed charts
operating guide, master test plan and
operational acceptance tests (OATs), defined
operational profiles, defined
operations risk
outsourced testing 2nd 3rd
outsourcing as
overlaps in distributed testing
overreliance on test tools

[top]

[P]

pairwise testing, defined
Pareto analysis
path testing, defined
Payment Card Industry Data Security (PCI DSS) standard
people
perceived level of risk
performance-related risk 2nd
personality types
plan deviations, logging
project risks (planning risks) 2nd 3rd
poor specifications, working with
portability testing, defined
postponing
practical considerations
prescriptive process improvement models 2nd 3rd
prevention, costs of
preventive testing
principles of
procedure (test case) metrics
metrics for (procedure metrics)
process factors in test estimation
Plan phase (process improvement)
Do phase (process improvement)
Check phase (process improvement)

Act phase (process improvement)
Prioritize and Plan phase (process improvement)
Operate phase (process improvement)
Measure phase (process improvement)
Initiate phase (process improvement)
Define and Redefine phase (process improvement)
Validate phase (process improvement)
Evolve phase (process improvement)
process-compliant test strategies
product ethics
product risk metrics
of quality risks (product risks)
production environment for testing
professional ethics
professional pessimism
professionalism
progress charts
progress monitoring and control
progress monitoring case studies
project duration
project planning, test planning and
project retrospectives
planning risks (project risks)
project test points
project timeline, documentation and
PROOF acronym for session-based testing

[top]

[Q]

qualitative and quantitative values 2nd
quality policy
quality risk analysis 2nd
quality risk analysis and 2nd 3rd
quality risk analysis in 2nd
product risks (quality risks)
quality, cost of

[top]

[R]

ratings in skills
rational decision-making about risk
reactive test strategies
readers, in reviews
evolution of (reassessing) 2nd 3rd 4th 5th 6th
reassessing risk level 2nd 3rd 4th 5th 6th 7th 8th
recall, in taking exams
recoverability tests, defined
recurring costs of test automation
reducing 2nd
reducing test scope
regression testing
release decisions 2nd
release management
release notes
reliability risks
reliance on test tools
reporting
reporting case studies
requirements coverage metrics
requirements for
requirements reviews
requirements-based testing 2nd
results safeguarding
retrospectives, project
reusing test work products
recorders (review)
reviewers
reviews
secretary (reviews)

scribes (reviews)
managers (reviews)
reviews 2nd 3rd 4th 5th
reviews and
reviews preceding 2nd
right-shoring
impact of problem (risk analysis) 2nd
risk analysis
risk analysis and 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th
15th 16th 17th
risk assessment 2nd
risk control
risk coverage
risk coverage metrics 2nd
risk identification 2nd 3rd
risk identification and assessment 2nd
risk levels
risk mitigation 2nd
risk-based
risk-based testing 2nd 3rd 4th 5th
risk-based testing strategy
risk-based testing vs.
risks 2nd 3rd 4th 5th
robustness and generality of
robustness of management techniques
roles in reviews
rolling closure period value
root cause analysis
risk priority number (RPN) 2nd 3rd 4th 5th 6th 7th
rules in decision table

[top]

[S]

safeguarding 2nd
safeguarding information products
safeguarding logs 2nd

SIL (safety integrity level)
safety-critical system issues
safety-critical systems 2nd 3rd 4th 5th 6th 7th
SCCFA standard
schedule compression
scheduling
scope of testing
scripting languages
security considerations in nonfunctional testing
security testing, defined
self, ethics of
self-editing
self-testing
sequential lifecycle model 2nd
sequential lifecycle models 2nd
session reports (session sheets)
session-based testing 2nd 3rd 4th
session-based testing and
setup-related risk
severity of risk 2nd 3rd
SFMECA standard
SFTA standard
Shewhart cycle
signal quality risk
safety integrity level (SIL)
simulators
skills
skills for
skills inventories
skills, individual
Socratic approach to test planning
software attacks, defined
software characteristics tests, list of
software dependencies of test tools
software development knowledge and
software lifecycle
SPICE (Software Process Improvement and Capability Determination)
SQR (Software Quality Rank)
software requirements for testing
software skills
Software Usability Measurement Inventory (SUMI), defined

space industry standards
specialists, teams of
specifications, missing or poor
Software Process Improvement and Capability Determination (SPICE)
spiral lifecycle models 2nd
Software Quality Rank (SQR)
stability risks
staged process improvement models 2nd 3rd
stakeholder involvement
stakeholder involvement in 2nd 3rd 4th 5th 6th 7th 8th 9th
standard-compliant test strategies
standards bodies
state transition testing, defined
state-related risks 2nd
statement testing, defined
static analysis
static analyzers, defined
static test points
Systemic Test and Evaluation Process (STEP) 2nd
acquisition phase (STEP framework)
planning phase (STEP framework)
measurement phase (STEP framework)
STEP framework and
strategies
strategies for
strategies for testing
stress risk
structural test design techniques
subjective assessments of quality and testing
subsystem interface risk
success factors for
suitability testing, defined
suitability, defined
SUMI (Software Usability Measurement Inventory), defined
summary reports
system integration testing
system of systems issues
system of systems projects 2nd
STEP (Systemic Test and Evaluation Process) 2nd

[T]

tables of decisions
taxonomy of 2nd
team communication
team dynamics
teams of generalists
teams of specialists
technical and software skills
technical factors in risk identification
technical reviews
technical skills
technical support issues
technical support, testing by
technical-domain standards 2nd
techniques for 2nd
templates for
templates for test plans
test analysis and design 2nd
test analysts
test basis 2nd
test case (procedure) metrics 2nd
test cases vs. test procedures
test charters, defined
test closure 2nd
test conditions, defined
test control 2nd 3rd 4th 5th 6th 7th 8th
test control and
test coverage 2nd
test coverage metrics 2nd
test documentation 2nd
test estimation
test execution 2nd
test execution tools, defined
test fulfillment charts 2nd
test implementation 2nd 3rd
TIM (Test Improvement Model)
test improvement process

test incident reports 2nd
test item transmittal reports
test levels
test logging 2nd 3rd 4th
test logs, defined
test management issues 2nd 3rd
TMM (Test Maturity Model)
TMMI (Test Maturity Model Integration) 2nd
test oracles, defined
TOM (Test Organization Maturity)
test plan templates 2nd 3rd
test planning 2nd
test planning and 2nd 3rd 4th 5th 6th 7th 8th
test planning and control
test plans, defined 2nd
TPA (test point analysis)
weighting factors (test point analysis) 2nd
productivity factor (test point analysis)
function-dependent weighting factors (test point analysis)
environmental weight factor (test point analysis)
test policy documents 2nd 3rd
test cases (test procedures) 2nd
TPI (Test Process Improvement) 2nd 3rd
test progress charts 2nd
test release management
test scope
test scripts, defined
test sessions
test strategies
test strategy aligned with
test strategy documents
test suites, automating
test summary reports 2nd
test system development
test team development
test team dynamics 2nd 3rd
test tools and 2nd 3rd
testing and
testing in production
testing skills 2nd
testing skills and

testing within an organization
tests of software characteristics, list of tests, based on risk
Test Improvement Model (TIM)
time handling, risk with
time pressures during
time zones, in distributed testing 2nd
time-boxed testing
TMap model
Test Maturity Model (TMM)
Optimization maturity level (TMM)
management and measurement maturity level (TMM)
integration maturity level (TMM)
definition maturity level (TMM)
Test Maturity Model (TMM)
Test Maturity Model Integration (TMMI)
tolerable risks, knowing
Test Organization Maturity (TOM)
tool development, postponing
tools for 2nd 3rd 4th 5th 6th 7th 8th
tools requirements for nonfunctional testing
top-down test estimation
touchpoints, identifying
test point analysis (TPA)
Test Process Improvement (TPI)
infrastructure and tools cornerstone (TPI)
Efficient maturity level (TPI)
Controlled maturity level (TPI)
Test Process Improvement (TPI)
organization and tools cornerstone (TPI)
Optimizing maturity level (TPI)
traceability
tracking metrics and measures
transaction-related risks 2nd
transference, for risk control
travel considerations in distributed testing
trend charts
troubleshooting tools
trust with distributed testing
types of 2nd 3rd

[top]

[U]

upstream risk identification processes
usability risk
usability testing, defined
use case testing, defined
user acceptance testing, risk identification
user interface risk 2nd
user skills
user, business, and domain skills
users, testing by
using cost of exposure 2nd 3rd
using FMEA 2nd
using hazard analysis 2nd 3rd
using industry averages
using ISO 9126 2nd
using metrics in 2nd 3rd

[top]

[V]

V-model
verifying end of testing
volume-related risk 2nd

[top]

[W]

W-model

walk-throughs

Web tools

when defects can be detected 2nd 3rd 4th

when they can be detected

white-box test techniques, defined

wild pointer, defined

with CTP 2nd

with FMEA

with STEP 2nd

with test tools

with TMM 2nd

with TPI 2nd

establishing context (within critical test process)

testing (within critical test process)

within critical test processes 2nd 3rd 4th 5th 6th

work breakdown structure

work breakdown structure for 2nd 3rd 4th 5th 6th 7th 8th

[top]