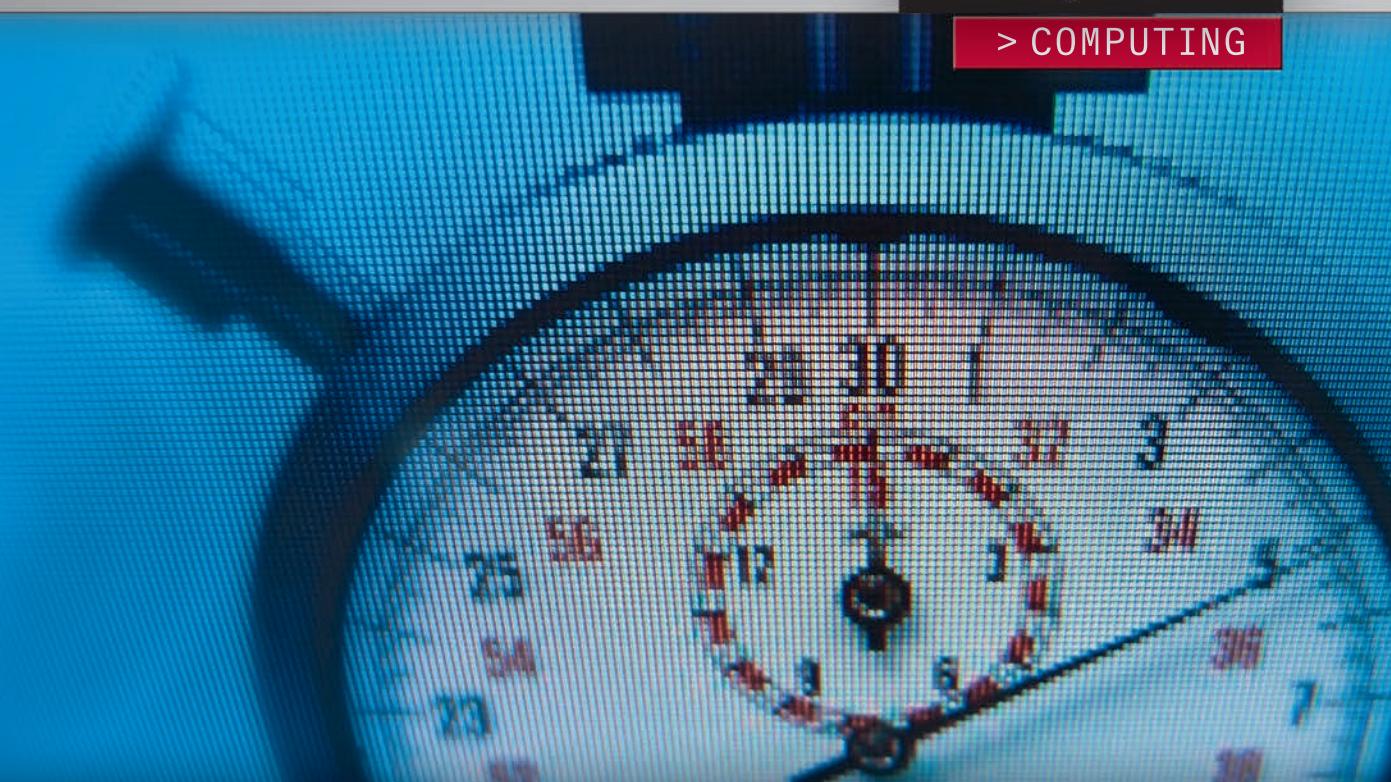


rockynook

> COMPUTING



2nd Edition

Jamie L. Mitchell · Rex Black

Advanced Software Testing Vol. 3

Guide to the ISTQB Advanced Certification
as an Advanced Technical Test Analyst



Advanced Software Testing—Vol. 3

2nd Edition

About the Authors



Jamie L. Mitchell

Jamie L. Mitchell has over 32 years of experience in developing and testing both hardware and software. He is a pioneer in the test automation field and has worked with a variety of vendor and open-source test automation tools since the first Windows tools were released with Windows 3.0. He has also written test tools for several platforms.

The former Lead Automation Engineer for American Express Distributed Integration Test / Worldwide, Jamie has successfully designed and implemented test automation projects for many top companies, including American Express, Mayo Clinic, IBM AS/400 division, ShowCase Corporation, and others. Jamie holds a Master of Computer Science degree from Lehigh University in Bethlehem, Pennsylvania, and a Certified Software Test Engineer certification from QAI.

He has been a frequent speaker on testing and automation at several international conferences, including STAR, QAI, and PSQT.



Rex Black

With over 30 years of software and systems engineering experience, Rex Black is president of RBCS (www.rbcus.com), a leader in software, hardware, and systems testing. For 20 years, RBCS has delivered consulting, outsourcing, and training services in the areas of software, hardware, and systems testing and quality. Employing the industry's most experienced and recognized consultants, RBCS conducts product testing, builds and improves testing groups, and provides testing staff for hundreds of clients worldwide. Ranging from Fortune 20 companies to start-ups,

RBCS clients save time and money through higher quality, improved product development, decreased tech support calls, improved reputation, and more.

As the leader of RBCS, Rex is the most prolific author practicing in the field of software testing today. He has written over 50 articles and ten books that have sold over 100,000 copies around the world, including numerous translations and foreign releases such as Japanese, Chinese, Indian, Spanish, Hebrew, Hungarian, and Russian editions. He is a popular speaker at conferences and events around the world.

Rex is the past president of the International Software Testing Qualifications Board (ISTQB) and of the American Software Testing Qualifications Board (ASTQB).

Jamie L. Mitchell · Rex Black

Advanced Software Testing—Vol. 3

**Guide to the ISTQB Advanced Certification
as an Advanced Technical Test Analyst**

2nd Edition

rockynook

Jamie L. Mitchell
jamie@go-jmc.com

Rex Black
rex_black@rbcs-us.com

Editor: Dr. Michael Barabas
Projectmanager: Matthias Rossmanith
Copyeditor: Judy Flynn
Layout and Type: Josef Hegele
Proofreader: Julie Simpson
Cover Design: Helmut Kraus, www.exclam.de
Printer: Sheridan
Printed in USA

ISBN: 978-1-937538-64-4

2nd Edition © 2015 by Jamie L. Mitchell and Rex Black

Rocky Nook
802 East Cota Street, 3rd Floor
Santa Barbara, CA 93103

www.rockynook.com

Library of Congress Cataloging-in-Publication Data
Black, Rex, 1964-
Advanced software testing : guide to the ISTQB advanced certification as an advanced technical test analyst / Rex Black, Jamie L. Mitchell.-1st ed.
p. cm.-(Advanced software testing)
ISBN 978-1-933952-19-2 (v. 1 : alk. paper)-ISBN 978-1-933952-36-9
(v. 2 : alk. paper)-ISBN 978-1-937538-64-4 (v. 3 : alk. paper)
1. Electronic data processing personnel-Certification. 2. Computer software-Examinations-Study guides. 3. Computer software-Testing. I. Title.
QA76.3.B548 2008
005.1'4-dc22
2008030162

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 use, 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.

Jamie Mitchell's Acknowledgements

What a long, strange trip it's been. The last 30 years have taken me from being a bench technician, fixing electronic audio components, to this time and place, where I have cowritten a book on some of the most technical aspects of software testing. It's a trip that has been both shared and guided by a host of people that I would like to thank.

To the many at both Moravian College and Lehigh University who started me off in my "Exciting Career in Computers," for your patience and leadership that instilled in me a burning desire to excel, I thank you.

To Terry Schardt, who hired me as a developer but made me a tester, thanks for pushing me to the dark side. To Tom Mundt and Chuck Awe, who gave me an incredible chance to lead, and to Barindralal Pal, who taught me that to lead was to keep on learning new techniques, thank you.

To Dean Nelson, who first asked me to become a consultant, and Larry Decklever, who continued my training, many thanks. A shout-out to Beth and Jan, who participated with me in "choir rehearsals" at Joe Senser's when things were darkest. Have one on me.

To my colleagues at TCQAA, SQE, and QAI who gave me chances to develop a voice while I learned how to speak, my heartfelt gratitude. To the people I am working with at ISTQB and ASTQB: I hope to be worthy of the honor of working with you and expanding the field of testing. Thanks for the opportunity.

In my professional life, I have been tutored, taught, mentored, and shown the way by a host of people whose names deserve to be mentioned, but the list is too abundant to recall every name. I would like to give all of you a collective thanks; I would be poorer for not knowing you.

To all the bosses who never fired me while I struggled to learn test automation—failing too many times to count—allowing me the luxury of learning the lessons that made me better at it: a collective shout-out to you. One specific mention: Kudos to Dag Roppe, my team leader, who let me build a keyword-

driven framework in the late 90's that was a wild success until the 16-bit tool finally maxed out and caused the whole architecture to fold in on itself: I appreciate your patience and regret you lost your job for letting me go so long on my seminal failure.

To Rex Black, for giving me a chance to coauthor the Advanced Technical Test Analyst course and this book: Thank you for your generosity and the opportunity to learn at your feet. For my partner in crime, Judy McKay: Even though our first tool attempt did not fly, I have learned a lot from you and appreciate both your patience and kindness. Hoist a fruity drink from me. To Laurel and Dena and Leslie: Your patience with me is noted and appreciated. Thanks for being there.

In the spirit of "It Takes a Village," no one really works alone today. While writing this book, I got help from several people who tried to keep me on the straight and narrow. I want to especially thank Amr Ali who added valuable input in his timely book reviews and gave me many valued suggestions during the review period. I would also like to thank Leslie Segal who provided deep insight into several topics while I worked on this book.

And finally, to my family, who have seen so much less of me over the last 30 years than they might have wanted, as I strove to become all that I could be in my chosen profession; words alone cannot convey my thanks. To Beano, who spent innumerable hours helping me steal the time needed to get through school and set me on the path to here, my undying love and gratitude. To my loving wife, Susan, who covered for me at many of the real-life tasks while I toiled, trying to climb the ladder: my love and appreciation. I might not always remember to say it, but I do think it. And to my kids, Christopher and Kimberly, who have always been smarter than me but allowed me to pretend that I was the boss of them, thanks. Your tolerance and enduring support have been much appreciated.

Last, and probably least, to "da boys," Baxter and Boomer, Bitzi and Buster, and now our renegade Mozart: Whether sitting in my lap while I was trying to learn how to test or sitting at my feet while I was writing this book, you guys have been my sanity check. You never cared how successful I was, as long as the doggie chow appeared in your bowls, morning and night. May your tails wag forever, here and over the rainbow bridge. Thanks.

Rex Black's 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 International Software Testing Qualifications Board. I continue to work on both boards. 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 sense of accomplishment 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 and the previous edition of it. The material in this book, in our Advanced Technical Test Analyst instructor-led training course, and in our Advanced Technical Test Analyst e-learning course were coauthored by Jamie Mitchell, and reviewed, re-reviewed, and polished with hours of dedicated assistance by Bernard Homés, Gary Rueda Sandoval, Jan Sabak, Joanna Kazun, Corné Kruger, Ed Weller, Dawn Haynes, José Mata, Judy McKay, Paul Jorgensen, and Pat Masters.

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

In the creation of the training courses and the materials that make up this book, Jamie and I have drawn on all the experiences we have had as authors, practitioners, consultants, and trainers. 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 20 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 pride for me but not a whole lot of money. Again, I hope that our mutual successes 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. It's hard to believe that my first book, *Managing the Testing Process*, was published the year before my first daughter was born, and now she is a 14-year-old young lady. My second daughter was born right about the time I first got involved with the ISTQB program, in 2002. As proud as I am of my books and my work with the ISTQB, well, those two girls are the greatest things I've ever been part of doing. My wife (who also happens to be my astute business partner) has done all of the hard work of keeping the plates spinning on the pencils while I do things like speak at conferences and write books. As Brad Paisley put it, "Now there are men who make history, there are men who change the world, and there are men like me that simply find the right girl." If I had nothing but my family, I'd still be the luckiest guy I know.

Of course, in every life, even the luckiest life, some rain will fall. Earlier this year, my family and I lost our dog Cosmo after a long—one might say interminable—series of illnesses spanning years. To all but the last of these maladies, he absolutely refused to succumb, due to the fact, as I'm sure he saw it, that there

was surely more food to steal from someone's plate. Within two weeks of Cosmo's sad departure, a much less predictable canine tragedy occurred. Our other dog, a rescue black Labrador named Hank, was stricken with a cruel, aggressive, and incurable cancer. He fought it with his usual brio and quirky style, enjoying every day of life. However, as cancer will do to dogs, it killed him in September as I was working my way through this book. Chase butterfly shadows and chipmunks in heaven, my friend, and maybe I'll see you there someday.

Table of Contents

Jamie Mitchell's Acknowledgements	v
Rex Black's Acknowledgements	vii
Introduction	xxi
1 The Technical Test Analyst's Tasks in Risk-Based Testing	1
1.1 Introduction	2
1.2 Risk Identification	4
1.3 Risk Assessment	7
1.4 Risk Mitigation or Risk Control	10
1.5 An Example of Risk Identification and Assessment Results	14
1.6 Risk-Aware Testing Standard	16
1.7 Sample Exam Questions	18
2 Structure-Based Testing	21
2.1 Introduction	22
2.1.1 Control Flow Testing Theory	24
2.1.2 Building Control Flow Graphs	25
2.1.3 Statement Coverage	29
2.1.4 Decision Coverage	33
2.1.5 Loop Coverage	37

2.1.6	Hexadecimal Converter Exercise	41
2.1.7	Hexadecimal Converter Exercise Debrief	43
2.2	Condition Coverage	43
2.3	Decision Condition Coverage	47
2.4	Modified Condition/Decision Coverage (MC/DC)	48
2.4.1	Complicating Issues: Short-Circuiting	55
2.4.2	Complicating Issues: Coupling	57
2.5	Multiple Condition Coverage	58
2.5.1	Control Flow Exercise	65
2.5.2	Control Flow Exercise Debrief	66
2.6	Path Testing	70
2.6.1	Path Testing via Flow Graphs	71
2.6.2	Basis Path Testing	77
2.6.3	Cyclomatic Complexity Exercise	82
2.6.4	Cyclomatic Complexity Exercise Debrief	83
2.7	API Testing	84
2.8	Selecting a Structure-Based Technique	93
2.8.1	Structure-Based Testing Exercise Debrief	101
2.9	A Final Word on Structural Testing	107
2.10	Sample Exam Questions	108
3	Analytical Techniques	115
3.1	Introduction	115
3.2	Static Analysis	116
3.2.1	Control Flow Analysis	116
3.2.2	Data Flow Analysis	127
3.2.2.1	Define-Use Pairs	130
3.2.2.2	Define-Use Pair Example	133

3.2.2.3	Data Flow Exercise	139
3.2.2.4	Data Flow Exercise Debrief.....	139
3.2.2.5	A Data Flow Strategy	140
3.2.3	Static Analysis to Improve Maintainability	144
3.2.3.1	Code Parsing Tools	144
3.2.3.2	Standards and Guidelines	147
3.2.4	Call Graphs	149
3.2.4.1	Call-Graph-Based Integration Testing	150
3.2.4.2	McCabe's Design Predicate Approach to Integration ...	153
3.2.4.3	Hex Converter Example	156
3.2.4.4	McCabe Design Predicate Exercise	161
3.2.4.5	McCabe Design Predicate Exercise Debrief	162
3.3	Dynamic Analysis	163
3.3.1	Memory Leak Detection	165
3.3.2	Wild Pointer Detection	167
3.3.3	Dynamic Analysis Exercise	168
3.3.4	Dynamic Analysis Exercise Debrief	168
3.4	Sample Exam Questions	169
4	Quality Characteristics for Technical Testing	177
4.1	Introduction	178
4.2	Security Testing	181
4.2.1	Security Issues	182
4.2.1.1	Piracy	182
4.2.1.2	Buffer Overflow	183
4.2.1.3	Denial of Service	184
4.2.1.4	Data Transfer Interception	185
4.2.1.5	Breaking Encryption	185
4.2.1.6	Logic Bombs/Viruses/Worms	186
4.2.1.7	Cross-Site Scripting	187

4.2.1.8	Timely Information	188
4.2.1.9	Internal Security Metrics	192
4.2.1.10	External Security Metrics	194
4.2.1.11	Exercise: Security	195
4.2.1.12	Exercise: Security Debrief	195
4.3	Reliability Testing	196
4.3.1	Maturity	201
4.3.1.1	Internal Maturity Metrics	202
4.3.1.2	External Maturity Metrics	203
4.3.2	Fault Tolerance	207
4.3.2.1	Internal Fault Tolerance Metrics	208
4.3.2.2	External Fault Tolerance Metrics	209
4.3.3	Recoverability	211
4.3.3.1	Internal Recoverability Metrics	213
4.3.3.2	External Recoverability Metrics	214
4.3.4	Compliance	216
4.3.4.1	Internal Compliance Metrics	216
4.3.4.2	External Compliance Metrics	216
4.3.5	An Example of Good Reliability Testing	217
4.3.6	Exercise: Reliability Testing	219
4.3.7	Exercise: Reliability Testing Debrief	219
4.4	Efficiency Testing	220
4.4.1	Multiple Flavors of Efficiency Testing	221
4.4.2	Modeling the System	225
4.4.2.1	Identify the Test Environment	227
4.4.2.2	Identify the Performance Acceptance Criteria	227
4.4.2.3	Plan and Design Tests	228
4.4.2.4	Configure the Test Environment	228
4.4.2.5	Implement the Test Design	229
4.4.2.6	Execute the Test	229
4.4.2.7	Analyze the Results, Tune and Retest	230

4.4.3	Time Behavior	230
4.4.3.1	Internal Time Behavior Metrics	235
4.4.3.2	External Time Behavior Metrics	236
4.4.4	Resource Utilization	238
4.4.4.1	Internal Resource Utilization Metrics	238
4.4.4.2	External Resource Utilization Metrics	239
4.4.5	Compliance	242
4.4.5.1	Internal Compliance Metric	242
4.4.5.2	External Compliance Metric	242
4.4.6	Exercise: Efficiency Testing	242
4.4.7	Exercise: Efficiency Testing Debrief	243
4.5	Maintainability Testing	244
4.5.1	Analyzability	248
4.5.1.1	Internal Analyzability Metrics	250
4.5.1.2	External Analyzability Metrics	250
4.5.2	Changeability	252
4.5.2.1	Internal Changeability Metrics	255
4.5.2.2	External Changeability Metrics	255
4.5.3	Stability	256
4.5.3.1	Internal Stability Metrics	257
4.5.3.2	External Stability Metrics	258
4.5.4	Testability	258
4.5.4.1	Internal Testability Metrics	259
4.5.4.2	External Testability Metrics	260
4.5.5	Compliance	261
4.5.5.1	Internal Compliance Metric	261
4.5.5.2	External Compliance Metric	261
4.5.6	Exercise: Maintainability Testing	261
4.5.7	Exercise: Maintainability Testing Debrief	262

4.6	Portability Testing	263
4.6.1	Adaptability	264
4.6.1.1	Internal Adaptability Metrics	266
4.6.1.2	External Adaptability Metrics	267
4.6.2	Replaceability	268
4.6.2.1	Internal Replaceability Metrics	270
4.6.2.2	External Replaceability Metrics	270
4.6.3	Installability	271
4.6.3.1	Internal Installability Metrics	274
4.6.3.2	External Installability Metrics	275
4.6.4	Coexistence	275
4.6.4.1	Internal Coexistence Metrics	276
4.6.4.2	External Coexistence Metrics	277
4.6.5	Compliance	278
4.6.5.1	Internal Compliance Metrics	278
4.6.5.2	External Compliance Metrics	278
4.6.6	Exercise: Portability Testing	278
4.6.7	Exercise: Portability Testing Debrief	278
4.7	General Planning Issues	280
4.8	Sample Exam Questions	285
5	Reviews	287
5.1	Introduction	287
5.2	Using Checklists in Reviews	292
5.2.1	Some General Checklist Items for Design and Architecture Reviews	296
5.2.2	Deutsch's Design Review Checklist	299
5.2.3	Some General Checklist Items for Code Reviews	301

5.2.4	Marick's Code Review Checklist	304
5.2.5	The Open Laszlo Code Review Checklist	306
5.3	Deutsch Checklist Review Exercise	308
5.4	Deutsch Checklist Review Exercise Debrief	308
5.5	Code Review Exercise	310
5.6	Code Review Exercise Debrief	311
5.7	Sample Exam Questions	316
6	Test Tools and Automation	319
6.1	Integration and Information Interchange between Tools	319
6.2	Defining the Test Automation Project	323
6.2.1	Preparing for a Test Automation Project	324
6.2.2	Why Automation Projects Fail	329
6.2.3	Automation Architectures (Data Driven vs. Keyword Driven)	335
6.2.3.1	The Capture/Replay Architecture	337
6.2.3.2	Evolving from Capture/Replay	341
6.2.3.3	The Simple Framework Architecture	343
6.2.3.4	The Data-Driven Architecture	346
6.2.3.5	The Keyword-Driven Architecture	348
6.2.4	Creating Keyword-Driven Tables	350
6.2.4.1	Building an Intelligent Front End	352
6.2.4.2	Keywords and the Virtual Machine	354
6.2.4.3	Benefits of the Keyword Architecture	356
6.2.4.4	Creating Keyword-Driven Tables Exercise	359
6.2.4.5	Keyword-Driven Exercise Debrief	360
6.3	Specific Test Tools	362
6.3.1	Fault Seeding and Fault Injection Tools	362
6.3.2	Performance Testing and Monitoring Tools	363
6.3.2.1	Data for Performance Testing	365

6.3.2.2	Building Scripts	366
6.3.2.3	Measurement Tools	367
6.3.2.4	Performing the Testing	368
6.3.2.5	Performance Testing Exercise.....	370
6.3.2.6	Performance Testing Exercise Debrief	370
6.3.3	Tools for Web Testing	373
6.3.4	Model-Based Testing Tools	374
6.3.5	Tools to Support Component Testing and Build Process	380
6.4	Sample Exam Questions	383
7	Preparing for the Exam	387

Appendix

A	Bibliography	401
	Advanced Syllabus Referenced Standards	401
	Advanced Syllabus Referenced Books	401
	Advanced Syllabus Other References.....	402
	Other Referenced Books.....	403
	Other References.....	404
	Referenced Standards	407

B	HELLOCARMS The Next Generation of Home Equity Lending	
	System Requirements Document	409
I	Table of Contents	411
II	Versioning	413
III	Glossary	415
000	Introduction	417
001	Informal Use Case	418
003	Scope	419
004	System Business Benefits	420
010	Functional System Requirements	421
020	Reliability System Requirements	425
030	Usability System Requirements	426
040	Efficiency System Requirements	427
050	Maintainability System Requirements	429
060	Portability System Requirements	430
A	Acknowledgement	431
C	Answers to Sample Questions	433
Index	435

Introduction

This is a book on advanced software testing for technical test analysts. By that we mean that we address topics that a technical practitioner who has chosen software testing as a career should know. We focus on those skills and techniques related to test analysis, test design, test tools and automation, test execution, and test results evaluation. We take these topics in a more technical direction than in the volume for test analysts, since we focus on test design using structural techniques and details about the use of dynamic analysis to monitor the current internal state of the system, as well as how it interfaces with the OS and other external entities. We assume that you know the basic concepts of test engineering, test design, test tools, testing in the software development life cycle, and test management. You are ready to mature your level of understanding of these concepts and to apply these advanced concepts to your daily work as a test professional.

This book follows the International Software Testing Qualifications Board's (ISTQB) Advanced Technical Test Analyst syllabus 2012, with a focus on the material and learning objectives for the advanced technical test analyst. As such, this book can help you prepare for ISTQB Advanced Level Technical Test Analyst exam. You can use this book to self-study for this exam 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 Technical Test Analyst 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, automation test engineer, manual test engineer, programmer, or in any other field where a sophisticated understanding of software testing is needed, especially an understanding of the particularly technical aspects of testing such as white-box testing and test automation, then this book is for you.

This book focuses on technical test analysis. It consists of seven chapters, addressing the following material:

1. The Technical Test Analyst’s Tasks in Risk-Based Testing
2. Structure-Based Testing
3. Analytical Techniques
4. Quality Characteristics for Technical Testing
5. Reviews
6. Test Tools and Automation
7. Preparing for the Exam

What should a technical test analyst 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?

- Recognize and classify typical risks associated with performance, security, reliability, portability and maintainability of software systems.
- Create test plans detailing planning, design, and execution of tests to mitigate performance, security, reliability, portability, and maintainability risks.
- Select and apply appropriate structural design techniques so tests provide adequate levels of confidence, via code coverage and design coverage.
- Effectively participate in technical reviews with developers and architects, applying knowledge of typical mistakes made in code and architecture.
- Recognize risks in code and architecture and create test plan elements to mitigate those risks via dynamic analysis.
- Propose improvements to security, maintainability, and testability of code via static analysis.
- Outline costs and benefits expected from types of test automation.
- Select appropriate tools to automate technical testing tasks.
- Understand technical issues and concepts in test automation.

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

In writing this book, we’ve kept foremost in our minds the question of how to make this material useful to you. If you are using this book to prepare for an ISTQB Advanced Level Technical Test Analyst exam, then we recommend that you read Chapter 7 first, then read the other six chapters in order. If you are using this book to expand your overall understanding of testing to an advanced and highly technical level but do not intend to take an ISTQB Advanced Level

Technical Test Analyst exam, then we recommend that you read Chapters 1 through 6 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 six chapters is divided into sections. For the most part, we have followed the organization of the ISTQB Advanced Syllabus to the point of section divisions, but subsection 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 7.

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 we've included practical, real-world exercises for the key concepts. We 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 testing professional only by applying advanced test techniques to actual software testing.

1 The Technical Test Analyst's Tasks in Risk-Based Testing

Wild Dog ... said, “I will go up and see and look, and say; for I think it is good. Cat, come with me.”

“Nenni!” said the Cat. “I am the Cat who walks by himself, and all places are alike to me. I will not come.”

Rudyard Kipling, from his story “The Cat That Walked by Himself,” a colorful illustration of the reasons for the phrase “herding cats,” which is often applied to describe the task of managing software engineering projects.

The first chapter of the Advanced Technical Test Analyst syllabus is concerned with your role as a technical test analyst in the process of risk-based testing. There are four sections.

1. Introduction
2. Risk Identification
3. Risk Assessment
4. Risk Mitigation

While the Advanced Technical Test Analyst syllabus does not include K3 or K4 learning objectives related to risk-based testing for technical test analysts in this chapter, in Chapter 4 you’ll have a chance to identify quality risks related to various nonfunctional quality characteristics.

There is a common learning objective that we will be covering in each applicable section. Therefore, we list it here to avoid listing it in each section.

Common Learning Objective

TTA-1.x.1 (K2) Summarize the activities of the technical test analyst within a risk-based approach for planning and executing testing.

Let's look at these four sections now.

1.1 Introduction

Learning objectives

Recall of content only.

The decision of which test strategies to select is generally the purview of the test manager, and thus this topic is covered in detail in *Advanced Software Testing: Volume 2*. It is often the case that a test manager will select a strategy called analytical risk-based testing. Specifically, the test manager will usually select one of the techniques described in that book to implement risk-based testing, such as Pragmatic Risk Analysis and Management (PRAM), which we will use for examples in this chapter.

ISTQB Glossary

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

product risk: A risk directly related to the test object. See also risk. [Note: We will also use the commonly used synonym *quality risk* in this book.]

risk impact: The damage that will be caused if the risk becomes an actual outcome or event.

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 likelihood: The estimated probability that a risk will become an actual outcome or event.

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

If such a technique has been selected, as a technical test analyst, you will need to understand risk-based testing and the selected technique for implementing it very well because you will play a key role in it. In this chapter, you'll learn how to perform risk analysis, to allocate test effort based on risk, and to sequence tests according to risk. These are the key tasks for a technical test analyst doing risk-based testing.

Risk-based testing includes three primary activities:

- Risk identification; i.e., figuring out what the different project and quality risks are for the project
- Assessing the level of risk—typically based on likelihood and impact—for each identified risk item
- Risk mitigation via test execution and other activities

In some sense, these activities are sequential, at least in when they start. They are staged such that risk identification starts first. Risk assessment comes next. Risk mitigation starts once risk assessment has determined the level of risk. However, since risk management should be continuous in a project, the reality is that risk identification, risk assessment, and risk mitigation are all recurring activities. In agile projects, risk identification, risk assessment, and risk mitigation occur both during release planning and during iteration planning.

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 ways to mitigate risks. So, risk management should include all project stakeholders.

In many cases, though, 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 might 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. Or, business analysts on IT projects can sometimes represent the users rather than involve them in potentially distressing risk analysis sessions with conversations about what could go wrong and how bad it would be.

Technical test analysts bring particular expertise to risk management due to their defect-focused outlook, especially with respect to their knowledge of the technical risks that are inherent in the project, such as risks related to security, system reliability, and performance. So, they should participate whenever possible. In fact, in many cases, the test manager will lead the quality risk analysis effort, with technical test analysts providing key support in the process.

ISTQB Glossary

risk analysis: The process of assessing identified project or product risks to determine their level of risk, typically by estimating their impact and probability of occurrence (likelihood).

risk assessment: The process of identifying and subsequently analyzing the identified project or product risk to determine its level of risk, typically by assigning likelihood and impact ratings. See also product risk, project risk, risk, risk impact, risk level, risk likelihood.

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

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

risk mitigation: See risk control.

If you plan to take the Advanced Level Technical Test Analyst exam, remember that all material from the Foundation syllabus, including material related to risk-based testing in Chapter 5, section 4 of the Foundation syllabus, is examinable.

1.2 Risk Identification

Learning objectives

Only common learning objectives.

Risk is the possibility of a negative or undesirable outcome or event. A risk is any problem that may 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 of risk is a product or quality risk. When the primary effect of a potential problem is on product quality, such potential problems are called product risks. A synonym for *product risk*, which we use most frequently ourselves, 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

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

The second type of risk is a project or planning risk. When the primary effect of a potential problem is on project success, such potential problems are called a project risk. (By project success, we mean complete achievement of all objectives for the project, not simply delivering software at the end.) 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.

Of course, you can consider a quality risk as a special type of project risk. While the ISTQB definition of project risk is given earlier, we like the informal definition of a project risk as *anything that might prevent the project from delivering the right product, on time and on budget*. However, the difference is that you can run a test against the system or software to determine whether a quality risk has become an actual outcome. You can test for system crashes, for example. Other project risks are usually not testable. You can't test for a staffing shortage.

For proper risk-based testing, we need to identify both product and project risks. We can identify both kinds of risks using the following techniques:

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

These techniques can be combined; for example, you can have product experts involved in risk brainstorming sessions as part of a risk workshop.

The test manager will usually organize the specific risk identification activities relevant for the project. As a technical test analyst with unique technical skills, you will be particularly well suited for conducting expert interviews, brainstorming with co-workers, and analyzing current and past experiences to

determine where the likely areas of product risk lie. In particular, technical test analysts work closely with their technical peers (e.g., developers, architects, operations engineers) to determine the areas of technical risk. You should focus on areas like performance risks, security risks, reliability risks, and other risks relating to the software quality characteristics covered in Chapter 4.

Conceivably, you can use a single integrated process to identify both project and product risks. Rex usually separates them into two separate processes because they have two separate deliverables. He includes the project risk identification process in the test planning process, and you should expect to assist the test manager in the test planning project, as discussed in Chapter 4. In parallel, the initial quality risk identification process occurs early in the project, and in agile projects, they occur during iteration planning.

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, if 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 by-products, which you should plan to capture and escalate to the proper person.

In the introduction, we 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 you'll miss.

How far should you take this process? Well, it depends on the technique. In lightweight techniques, which we frequently use, such as Pragmatic Risk Analysis and Management, risk identification stops at the risk items. Each risk item must be specific enough to allow for identification and assessment 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 as 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 sometimes used on safety-critical and embedded systems.

Other formal techniques look upstream to identify the source of the risk. Hazard analysis is an example of such a formal risk management technique. We've never used this technique ourselves, but Rex has talked to clients who have used it for safety-critical medical systems.¹

1.3 Risk Assessment

Learning objectives

TTA-1.3.1 (K2) Summarize the generic risk factors that the technical test analyst typically needs to consider.

The next step in the risk management process is risk assessment. 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.

In categorization of risk items, we can use ISO 9126 or other quality categories to organize the risk items. In our opinion—and in the Pragmatic Risk Analysis and Management process described here—it doesn't matter so much what category a risk item goes into, usually, as long as we don't forget it. However, in complex projects and for large organizations, the category of risk can determine who is responsible for managing the risk. A practical implication of categorization such as work ownership will make the categorization important.

The other part of risk assessment is determining the level of risk. There are a number of ways to classify the level of risk. The simplest is to look at two factors:

- The likelihood of the problem occurring; i.e., being present in the product when it is delivered for testing.
- The impact of the problem should it occur; i.e., being present in the product when it is delivered to customers or users after testing.

1. For more information on risk identification and analysis techniques, you can see either *Managing the Testing Process, 3rd Edition* (for both formal and informal techniques, including Pragmatic Risk Analysis and Management) or *Pragmatic Software Testing* (which focuses on Pragmatic Risk Analysis and Management), both by Rex Black. If you want to use Failure Mode and Effect Analysis, then we recommend reading D.H. Stamatis's *Failure Mode and Effect Analysis* for a thorough discussion of the technique, followed by *Managing the Testing Process, 3rd Edition* for a discussion of how the technique applies to software testing.

Note the distinction made here in terms of project timeline. Likelihood is the likelihood of a problem existing in the software during testing, not the likelihood of the problem being encountered by the user after testing. The likelihood of a user encountering the problem influences impact. Likelihood arises from technical considerations, typically, while impact arises from business considerations. As such, technical test analysts usually focus on supporting the evaluation of likelihood, while test analysts focus on evaluation of impact.

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

- Complexity of technology, architecture, code, and database(s)
- Personnel and training issues, especially those that relate to absence of sufficient skills to solve the technical challenges of implementation
- Intra-team and inter-team conflict, especially in terms of disagreement about technical and nonfunctional requirements
- Supplier and vendor contractual problems
- Geographical distribution of the development organization, as with outsourcing, or other complex team structures (such as those associated with large systems-of-systems development projects), particularly when unresolved communication barriers or disagreements about processes and product priorities exist
- Legacy or established designs and technologies that must integrate with 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 life cycle, resulting in a large percentage of the defects being detected in higher levels of testing, such as system test, system integration test, and acceptance test
- High rates of requirements, design, and code changes in the project without adequate structures or processes in place to manage those changes
- High defect introduction rates, including large rates of regression associated with bug fixes during test execution
- Complex interfacing and integration issues

While impact, being based on business factors, is typically the focus of the test analyst, here are some of the factors that the test analyst should consider:

- The frequency of use 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

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 monetary quantity. If we multiple 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 someday in the future of software engineering to be able to 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, 75, 50, 25, or 10 percent.

This is not to say—by any means—that a qualitative approach should be seen as 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 Rex has found is that if he accepts the limits of his data and applies appropriate lightweight quality risk management approaches, the results are not only perfectly useful, but also essential to a well-managed test process.

Unless your risk analysis is based on extensive and statistically valid risk data, it 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 we 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 differ-

ent 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 a guide for risk mitigation activities—including testing.

1.4 Risk Mitigation or Risk Control

Learning objectives

Only common learning objectives.

Having identified and assessed risks, we now must control them. The Advanced Technical Test Analyst syllabus refers to this as *risk mitigation*, which is accurate if we're referring to using testing to deal with quality (or product) risks. However, for risks in general, including project risks, the better term is *risk control*. We have four main options for risk control:

- Mitigation, where we take preventive measures to reduce the likelihood of the risk occurring and/or the impact of a risk should it occur. Testing prior to release is a form of mitigation for quality risks.
- Contingency, where we have a plan or perhaps multiple plans to reduce the impact of a risk should it occur. Having a technical support team in place is a form of contingency for quality risks.
- Transference, where we get another party to accept the consequences of a risk should it occur. Delivering a potentially buggy product effectively transfers quality risks onto users.
- Finally, ignoring or accepting the risk and its consequences should it occur. For trivial or highly unlikely risks, this is the most common option, sometimes unconsciously taken.

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 associated risks.

Analytical risk-based testing is focused on creating quality risk mitigation opportunities for the test team, including for technical test analysts. Risk-based testing mitigates quality risks via testing throughout the entire life cycle.

Testing can mitigate quality risk in various ways, but there are two very common ones:

- During all test activities, test managers, technical test analysts, and test analysts allocate effort for each quality risk item proportionally to the level of risk. Technical test analysts and test analysts select test techniques in a way that matches the rigor and extensiveness of the technique with the level of risk. Test managers, technical test analysts, and test analysts carry out test activities in 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, technical test analysts, and test analysts work with the project team to ensure that the repair of defects is appropriate to the level of risk.
- Test managers, technical test analysts, and test analysts report test results and project status in terms of residual risks. For example, which tests have not yet been run or have been skipped? Which tests have been run? Which have passed? Which have failed? Which defects have not yet been 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 happen only once in a project. Quality risk mitigation should occur throughout the life cycle. 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 performing other test control activities.

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 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.

As a technical test analyst, you might need to assist the test manager with identifying, assessing, and suggesting control options for test-related project risks as part of test planning. As technical test analysts, we're particularly concerned with test-affecting project risks like the following:

- Test environment and tools readiness
- Test staff availability and qualification
- Low quality of work products delivered to testing
- Overly high rates of change for work products delivered to testing
- Lack of standards, rules, and techniques for the testing effort

While it's usually the test manager's job to make sure these risks are controlled, the lack of adequate controls in these areas will affect the technical test analyst.

One idea discussed in the Foundation syllabus, a basic principle of testing, is the principle of early testing and QA. This principle stresses the preventive potential of testing. Preventive testing is part of analytical risk-based testing. We should try to 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, 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 take quality risk mitigation actions throughout the life cycle. Technical test analysts should look for opportunities to control risk using various techniques:

- Choosing an appropriate test design technique
- Reviews and inspections
- 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 and regression testing

Preventive test strategies acknowledge that quality risks can and should be mitigated by a broad range of activities. Many of these activities are static tests rather than dynamic tests. For example, if the requirements are not well written, perhaps we should institute reviews to improve their quality rather than relying on tests that we will run once the badly written requirements become a bad design and ultimately bad, buggy code.

Dynamic testing is not effective against all kinds of quality risks. In some cases, you can estimate the risk reduction effectiveness of testing in general and for specific test techniques for given risk items. There's not much point in using dynamic testing to reduce risk where there is a low level of test effectiveness. For example, code maintainability issues related to poor commenting or use of unstructured programming techniques will not tend to show up—at least, not initially—during dynamic testing.

Once we get to test execution, we run tests 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. Of course, running a test only demonstrates operation under certain conditions and does not constitute a proof of correctness under all possible conditions.

We 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.² Of course, in doing risk-based testing, if we only test based on our risk analysis, this can leave blind spots, so we need to use testing outside the predefined test procedures to see if we have missed anything, such as using experience-based and defect-based techniques.

2. You can find examples of how to carry out risk-based test reporting in Rex Black's book *Critical Testing Processes* and in the companion volume to this book, *Advanced Software Testing – Volume 2*, which addresses test management. A case study of risk-based test reporting at Sony is described in "Advanced Risk Based Test Results Reporting: Putting Residual Quality Risk Measurement in Motion," found at <http://www.rbcstesting.com/images/documents/STQA-Magazine-1210.pdf>.

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. Notice that, in general, those tests not yet run are less important than those tests already run. If we do curtail further testing, that property of risk-based test execution serves to transfer the remaining risk onto the users, customers, help desk and technical support personnel, or operational 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 for to decide whether to adjust our risk analysis? We can start with the following main factors:

- Totally new or very much changed quality 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

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

1.5 An Example of Risk Identification and Assessment Results

In Figure 1–1, you see an example of a quality risk analysis document. It is a case study from an actual project. This document—and the approach we used—followed the Pragmatic Risk Analysis and Management approach.

In this table, you can see some examples of quality risk items in three non-functional categories: performance, reliability/robustness, and resource utilization. These quality risk items were identified and assessed for a mainframe operating system management tool called Sysview, a mission-critical data center operations utility.³

3. You can find a detailed description of the use of risk-based testing on a project on the RBCS website in the article “A Case Study in Successful Risk-based Testing at CA,” found at <http://www.rbcus.com/images/documents/A-Case-Study-in-Risk-Based-Testing.pdf>.

Row ID Number	A	B	C	D			E	F	G	H	I	J	K	L
				Likelihood	Impact	RPN								
49	2.000	Performance	Failures to perform as required under expected loads											
50	2.001		Performance degrades under reasonable customer load	4	2	8	Ext					Regression		
51	2.002		MQ filter results in performance degradation	3	2	6	Ext					RN-pg55		
52														
54														
55	3.000	Reliability/Robustness	Failures to meet reasonable expectations of availability and failures to handle foreseen and unforeseen illegal events, inputs, etc.											
56	3.001		Product abend causes lost data	4	2	8	Ext					Regression		
57	3.002		Product abend causes system hang	4	1	4	Ext					Regression		
58	3.003		GMI does not recover from error conditions	3	2	6	Ext					DDS Vantage 6.0		
59														
60														
61														
62	4.000	Resource Utilization	Problems related to the excessive consumption of resources by Sysview											
63	4.001		Excessive System CPU consumption consumes too much resource	4	2	8	Ext					Regression		
64	4.002		64 bit check fails backwards compatibility with resource allocation	4	3	12	Ext					PRD 2.1.7		
65	4.003		CAPTURE consumes large amounts of CPU resources	4	3	12	Ext					Regression		
66	4.004		CICS consumes large amounts of CPU resources	4	3	12	Ext					Regression		
67	4.005		AUDIT consumes large amounts of CPU resources	3	3	9	Ext					Regression		
68	4.006		MVS consumes large amounts of CPU resources	4	3	12	Ext					Regression		
69	4.007		MQ consumes large amounts of CPU resources	4	3	12	Ext					Regression		
70	4.008		IMS consumes large amounts of CPU resources	4	3	12	Ext					Regression		
71	4.009		DataComm consumes large amounts of CPU resources	4	3	12	Ext					Regression		

Figure 1–1 An example of a quality risk analysis document using Pragmatic Risk Analysis and Management, focusing on nonfunctional risk areas

Example: Plan to Execution, Risk Based

We've mentioned that, properly done, risk-based testing manages risk throughout the life cycle. Let's look at how that happens, based on our usual approach to a test project.

During test planning, risk management comes first. We perform a quality risk analysis early in the project, ideally once the first draft of a requirements specification is available. From that quality risk analysis, we build an estimate for negotiation with and, we hope, approval by project stakeholders and management.

Once the project stakeholders and management agree on the estimate, we create a plan for the testing. The plan assigns testing effort and sequences tests based on the level of quality risk. It also plans for project risks that could affect testing.

During test control, we will periodically adjust the risk analysis throughout the project. That can lead to adding, increasing, or decreasing test coverage; removing, delaying, or accelerating the priority of tests; and other such activities.

During test analysis and design, we work with the test team to allocate test development and execution effort based on risk. Because we want to report test results in terms of risk, we maintain traceability to the quality risks.

During implementation and execution, we sequence the procedures based on the level of risk. We ensure that the test team, including the test analysts and technical test analysts, uses exploratory testing and other reactive techniques to detect unanticipated high-risk areas.

During the evaluation of exit criteria and reporting, we work with our test team to measure test coverage against risk. When reporting test results (and thus release readiness), we talk not only in terms of test cases run and bugs found but also in terms of residual risk.

1.6 Risk-Aware Testing Standard

An interesting example of how risk management, including quality risk management, plays into the engineering of complex and/or safety-critical systems is found in the ISO/IEC standard 61508. This standard applies to 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 is very much focused on risks. Risk analysis is required. It considers two primary factors as determining the level of risk, likelihood, and impact. During a project, we are 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. Thus, we must take defect-preventing 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 and verification report, and software functional safety report.

The standard addresses the author-bias problem. As discussed in the Foundation syllabus, this is the problem with self-testing: the fact that you bring the same blind spots and bad assumptions to testing your own work that you brought to creating that work. Therefore, the standard calls for tester independence, indeed insisting on it for those performing any safety-related tests. And since 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 (or 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 we'll cover in this book and in the companion volume for test analysis (*Advanced Software Testing – Volume 1*) that address various functional and black-box testing design techniques. Many of the techniques are ones that are described in subsequent chapters, including dynamic analysis, data recording and analysis, performance testing, interface testing, static analysis, and complexity metrics. Additionally, since thorough coverage, including during regression testing, is important in reducing the likelihood of missed bugs, the standard mandates the use of applicable automated test tools, which we'll also cover here in this book.

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 of testing is required, the standard states that it should be documented and independently verified. In other words, the standard can require auditing or outside reviews of testing activities. In addition, the standard also requires reviews for test cases, test procedures, and test results along with verification of data integrity under test conditions.

The standard requires the use of structural test design techniques. Structural coverage requirements are 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.

1.7 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 Technical Test Analyst exam. The questions in this section illustrate what is called a scenario question.

Scenario

Assume you are testing a computer-controlled braking system for an automobile. This system includes the possibility of remote activation to initiate a gradual braking followed by disabling the motor upon a full stop if the owner or the police report that the automobile is stolen or otherwise being illegally operated. Project documents and the product marketing collateral refer to this feature as *emergency vehicle control override*. The marketing team is heavily promoting this feature in advertisements as a major innovation for an automobile at this price.

Consider the following two statements:

- I. Testing will cover the possibility of the failure of the emergency vehicle control override feature to engage properly and also the possibility of the emergency vehicle control override engaging without proper authorization.
 - II. The reliability tests will include sending a large volume of invalid commands to the emergency vehicle control override system. Ten percent of these invalid commands will consist of deliberately engineered invalid commands that cover all invalid equivalence partitions and/or boundary values that apply to command fields; ten percent will consist of deliberately engineered invalid commands that cover all pairs of command sequences, both valid and invalid; the remaining invalid commands will be random corruptions of valid commands rendered invalid by the failure to match the checksum.
1. If the project is following a risk-based testing strategy, which of the following is a quality risk item that would result in the kind of testing specified in statements I and II above?
 - A. The emergency vehicle control override system fails to accept valid commands.
 - B. The emergency vehicle control override system is too difficult to install.

- C. The emergency vehicle control override system accepts invalid commands.
 - D. The emergency vehicle control override system alerts unauthorized drivers.
2. Assume that each quality risk item is assessed for likelihood and impact to determine the extent of testing to be performed. Consider only the information in the scenario, in question 1, and in your answer to that question. Which of the following statements is supported by this information?
- A. The likelihood and impact are both high.
 - B. The likelihood is high.
 - C. The impact is high.
 - D. No conclusion can be reached about likelihood or impact.

2 Structure-Based Testing

During the development of any non-trivial program, software structure is almost always created that cannot be determined from top-level software specifications.

—Michael Dale Herring

The second chapter of the Advanced Level Syllabus – Technical Test Analyst is concerned with structure-based (also known as white-box or code-based) test techniques. This syllabus contains eight sections:

1. Introduction
2. Condition Testing
3. Decision Condition Testing
4. Modified Condition/Decision Testing (MC/DC)
5. Multiple Condition Testing
6. Path Testing
7. API Testing
8. Selecting a Structure-Based Technique

In addition, we will cover several other topics that we feel are either required to understand these ISTQB mandated topics or extend into closely related topics that we feel a well-prepared technical test analyst should comprehend. Several of these were covered in the Foundation syllabus, but we feel it is important to go into a little more depth on these topics to help you better understand the material in this Advanced syllabus. These additional topics are as follows:

- Control Flow Testing Theory
- Statement Coverage Testing
- Decision Coverage Testing
- Loop Coverage Testing

2.1 Introduction

Learning objectives

Recall of content only.

Structure-based testing (also known as white-box testing) was briefly discussed in the Foundation level syllabus. Clearly, this is a topic that we must dig into more deeply in this book.

Structure-based testing uses the internal structure of the system as a test basis for deriving dynamic test cases. In other words, we are going to use information about how the system is designed and built to derive our tests.

The question that should come to mind is why. We have all kinds of specification-based (black-box) testing methods to choose from. Why do we need more? We don't have time or resources to spare for extra testing, do we?

Well, consider a world-class, outstanding system test team using all black-box and experience-based techniques. Suppose they go through all of their testing, using decision tables, state-based tests, boundary analysis, and equivalence classes. They do exploratory and attack-based testing and error guessing and use checklist-based methods. After all that, have they done enough testing? Perhaps for some. But research has shown, that even with all of that testing, and all of that effort, they may have missed a few things.

There is a really good possibility that as much as 70 percent of all of the code that makes up the system might never have been executed once! Not once!

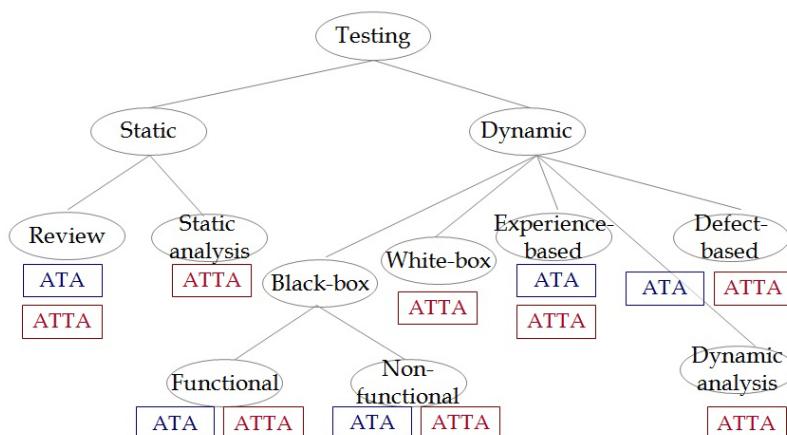


Figure 2–1 Advanced syllabus testing techniques

ISTQB Glossary

structure-based 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.

How can that be? Well, a good system is going to have a lot of code that is only there to handle the unusual, exceptional conditions that may occur. The happy path is often fairly straightforward to build—and test. And, if every user were an expert, and no one ever made mistakes, and everyone followed the happy path without deviation, we would not need to worry so much about testing the rest. If systems did not sometimes go down, and networks sometimes fail, and databases get busy and stuff didn't happen ...

But, unfortunately, many people are novices at using software, and even experts forget things. And people do make mistakes and multi-strike the keys and look away at the wrong time. And virtually no one follows only the happy path without stepping off it occasionally. Stuff happens. And, the software must be written so that when weird stuff happens, it does not roll over and die.

To handle these less likely conditions, developers design systems and write code to survive the bad stuff. That makes most systems convoluted and complex. Levels of complexity are placed on top of levels of complexity; the resulting system is usually hard to test well. We have to be able to look inside so we can test all of that complexity.

In addition, black-box testing is predicated on having models that expose the behaviors and list all requirements. Unfortunately, no matter how complete, not all behaviors and requirements are going to be visible to the testers. Requirements are often changed on the fly, features added, changed, or removed. Functionality often requires the developers to build “helper” functionality to be able to deliver features. Internal data flows that have asynchronous timing triggers often occur between hidden devices, invisible to black-box testers. Finally, malicious code—since it was probably not planned for in the specifications—will not be detected using black-box techniques. We must look under the covers to find it.

Much of white-box testing is involved with coverage, making sure we have tested everything we need to based on the context of project needs. Using white-box testing on top of black-box testing allows us to measure the coverage we got and add more testing when needed to make sure we have tested all of the

important complexity we should. In this chapter, we are going to discuss how to design, create, and execute white-box testing.

2.1.1 Control Flow Testing Theory

Our first step into structural testing will be to discuss a technique called control flow testing. Control flow testing is done through control flow graphs, which provide a way to abstract a code module to better understand what it does. Control flow graphs give us a visual representation of the structure of the code. The algorithm for all control flow testing consists of converting a section of the code into a control graph and then analyzing the possible paths through the graph. There are a variety of techniques that we can apply to decide just how thoroughly we want to test the code. Then we can create test cases to test to that chosen level.

If there are different levels of control flow testing we can choose, we need to come up with a differentiator that helps us decide which level of coverage to choose. How much should we test? Possible answers range from no testing at all (a complete laissez-faire approach) to total exhaustive testing, hitting every possible path through the software. The end points are actually a little silly; no one is going to (intentionally) build a system and send it out without running it at least once. At the other end of the spectrum, exhaustive testing would require a near infinite amount of time and resources. In the middle, however, there is a wide range of coverage that is possible.

We will look at different reasons for testing to some of these different levels of coverage later in this chapter. In this section, we just want to discuss what these levels of control flow coverage are named.

At the low end of control flow testing, we have *statement* coverage. Common industry synonyms for *statement* are *instruction* and *code*. They all mean the same thing: have we exercised, at one time or another, every single line of executable code in the system? That would give us 100 percent statement coverage. It is possible to test less than that; people not doing white-box testing do it all the time, they just don't measure it. Remember, thorough black-box testing without doing any white-box testing may total less than 30 percent statement coverage, a number often discussed at conferences urging more white-box testing.

To calculate the total statement coverage actually achieved, divide the total number of executable statements into the number of statements that have actually been exercised by your testing. For example, if there are 1,000 total exe-

ISTQB Glossary

control flow testing: An approach to structure-based testing in which test cases are designed to execute specific sequences of events. Various techniques exist for control flow testing, decision testing, condition testing, and path testing, that each have their specific approach and level of control flow coverage.

cutable statements and you have executed 500 of them in your testing, you have achieved 50 percent statement coverage. This same calculation may be done for each different level of coverage we discuss in this chapter.

The next step up in control flow testing is called *decision* (or *branch*) coverage. This is determined by the total number of all of the outcomes of all of the decisions in the code that we have exercised. We will honor the ISTQB decision to treat *branch* testing and *decision* testing as synonyms. There are very slight differences between decision testing and branch testing, but those differences are insignificant at the level at which we will examine them.¹

Then we have *condition* coverage, where we ensure that we evaluate each atomic condition that makes up a decision at least once, and *multiple condition* coverage, where we test all possible combinations of outcomes for individual atomic conditions inside all decisions.

We will add a level of coverage called *loop* coverage, not discussed by ISTQB but we think interesting anyway.

If this sounds complex, well, it is a bit. In the upcoming pages, we will explain what all of these terms and all the concepts mean. It is not as confusing as it might be—we just need to take it one step at a time and all will be clear. Each successive technique essentially adds more coverage than could be achieved by the previous technique.

2.1.2 Building Control Flow Graphs

Before we can discuss control flow testing, we must define how to create a control flow graph. In the next few paragraphs, we will discuss the individual pieces that make up control flow graphs.

1. The United States Federal Aviation Administration makes a distinction between branch coverage and decision coverage with branch coverage deemed weaker. If you are interested in this distinction, see the FAA report *Software Verification Tools Assessment Study* (June 2007) at www.faa.gov

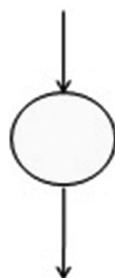


Figure 2–2 The process block

In Figure 2–2, we see the process block. Graphically, it consists of a node (bubble or circle) with one path leading to it and one path leading from it. Essentially, this represents a chunk of code that executes sequentially: that is, no decisions are made inside of it. The flow of execution reaches the process block, executes through that block of code in exactly the same way each time, and then exits, going elsewhere.

This concept is essential to understanding control flow testing. Decisions are the most significant part of the control flow concept; individual lines of code where no decisions are made do not affect the control flow and thus can be effectively ignored. The process block has *no* decisions made inside it. Whether the process block has one line or a million lines of code, we only need one test to execute it completely. The first line of code executes, the second executes, the third ... right up to the millionth line of code. Excepting the possibility of an error (e.g., differing data may cause a divide by zero), there is no deviation no matter how many different test cases are run. Entry is at the top, exit is at the bottom, and every line of code executes every time.

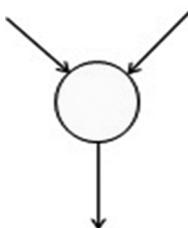


Figure 2–3 The junction point

The second structure we need to discuss is called a junction point, seen in Figure 2–3. This structure may have any number of different paths leading into the process block, with only one path leading out. No matter how many different paths we have throughout a module, eventually they must converge—or exit the module. Again, no decisions are made in this block; all roads lead to it with only one road out.



Figure 2–4 Two decision points

A decision point is very important; indeed, it's key to the concept of control flow. A decision point is represented as a node with one input and two or more possible outputs. Its name describes the action inside the node: a decision as to which way to exit is made and control flow continues out that path while ignoring all of the other possible choices. In Figure 2–4, we see two decision points: one with two outputs, one with five outputs.

How is the choice of output path made? Each programming language has a number of different ways of making decisions. In each case, a logical decision, based on comparing specific values, is made. We will discuss these later; for now, it is sufficient to say a decision is made and control flow continues one way and not others.

Note that these decision points force us to have multiple tests, at least one test for each way to make the decision differently, changing the way we traverse the code. In a very real way, it is the decisions that a computer can make that make it interesting, useful, and complex to test.

The next step is to combine these three relatively simple structures into useful control flow graphs.

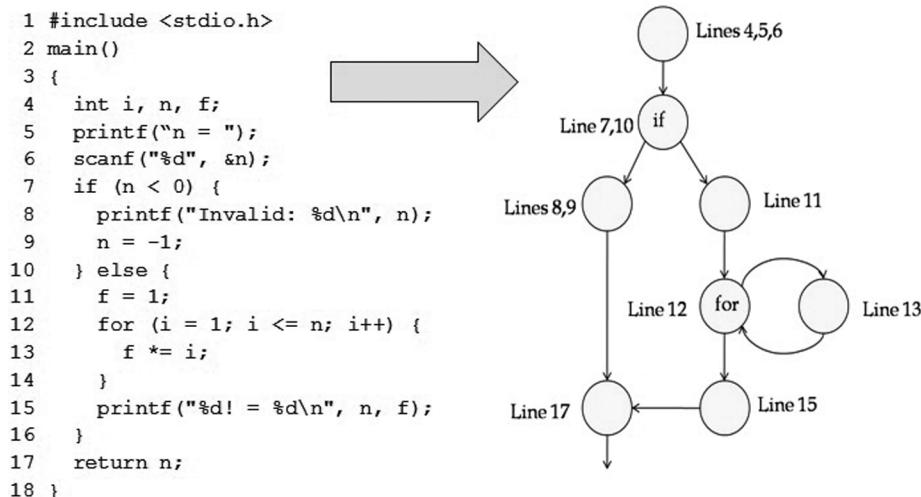


Figure 2–5 A simple control flow graph

In Figure 2–5, we have a small module of code. Written in C, this module consists of a routine to calculate a factorial. Just to refresh your memory, a factorial is the product of all positive integers less than or equal to n and is designated mathematically as $n!$. $0!$ is a special case that is explicitly defined to be equal to 1. The following are the first three factorials:

$$1! = 1$$

$$2! = 1 * 2 ==> 2$$

$$3! = 1 * 2 * 3 ==> 6 \quad \text{etc.}$$

To create a control flow diagram from this code, we do the following:

1. The top process block contains up to and including line 6. Note there are no decisions, so all lines go into the same process block. By combining multiple lines of code where there is no decision made into a single process block, we simplify the graph. Note that we could conceivably have drawn a separate process block for each line of code.
2. At line 7 there is a reserved word in the C language: *if*. This denotes that the code is going to make a decision. Note that the decision can go one of two ways, but not both. If the decision resolves to TRUE, the left branch is taken and lines 8 and 9 are executed and then the thread jumps to line 17.
3. On the other hand, if the decision at line 7 resolves to FALSE, the thread jumps to line 10 to execute the *else* clause.

4. Line 11 sets a value and then falls through to line 12 where another decision is made. This line has the reserved word *for*, which means we may or may not loop.
5. At line 12, a decision is made in the *for* loop body using the second phrase in the statement: $(i <= n)$. If this evaluates to TRUE, the loop will fire, causing the code to go to line 13, where it calculates the value of *f* and then goes right back to the loop at line 12.
6. If the *for* loop evaluates to FALSE, the thread falls through to line 15 and thence to line 17.
7. Once the thread gets to line 17, the function ends at line 18.

To review the control flow graph: there are six process blocks, two decision blocks (lines 7 and 12), and two junction points (lines 12 and 17).

2.1.3 Statement Coverage

Now that we have discussed control flow graphing, let's take a look at the first level of coverage we mentioned earlier, statement coverage (also called instruction or code coverage). The concept is relatively easy to understand: executable statements are the basis for test design selection. To achieve statement coverage, we pick test data that forces the thread of execution to go through each line of code that the system contains.

The first edition of this book was challenged by some who did not believe that statement coverage is a control flow design technique. We have decided that for logical consistency, and for the understanding of those who are just learning this subject, we will discuss statement coverage under control flow testing whether or not it strictly belongs here.

To calculate the current level of statement coverage that we have attained, divide the number of code statements that have been executed during the testing by the number of code statements in the entire module/system; if the quotient is equal to 1, we have 100 percent statement coverage.

The bug hypothesis is pretty much as expected; bugs can lurk in code that has not been executed.

Statement-level coverage is considered the least effective of all control flow techniques. In order to reach this modest level of coverage, a tester must come up with enough test cases to force every line to execute at least once. While this does not sound too onerous, it must be done purposefully. One good practice for an advanced technical test analyst is to make sure the developers they work

ISTQB Glossary

statement testing: A white-box test design technique in which test cases are designed to execute statements.

with are familiar with the concepts we are discussing here. Achieving (at least) statement coverage can (and should) be done while unit testing.

There is certainly no guarantee that even college-educated developers learned how important this coverage level is. Jamie had an undergraduate computer science major and earned a master's degree in computer science from reputable colleges and yet was never exposed to any of this information until after graduation, when he started reading test books and attending conferences. Rex can attest that the concepts of white-box coverage—indeed, test coverage in general—were not discussed when he got his degree in computer science and engineering at UCLA.

The Institute of Electrical and Electronics Engineers (IEEE), in the unit test standard ANSI 87B (1987), stated that statement coverage was the minimum level of coverage that should be acceptable. Boris Beizer, in his seminal work, *Software Testing Techniques* (ITC Press 1990), has a slightly more inflammatory take on it: "... testing less than this for new software is unconscionable and should be criminalized."

Also from that book, Beizer lays out some rules of common sense:

1. *Not testing a piece of code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs.*
2. *The high-probability paths are always thoroughly tested if only to demonstrate that the system works properly. If you have to leave some code untested at the unit level, it is more rational to leave the normal, high-probability paths untested, because someone else is sure to exercise them during integration testing or system testing.*
3. *Logic errors and fuzzy thinking are inversely proportional to the probability of the path's execution.*
4. *The subjective probability of executing a path as seen by the routine's designer and its objective execution probability are far apart. Only analysis can reveal the probability of a path, and most programmers' intuition with regard to path probabilities is miserable.*
5. *The subjective evaluation of the importance of a code segment as judged by its programmer is biased by aesthetic sense, ego, and familiarity. Elegant code*

might be heavily tested to demonstrate its elegance or to defend the concept, whereas straightforward code might be given cursory testing because “How could anything go wrong with that?”

Have we convinced you that this is important? Let's look at how to do it.

```

1 #include <stdio.h>
2 main()
3 {
4     int i, n, f;
5     printf("n = ");
6     scanf("%d", &n);
7     if (n < 0) {
8         printf("Invalid: %d\n", n);
9         n = -1;
10    } else {
11        f = 1;
12        for (i = 1; i <= n; i++) {
13            f *= i;
14        }
15        printf("%d! = %d\n", n, f);
16    }
17    return n;
18 }
```

Test value:
 $n < 0$
 $n > 0$

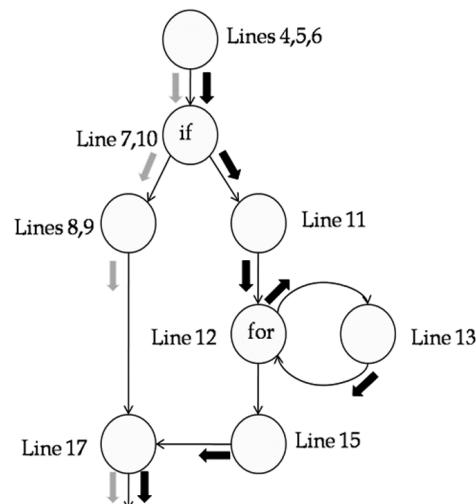


Figure 2–6 Statement coverage example

Looking at the same code and control flow graph we looked at earlier, Figure 2–6 shows the execution of our factorial function when an input value less than zero is tested. The gray arrows show the path taken through the code with this test. Note that we have covered some of the lines of code, but not all. Based on the graph, we still have a way to go to reach statement coverage.

Following the black arrows, we see the result of testing with a value greater than 0. At this point, the *if* in line 7 resolves to FALSE and so we go down the untested branch. Because the input value is at least 1, the loop will fire at least once (more times if the input is greater than 1). When *i* increments and becomes larger than *n* (due to the third phrase in the parentheses, *i++*), the loop will stop and the thread of execution will fall through to line 15, line 17, and out.

Note that you do not really need the control flow graph to calculate this coverage; you could simply look at the code and see the same thing. However, for many people, it is much easier to read the graph than the code.

At this point, we have achieved statement coverage. If you are unsure, go back and look at Figure 2–6 again. Every node (and line of code) has been covered by at least one arrow. We have run two test cases ($n < 0$, $n > 0$). At this point,

we might ask ourselves if we are done testing. If we were to do more testing than we need, it is wasteful of resources and time. While it might be wonderful to always test enough that there are no more bugs possible, it would (even if it were possible) make software so expensive that no one could afford it.

Tools are available to determine if your testing has achieved 100 percent statement coverage. Coverage tools were introduced in the ISTQB Foundation level syllabus.

The real question we should ask is, Did we test enough for the context of our project? And the answer, of course, is that it depends on the project. Doing less testing than necessary will increase the risk of really bad things happening to the system in production. Every test group walks a balance beam in making the “do we need more testing?” decision.

Maybe we need to see where more coverage might come from and why we might need it. Possibly we can do a little more testing and get a lot better coverage.

Figure 2–7 shows a really simple piece of code and a matching control flow graph. Test case 1 is represented by the gray arrows; the result of running this single test with the given values is full 100 percent statement coverage. Notice on line 2, we have an *if* statement that compares *a* to *b*. Since the inputted values ($a = 3, b = 2$), when plugged into this decision, will evaluate to TRUE, the line $z = 12$ will be executed, and the code will fall out of the *if* statement to line 4, where the final line of code will execute and set the variable *Rep* to 6 by dividing 72 by 12.

1. $z = 0;$
2. $\text{if } (a > b) \text{ then}$
3. $z = 12;$
4. $\text{Rep} = 72/z;$

Test 1: $a = 3, b = 2 \rightarrow \text{Rep} = 6$
 Test 2: $a = 2, b = 3 \rightarrow \text{Rep} = ?$

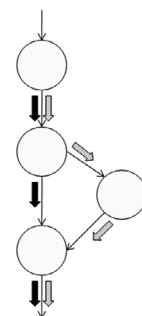


Figure 2–7 Where statement coverage fails

A few moments ago, we asked if statement coverage was enough testing. Here, we can see a potential problem with stopping at only statement coverage. Notice that if we pass in the values shown as test 2 ($a = 2, b = 3$), we have a different

decision made at the conditional on line 2. In this case, a is not greater than b (i.e., the decision resolves to FALSE) and line 3 is not executed. No big deal, right. We still fall out of the *if* to line 4. There, the calculation $72/z$ is performed, exactly the way we would expect. However, there is a nasty little surprise at this point. Since z was not reset to 12 inside the branch, it remained set to 0. Therefore, expanding the calculation performed on line 4, we have 72 divided by 0.

Oops!

You might remember from elementary school that anything divided by 0 is not defined. Seriously, in our mathematics, it is simply not allowed. So what should the computer do at this point? If there is an exception handler somewhere in the execution stack, it should fire, unwinding all of the calculations that were made since it was set. If there is no exception handler, the system may crash—hard! The processor that our system is running on likely has a “hard stop” built into its micro-code for this purpose. In reality, most operating systems are not going to let the CPU crash with a divide by zero failure—but the OS will likely make the offending application disappear like a bad dream.

But, you might say, we had statement coverage when we tested! This just isn’t fair! As we noted earlier, statement coverage by itself is a bare minimum coverage level that is almost guaranteed to miss defects. We need to look at another, higher level of coverage that can catch this particular kind of defect.

2.1.4 Decision Coverage

The next strongest level of structural coverage is called decision (or branch) coverage.

Rather than looking at individual statements, this level of coverage looks at the decisions themselves. Every decision has the possibility of being resolved as either TRUE or FALSE. No other possibilities. Binary results, TRUE or FALSE. For those who point out that the switch statement can make more than two decisions, well, conceptually that appears to be true. The switch statement is a complex set of decisions, often built as a table by the compiler. The generated code, however, is really just a number of binary compares that continue sequentially until either a match is found or the default condition is reached. Each atomic decision in the switch statement is still a comparison between two values that evaluates either TRUE or FALSE.

To get to the 100 percent decision level of coverage, every decision made by the code must be resolved both ways at one time or another, TRUE and FALSE. That means—at minimum—two test cases must be run, one with data that

causes the evaluation to resolve TRUE and a separate test case where the data causes the decision to resolve FALSE. If you omit one test or the other, then you do not achieve 100 percent decision coverage.

Our bug hypothesis was proved out in Figure 2–7. An untested branch may leave a land mine in the code that can cause a failure even though every line was executed at least once. The example we went over may seem too simplistic to be true, but this is exactly what often happens. The failure to set (or reset) a value in the conditional causes a problem later on in the code.

Note that if we execute each decision both ways, giving us decision coverage, it guarantees statement coverage in the same code. Therefore, decision coverage is said to be stronger than statement coverage. Statement coverage, as the weakest coverage, does not guarantee anything beyond each statement being executed at least once.

As before, we could calculate the exact level of decision coverage by dividing the number of decision outcomes tested by the total number of decision outcomes in the module/system. For this book, we are going to speak as if we always want to achieve full—that is 100 percent—decision coverage. In real life, your mileage might vary. There are tools available to measure decision coverage. However, we have found that some of the tools that claim to calculate decision coverage do not work correctly; you should verify the correctness of the tool you are using if it matters to your organization.

Having defined this higher level of coverage, let's dig into it.

The ability to take one rather than the other path in a computer is what really makes a computer powerful. Each computer program likely makes millions of decisions a minute. But exactly what is a decision?

As noted earlier, each predicate eventually must resolve to one of two values, TRUE or FALSE. As seen in our code, this might be a really simple expression: *if (a > b)* or *if (n < 0)*. However, we often need to consider more complex expressions in a decision. In fact, a decision can be arbitrarily complex, as long as it eventually resolves to either TRUE or FALSE. The following is a legal expression that resolves to a single Boolean value:

$(a > b) \parallel (x + y == -1) \&\& ((d) != \text{TRUE})$

This expression has three subexpressions in it, separated by the Boolean operators \parallel and $\&\&$. The first is an *OR* operator and the second is an *AND* operator. While the *AND* operator has a slightly higher precedence for evaluation than the *OR*, the rule for evaluation of Boolean operators in the same expression is to treat them as if they have equal precedence and to evaluate them according to

ISTQB Glossary

branch testing: A white-box test design technique in which test cases are designed to execute branches. ISTQB deems this identical to decision testing.

decision testing: A white-box test design technique in which test cases are designed to execute decision outcomes. ISTQB deems this identical to branch testing.

their associativity (i.e., left to right). Only parentheses would change the order of evaluation. Therefore, the first subexpression to be evaluated would be the leftmost, ($a > b$). This will evaluate to either TRUE or FALSE based on the actual runtime values. Next, the subexpression ($x + y == -1$) will evaluate to either TRUE or FALSE based on the runtime values. This result will be ORed to the first result and stored for a moment. Finally, the last subexpression, ($((d) != \text{TRUE})$), will evaluate to either TRUE or FALSE based on the value of d and will then be ANDed to the previous result.²

While this expression appears to be ugly, the compiler will evaluate the entire predicate—step-by-step—to inform us whether it is a legal expression or not. This actually points out something every developer should do: *write code that is understandable!* Frankly, we would not insult the term *understandable* by claiming this expression is!

There are a variety of different decisions that a programming language can make.

Table 2–1 C language decision constructs

Either/or Decisions	Loop Decisions
<pre>if (expr) {} else {}</pre>	<pre>while(expr) {}</pre>
<pre>switch (expr) { case const_1: {} break; case const_2: {} break; case const_3: {} break; case const_4: {} break; default {} }</pre>	<pre>do {} while (expr)</pre>
	<pre>for (expr_1; expr_2; expr_3) {}</pre>

2. This may not be strictly true if the compiler optimizes the code and/or short circuits it. We will discuss those topics later in the chapter. For this discussion, let's assume that it is true.

Here, in Table 2–1, you can see a short list of the decision constructs that the programming language C uses. Other languages (including C++, C#, and Java) use similar constructs. The commonality between all of these (and all of the other decision constructs in all imperative programming languages) is that each makes a decision that can go only two ways: TRUE or FALSE.

```

1 #include <stdio.h>
2 main()
3 {
4     int i, n, f;
5     printf("n = ");
6     scanf("%d", &n);
7     if (n < 0) {
8         printf("Invalid: %d\n", n);
9         n = -1;
10    } else {
11        f = 1;
12        for (i = 1; i <= n; i++) {
13            f *= i;
14        }
15        printf("%d! = %d\n", n, f);
16    }
17    return n;
18 }
```

Statement coverage: We tested
 $(n < 0)$ and $(n > 0)$
To get decision coverage,
we still need $(n == 0)$

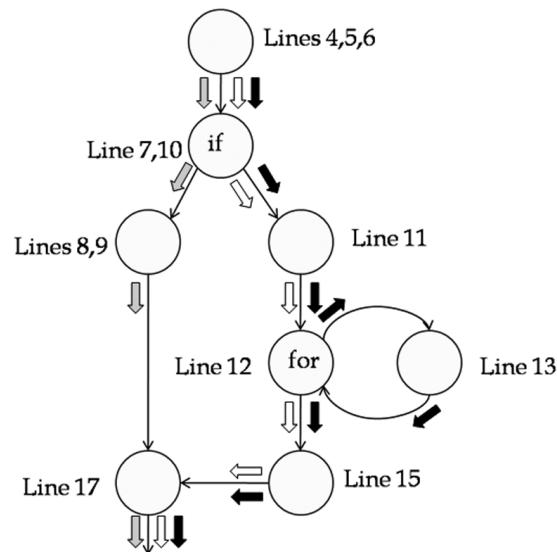


Figure 2–8 Decision coverage example

Looking back at our original example, with just two test cases we did not achieve decision coverage, even though we did attain statement coverage. We did not *not* execute the *for* loop during the test. Remember, a *for* loop evaluates an expression and decides whether to loop or not to loop based on the result.

In order to get decision coverage, we need to test with the value 0 inputted, as shown in Figure 2–8. When 0 is entered, the first decision evaluates to FALSE, so we take the *else* path. At line 12, the predicate (is 1 less than or equal to 0) evaluates to FALSE, so the loop is not taken. Recall that earlier, we had tested with a value greater than 0, which did cause the loop to execute. Now we have achieved decision coverage; it took three test cases.

At the risk of being pedantic, when the test was ($n > 0$), causing the loop to fire, it did eventually run down and not execute the *for* loop. When discussing a loop, there will always be a time that the loop ends and does fall out (or else we have an infinite loop, which brings other problems). That time after the final

loop does not count as not taking the loop. We need a test that never takes the loop to achieve decision coverage.

That brings us back to the same old question. Having tested to the decision level of coverage, have we done enough testing?

Consider the loop itself. We executed it zero times when we inputted 0. We executed it an indeterminate number of times when we inputted a value greater than zero. Is that sufficient testing?

Well, not surprisingly, there is another level of coverage called loop coverage. We will look at that next.

2.1.5 Loop Coverage

While not discussed in the ISTQB Advanced syllabus, loop testing should be considered an important control flow, structural test technique. If we want to completely test a loop, we would need to test it zero times (i.e., did not loop), one time, two times, three times, all the way up to n times where it hits the maximum it will ever loop. Bring your lunch; it might be an infinite wait!

Like other exhaustive testing techniques, full loop testing does not provide a reasonable amount of coverage. Different theories have been offered that try to prescribe how much testing we should give a loop. The basic minimum tends to be two tests, zero times through the loop and one time through the loop (giving us decision coverage). Others add a third test, multiple times through the loop, although they do not specify how many times. We prefer a stronger standard; we suggest that you try to test the loop zero and one time and then test the maximum number of times it is expected to cycle (if you know how many times that is likely to be). In a few moments, we will discuss Boris Bezier's standard, which is even more stringent.

We should be clear about loop testing. Some loops could execute an infinite number of times; each time through the loop creates one or more extra paths that could be tested. In the Foundation syllabus, a basic principle of testing was stated: "Exhaustive testing is impossible." Loops, and the possible infinite variations they can lead to, are the main reason exhaustive testing is impossible. Since we cannot test all of the ways loops will execute, we want to be pragmatic here, coming up with a level of coverage that gives us the most information in the least amount of tests. Our bug hypothesis is pretty clear; failing to test the loop (especially when it does not loop even once) may cause bugs to be shipped to production.

In the next few paragraphs, we will discuss how to get loop coverage in our factorial example. Here in Figure 2–9, following the gray arrows, we satisfy the first leg of loop coverage, zero iterations. Entering a zero, as we discussed earlier,

allows the loop to be skipped without causing it to loop. In line 12, the *for* loop condition evaluates as (*1 is less than or equal to 0*), or FALSE. This causes us to drop out of the loop without executing it.

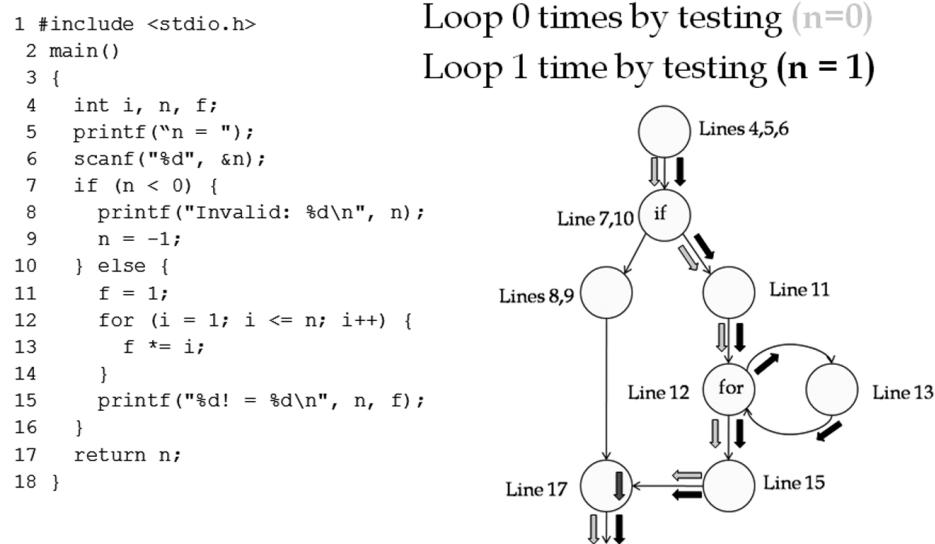


Figure 2–9 Loop coverage 0 and 1 times through

By entering a 1 (following the black arrows), we get the loop to execute just once. Upon entering line 12, the condition evaluates to (*1 is less than or equal to 1*), or TRUE. The loop executes and *i* is incremented. After the loop (line 13) is executed, the condition is evaluated again, (*2 is less than or equal to 1*), or FALSE. This causes us to drop out of the loop.

Zero and one time through the loop are relatively easy to come up with. How about the maximum times through? Each loop is likely to have a different way of figuring that out, and for some loops, it will be impossible to determine.

In this code, we have a monotonically increasing value which makes it easy to calculate the greatest number of loops possible. The maximum size of the data type used in the collector variable, *f*, will allow us to calculate the maximum number of iterations. We need to compare the size of the calculated factorial against the maximum integer size (the actual data type would have to be confirmed in the code).

In Table 2–2, we show a table with factorial values.

Table 2–2 Factorial values

n	n!
0	1
1	1
2	2
3	6
4	24
5	120
6	720
8	40,320
9	362,880
10	3,628,800
11	39,916,800
12	479,001,600
13	6,227,020,800
14	87,178,291,200
15	1,307,674,368,000

Assuming a signed 32-bit integer being used to hold the calculation, the maximum value that can be stored is 2,147,483,647. An input of 12 should give us a value of 479,001,600. An input value of 13 would cause an overflow (6,227,020,800). If the programmer used an unsigned 32-bit integer with a maximum size of 4,294,967,295, notice that the same number of iterations would be allowed; an input of 13 would still cause an overflow.

In Figure 2–10, we would go ahead and test the input of 12 and check the expected output of the function.

```

1 #include <stdio.h>
2 main()
3 {
4     int i, n, f;
5     printf("n = ");
6     scanf("%d", &n);
7     if (n < 0) {
8         printf("Invalid: %d\n", n);
9         n = -1;
10    } else {
11        f = 1;
12        for (i = 1; i <= n; i++) {
13            f *= i;
14        }
15        printf("f! = %d\n", n, f);
16    }
17    return n;
18 }
```

Loop 12 times by testing ($n==12$)
 Loop 13 times and it overflows

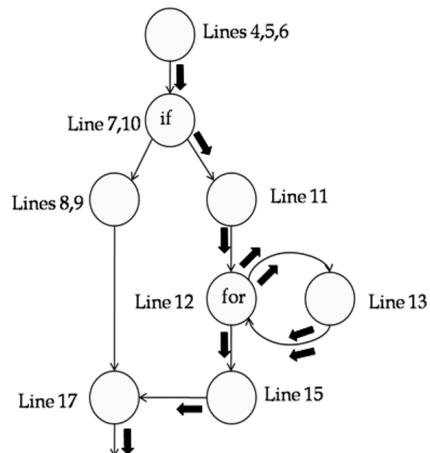


Figure 2-10 Loop coverage max times through

It should be noted that our rule for loop coverage does not deal comprehensively with negative testing. Remember that negative testing is checking invalid values to make sure the failure they cause is graceful, giving us a meaningful error message and allowing us to recover from the error. We probably want to test the value 13 to make sure the overflow is handled gracefully. This is consistent with the concept of boundary value testing discussed in the ISTQB Foundation level syllabus.

Boris Beizer, in his book *Software Testing Techniques*, had suggestions for how to test loops extensively. Note that he is essentially covering the three point boundary values of the loop variable with a few extra tests thrown in.

1. If possible, test a value that is one less than the expected minimum value the loop can take. For example, if we expect to loop with the control variable going from 0 to 100, try -1 and see what happens.
2. Try the minimum number of iterations—usually zero iterations. Occasionally, there will be a positive number as the minimum.
3. Try one more than the minimum number.
4. Try to loop once (this test case may be redundant and should be omitted if it is).
5. Try to loop twice (this might also be redundant).

6. A typical value. Beizer always believes in testing what he often calls a nominal value. Note that the number of loops, from one to max, is actually an equivalence set. The nominal value is often an extra test that we tend to leave out.
7. One less than the maximum value.
8. The maximum number of values.
9. One more than the maximum value.

The most important differentiation between Beizer's guidelines and our loop coverage described earlier is that he advocates negative testing of loops. Frankly, it is hard to argue against this thoroughness. Time and resources, of course, are always factors that we must consider when testing. We would argue that his guidelines are useful and should be considered when testing mission-critical or safety-critical software.

Finally, one of the banes of testing loops is when they are nested inside each other. We will end this topic with Beizer's advice for reducing the number of tests when dealing with nested loops.

1. Starting at the innermost loop, set all outer loops to minimum iteration setting.
2. Test the boundary values for the innermost loop as discussed earlier.
3. If you have done the outermost loop already, go to step 5.
4. Continue outward, one loop at a time until you have tested all loops.
5. Test the boundaries of all loops simultaneously. That is, set all to 0 loops, 1 loop, maximum loops, 1 more than maximum loops.

Beizer points out that practicality may not allow hitting each one of these values at the same time for step 5. As guidelines go, however, these will ensure pretty good testing of looping structures.

2.1.6 Hexadecimal Converter Exercise

In Figure 2–11, you'll find a C program, targeted for a UNIX platform, that accepts a string with hexadecimal characters (among other unwanted characters). It ignores the other characters and converts the hexadecimal characters to a numeric representation. If a Ctrl-C is inputted, the last digit that was converted is removed from the buffer. Please note that we removed the #includes to simplify the code; clearly those would need to be added to actually compile this for the UNIX platform.

If you test with the input strings “24ABd690BBCcc” and “ABC-def1234567890”, what level of coverage will you achieve?

What input strings could you add to achieve statement and branch coverage? Would those be sufficient for testing this program?

The answers are shown in the next section.

```

1.     jmp_buf sjbuf;
2.     unsigned long int hexnum;
3.     unsigned long int nhex;
4.
5.     main()
6.     /* Classify and count input chars */
7.     {
8.         int c, gotnum;
9.         void pophdigit();
10.
11.        hexnum = nhex = 0;
12.
13.        if (signal(SIGINT, SIG_IGN) != SIG_IGN) {
14.            signal(SIGINT, pophdigit);
15.            setjmp(sjbuf);
16.        }
17.        while ((c = getchar()) != EOF) {
18.            switch (c) {
19.                case '0':case '1':case '2':case '3':case '4':
20.                case '5':case '6':case '7':case '8':case '9':
21.                    /* Convert a decimal digit */
22.                    nhex++;
23.                    hexnum *= 0x10;
24.                    hexnum += (c - '0');
25.                    break;
26.                case 'a': case 'b': case 'c':
27.                case 'd': case 'e': case 'f':
28.                    /* Convert a lower case hex digit */
29.                    nhex++;
30.                    hexnum *= 0x10;
31.                    hexnum += (c - 'a' + 0xa);
32.                    break;
33.                case 'A': case 'B': case 'C':
34.                case 'D': case 'E': case 'F':
35.                    /* Convert an upper case hex digit */
36.                    nhex++;
37.                    hexnum *= 0x10;
38.                    hexnum += (c - 'A' + 0xA);
39.                    break;
40.                default:
41.                    /* Skip any non-hex characters */
42.                    break;
43.            }
44.        }
45.        if (nhex == 0) {
46.            fprintf(stderr,"hxconv: no hex digits to convert!\n");
47.        } else {
48.            printf("Got %d hex digits: %x\n", nhex, hexnum);
49.        }
50.
51.        return 0;
52.    }
53.    void pophdigit()
54.    /* Pop the last hex input out of hexnum if interrupted */
55.    signal(SIGINT, pophdigit);
56.    hexnum /= 0x10;
57.    nhex--;
58.    longjmp(sjbuf, 0);
59. }
```

Figure 2–11 Hexadecimal converter code

2.1.7 Hexadecimal Converter Exercise Debrief

The strings “24ABd690BBCcc” and “ABCdef1234567890” do not achieve any specified level of coverage that we have discussed in this book.

To get statement and decision coverage, you would need to add the following:

- At least one string containing a signal (Ctrl-C) to execute the signal() and pophdigit() code
- At least one string containing a non-hex digit
- At least one empty string to cause the *while* loop on line 17 to not loop

To get loop coverage, we would have to add this:

- One test that inputs the maximum number of hex digits that can be stored (based on the datatype of nhex)

There are also some interesting test cases based on placement that we would likely run. In at least one of these we would expect to find a bug (hint, hint):

- A string containing only a signal (Ctrl-C)
- A string containing a signal at the first position
- A string with more hex digits than can be stored
- A string with more signals than hex digits

2.2 Condition Coverage

Learning objectives

TTA-2.2.1 (K2) Understand how to achieve condition coverage and why it may be less rigorous testing than decision coverage.

So far we have looked at statement coverage (have we exercised every line of code?), decision coverage (have we exercised each decision both ways?), and loop coverage (have we exercised the loop enough?).

As Ron Popeil, the purveyor of many a fancy gadget in all-night infomercials used to say, “But wait! There’s more!”

While we have exercised the decision both ways, we have yet to discuss how the decision is made. Earlier, when we discussed decisions, you saw that they could be relatively simple, such as

$$A > B$$

or arbitrarily complex, such as

$$(a>b) \parallel (x+y==1) \&\& (d) != \text{TRUE}$$

ISTQB Glossary

condition testing: A white-box test design technique in which test cases are designed to execute condition outcomes.

If we input data to force the entire condition to TRUE, then we go one way; force it to FALSE and we go the other way. At this point, we have achieved decision coverage. But is that enough testing?

Could there be bugs hiding in the evaluation of the condition itself? The answer, of course, is a resounding yes! And, if we know there might be bugs there, we might want to test more.

Our next level of coverage is called condition coverage. The basic concept is that, when a decision is made by a complex expression that eventually evaluates to TRUE or FALSE, we want to make sure each atomic condition is tested both ways, TRUE and FALSE.

Here is an alternate definition, which we find slightly easier to understand:

An atomic condition is defined as “the simplest form of code that can result in a TRUE or FALSE outcome.”³

Our bug hypothesis is that defects may lurk in untested atomic conditions, even though the full decision has been tested both ways. As always, test data must be selected to ensure that each atomic condition actually be forced TRUE and FALSE at one time or another. Clearly, the more complex a decision expression is, the more test cases we will need to execute to achieve this level of coverage.

Once again, we could evaluate this coverage for values less than 100 percent, by dividing the number of Boolean operand values executed by the total number of Boolean operand values there are. But we won’t. Condition coverage, for this book, will only be discussed as 100 percent coverage.

Note an interesting corollary to this coverage. Decision and condition coverage will always be exactly the same when all decisions are made by simple atomic expressions. For example, for the conditional, *if (a == 1) {}*, condition coverage will be identical to decision coverage.

3. Definition from *The Software Test Engineer’s Handbook* by Graham Bath and Judy McKay (Rocky Nook, 2014).

ISTQB Glossary

atomic condition: A condition that cannot be decomposed, i.e., a condition that does not contain two or more single conditions joined by a logical operator (AND, OR, XOR).

Here, we'll look at some examples of complex expressions, all of which evaluate to TRUE or FALSE.

x

The first, shown above, has a single Boolean variable, *x*, which might evaluate to TRUE or FALSE. Note that in some languages, it does not even have to be a Boolean variable. For example, if this were the C language, it could be an integer, because any non-zero value constitutes a TRUE value and zero is deemed to be FALSE. The important thing to notice is that it is an atomic condition and it is also the entire expression.

D && F

The second, shown above, has two atomic conditions, *D* and *F*, which are combined together by the AND operator to determine the value of the entire expression.

(A || B) && (C == D)

The third is a little tricky. In this case, *A* and *B* are both atomic conditions, which are combined together by the OR operator to calculate a value for the subexpression *(A || B)*. Because *A* and *B* are both atomic conditions, the subexpression *(A || B)* cannot be an atomic condition. However, *(C == D)* is an atomic condition because it cannot be broken down any further. That makes a total of three atomic conditions: *A*, *B*, *(C == D)*.

(a>b)||((x+y== -1)&&((d)!=TRUE))

In the last complex expression, shown above, there are again three atomic conditions. The first, *(a>b)*, is an atomic condition that cannot be broken down further. The second, *(x+y == -1)*, is an atomic condition following the same rule. In the last subexpression, *(d)!=TRUE* is the atomic condition.

Just for the record, if we were to see the last expression in actual code during a code review, we would jump all over the expression with both feet. Unfortunately, it is an example of the way some people program.

In each of the preceding examples, we would need to come up with sufficient test cases that each of these atomic conditions was tested where they eval-

uated both TRUE and FALSE. Should we test to that extent, we would achieve 100 percent condition coverage. That means the following:

- x, D, F, A , and B would each need a test case where it evaluated to TRUE and one where it evaluated to FALSE.
- $(C==D)$ needs two test cases.
- $(a>b)$ needs two test cases.
- $(x+y== -1)$ needs two test cases.
- $((d) != \text{TRUE})$ needs two test cases.

Surely that must be enough testing. Maybe, maybe not. Consider the following pseudo code:

```
if (A && B) then
    {Do something}
else
    {Do something else}
```

In order to achieve condition coverage, we need to ensure that each atomic condition goes both TRUE and FALSE in at least one test case each.

Test 1: $A == \text{FALSE}, B == \text{TRUE}$ resolves to FALSE

Test 2: $A == \text{TRUE}, B == \text{FALSE}$ resolves to FALSE

Assume that our first test case has the values inputted to make A equal to FALSE and B equal to TRUE. That makes the entire expression evaluate to FALSE, so we execute the *else* clause. For our second test, we reverse that so A is set to TRUE and B is set to FALSE. That evaluates to FALSE so we again execute the *else* clause.

Do we now have condition coverage? A was set both TRUE and FALSE, as was B . Sure, we have achieved condition coverage. But ask yourself, do we have decision coverage? The answer is no! At no time, in either test case, did we force the predicate to resolve to TRUE. Condition coverage does not automatically guarantee decision coverage.

That is an important point. We made the distinction that decision coverage was *stronger* than statement coverage because 100 percent decision coverage implied 100 percent statement coverage (but not vice versa). In the 2011 ISTQB Foundation level syllabus, section 4.4.3 has the following sentence: “There are stronger levels of structural coverage beyond decision coverage, for example, condition coverage and ...”

As we showed in the previous example, condition coverage is *not* stronger than decision coverage because 100 percent condition coverage does not imply 100 percent decision coverage. That makes condition coverage—by itself—not very interesting for structural testing. However, it is still an important test design technique; we just need to extend the technique a bit to make it stronger.

We will look at three more levels of coverage that will make condition coverage stronger—at the cost of more testing, of course.

2.3 Decision Condition Coverage

Learning objectives

TTA-2.3.1 (K3) Write test cases by applying the decision condition testing test design technique to achieve a defined level of coverage.

The first of these we will call decision condition coverage. This level of coverage is just a combination of decision and condition coverage (to solve the shortcoming of condition only coverage pointed out in the preceding section). The concept is that we need to achieve condition level coverage where each atomic condition is tested both ways, TRUE and FALSE, *and* we also need to make sure that we achieve decision coverage by ensuring that the overall predicate be tested both ways, TRUE and FALSE.

To test to this level of coverage, we ensure that we have condition coverage, and then make sure we evaluate the full decision both ways. The bug hypothesis should be clear from the preceding discussion; not testing for decision coverage even if we have condition coverage may allow bugs to remain in the code.

Full decision condition coverage guarantees condition, decision, and statement coverage and thereby is stronger than all three.

Going back to the previous example, where we have condition coverage but not decision coverage for the predicate ($A \&\& B$), we already had two test cases as follows:

A is set to FALSE and B is set to TRUE.

A is set to TRUE and B is set to FALSE.

We can add a third test case:

Both A and B are set to TRUE.

ISTQB Glossary

decision condition testing: A white-box test design technique in which test cases are designed to execute condition outcomes and decision outcomes.

We now force the predicate to evaluate to TRUE, so we now have decision coverage also. An alternative would be to select our original test cases a bit more wisely, as follows:

A and *B* are both set to TRUE.

A and *B* are both set to FALSE.

In this case, we can achieve full decision condition coverage with only two test cases.

Whew! Finally, with all of these techniques, we have done enough testing. Right?

Well, maybe.

2.4 Modified Condition/Decision Coverage (MC/DC)

Learning objectives

TTA-2.4.1 (K3) Write test cases by applying the modified condition/decision coverage (MC/DC) testing test design technique to achieve a defined level of coverage.

There is another, stronger level of coverage that we must discuss. This one is called modified condition/decision coverage (usually abbreviated to MC/DC or sometimes MCDC).

This level of coverage is considered stronger than those we have covered (so far) because we add other factors to what we were already testing in decision condition coverage. Like decision condition coverage, MC/DC requires that each atomic condition be tested both ways and that decision coverage must be satisfied. It then adds two more constraints:

1. For each atomic condition, there is at least one test case where the entire predicate evaluates to FALSE only because that specific atomic condition evaluated to FALSE.

2. For each atomic condition, there is at least one test case where the entire predicate evaluates to TRUE only because that specific atomic condition evaluated to TRUE.

There has been some discussion about the correct definition for MC/DC coverage. For example, the standard DO-178C⁴ (discussed below) has a slightly looser definition, as follows:

Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect a decision's outcome. A condition is shown to independently affect a decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.

The definition we are using appears to be slightly more restrictive and so we will use it; this will ensure that when we are using it for safety- and mission-critical testing, it will cover all requirements for MC/DC coverage. By using our version of the definition rather than that in the syllabus or DO-178C, we supply coverage that's at least as good, and it simplifies the way we can create our tests, using the neutral-value design technique.

Note that, in order to use our definition, we need to deal with the unary negation operator. Assume we have the following predicate:

$A \&& (!B)$

According to rule 1, if A is TRUE, the predicate must evaluate to TRUE. Also, if B is TRUE, then the predicate must evaluate to TRUE. However, if B is TRUE, then $(!B)$ evaluates to FALSE and the predicate evaluates to FALSE. The only way we can use our “neutral value” design method (as seen below) is to deem that the atomic condition is the variable and the unary operator together. That is, $(!B)$ is deemed to be the atomic condition that, when it evaluates TRUE (i.e., B is FALSE), fulfills the MC/DC definition by forcing the predicate TRUE. This definition fits within the glossary definition.

4. The ISTQB syllabus refers to standard DO-178B. However, this standard was replaced by DO-178C in January 2012. Therefore, because a few of the definitions are slightly different, we have moved to that standard.

For either definition, when the specified atomic condition is toggled from one state to its opposite, while keeping all other atomic conditions fixed, the entire predicate will also toggle to its opposite value. Interestingly enough, this tends to be a lot less testing than it sounds like. In fact, in most cases, this level of coverage can usually be achieved in $(N + 1)$ test cases, where (N) is the total number of atomic conditions in the predicate.

Many references use the term *MC/DC* as a synonym for exhaustive testing. However, this level of testing does not rise to the level of exhaustive testing, which would include far more than just predicate testing (running every loop every possible number of times, for example). MC/DC does, however, exercise each complex predicate in a number of ways that should result in finding certain types of subtle defects.

MC/DC coverage may not be useful to all testers. It will likely be most useful to those who are testing mission- or safety-critical software. According to the most prevalent legend, it was originally created at Boeing for use when certain programming languages were to be used in safety-critical software development. It is the required level of coverage under the standard FAA DO-178C⁵ when the software is judged to fall into the Level A category (where there are likely to be catastrophic consequences in case of failure).

Avionics systems—which are covered by DO-178C—are a perfect example of why MC/DC coverage is so important. At one time, some software professionals believed that safety-critical software should be tested to a multiple condition level of coverage. That would require that every single possible combination of atomic conditions be tested—requiring essentially 2^n different test cases. See Table 2–3 in section 2.5 to compare the advantage MC/DC coverage has over multiple condition coverage.

There are several different algorithms that have been advanced to define the specific test cases needed to achieve MC/DC-level coverage. Perhaps the simplest to use is the test design technique using neutral values.⁶ This method works quite well with our definition of MC/DC coverage.

A neutral value for an atomic condition is defined as one that has no effect on the eventual outcome evaluation of the expression. The operator used in joining two atomic conditions determines exactly what value will be neutral.

5. DO-178C: Software Considerations in Airborne Systems and Equipment Certification

6. This technique—and our definition—is adapted from *TMap Next – for Result-Driven Testing* by Tim Koomen, Leo van der Aalst, Bart Broekman, and Michiel Vroon (UTN Publishers, 2006).

For example, suppose we are evaluating two atomic conditions, A and B . Keeping it simple, suppose we have two possible operators, AND and OR:

$A \text{ AND } B$

$A \text{ OR } B$

If A was the atomic condition we were interested in evaluating in our MC/DC test design, we would like the output result to be TRUE when A is TRUE and FALSE when A is FALSE. What value of B can we use that will enable that set of values? This is the value we call a neutral, as it has no effect on the evaluation.

If the operator is an AND, then the neutral value will be TRUE. That is, if we set B to the neutral value, we control the output completely by toggling A .

$$(A \text{ AND } B) \rightarrow (\text{TRUE AND } \text{TRUE} == \text{TRUE}), (\text{FALSE AND } \text{TRUE} == \text{FALSE})$$

On the other hand, if the operator is OR, the neutral value must evaluate to FALSE.

$$(A \text{ OR } B) \rightarrow (\text{TRUE OR } \text{FALSE} == \text{TRUE}), (\text{FALSE OR } \text{FALSE} == \text{FALSE})$$

Likewise, if we are interested in evaluating the value B , the value of A would be set to neutral in the same way. By using this concept of neutral substitutes, we can build a table for coming up with sufficient test values to achieve MC/DC coverage.

Let us consider a non-trivial predicate and then determine which test cases we would need to exercise to achieve full MC/DC coverage.

$(A \text{ OR } B) \text{ AND } (C \text{ AND } D)$

The first step is to build a table with three columns and as many rows as there are atomic conditions plus one (the extra row is used as a header). That means, in this case, we start with three columns and five rows, as shown in Table 2–3.

Table 2–3 Non-trivial predicate (1)

$(A \text{ OR } B) \text{ AND } (C \text{ AND } D)$	T	F
A	T -- -- --	F -- -- --
B	-- T -- --	-- F -- --
C	-- -- T --	-- -- F --
D	-- -- -- T	-- -- -- F

The second column will be used for when the predicate evaluates to TRUE and the third column will be used for when the predicate evaluates to FALSE. Each of the rows represents a specific atomic value as shown. Finally, each cell shown at the junction of an output and an atomic condition is set up so that the neutral values are set as dashes. In the second column (representing a TRUE output) and the first row (representing the atomic condition A), the contents represent a value:

TRUE	Neutral	Neutral	Neutral
------	---------	---------	---------

If the values substituted for the neutrals are indeed neutral, then cell (2,2) should contain values that will evaluate to TRUE, and cell (2,3) should contain values that evaluate to FALSE.

The next step (as seen in Table 2–4) is to substitute the correct values for the neutrals in each cell, basing them on the operators that are actually used in the predicate. This can be confusing because of the parentheses. Look carefully at the explanation given for each cell to achieve the possible combinations of inputs that can be used.

Table 2–4 Non-trivial predicate (2)

(A OR B) AND (C AND D)	T	F
A	T F T T	F F T T
B	F T T T	F F T T
C	T F T T	T T F T
D	T F T T	T T T F

- Cell (2,2): (TRUE OR B) AND (C AND D)
 - B is ORed to A, therefore its neutral value is FALSE.
 - (C AND D) is ANDed to (A OR B), and therefore neutral is TRUE.
 - Since (C AND D) must be TRUE, both C and D must be set to TRUE.
- Cell (2,3): (FALSE OR B) AND (C AND D)
 - B is ORed to A, therefore its neutral value is FALSE.
 - (C AND D) is ANDed to (A OR B), and therefore neutral is TRUE.
 - Since (C AND D) must be TRUE, both C and D must be set to TRUE.

- Cell (3,2): (A OR TRUE) AND (C AND D)
 - A is ORed to B, therefore its neutral value is FALSE.
 - (C AND D) is ANDed to (A OR B), and therefore neutral is TRUE.
 - Since (C AND D) must be TRUE, both C and D must be set to TRUE.
- Cell (3,3): (A OR FALSE) AND (C AND D)
 - A is ORed to B, therefore its neutral value is FALSE.
 - (C AND D) is ANDed to (A OR B), and therefore neutral is TRUE.
 - Since (C AND D) must be TRUE, both C and D must be set to TRUE.
- Cell (4,2): (A OR B) AND (TRUE AND D)
 - D is ANDed to C, therefore its neutral value is TRUE.
 - (A OR B) is ANDed to (C AND D), therefore neutral is TRUE.
 - (A OR B) can achieve a value of TRUE in three different ways. It turns out not to matter which of those you use.
 - (F OR T)
 - (T OR F)
 - (T OR T)
- Cell (4,3): (A OR B) AND (FALSE AND D)
 - D is ANDed to C, therefore its neutral value is TRUE.
 - (A OR B) is ANDed to (C AND D), therefore neutral is TRUE.
 - (A OR B) can achieve a value of TRUE in three different ways.
 - It turns out not to matter which of those you use.
 - (F OR T)
 - (T OR F)
 - (T OR T)
- Cell (5,2): (A OR B) AND (C AND TRUE)
 - C is ANDed to D, therefore its neutral value is TRUE.
 - (A OR B) is ANDed to (C AND D), therefore neutral is TRUE.
 - (A OR B) can achieve a value of TRUE in three different ways.
 - It turns out not to matter which of those you use.
 - (F OR T)
 - (T OR F)
 - (T OR T)

■ Cell (5,3): (A OR B) AND (C AND FALSE)

- C is ANDed to D, therefore its neutral value is TRUE.
- (A OR B) is ANDed to (C AND D), therefore neutral is TRUE.
- (A OR B) can achieve a value of TRUE in three different ways.
- It turns out not to matter which of those you use.
 - (F OR T)
 - (T OR F)
 - (T OR T)

Okay, that is a mouthful. We have made it look really complex, but it is not. In truth, when you do this yourself, it will take a very short time (assuming you do not have to write it all out in text form for a book).

There is one final step you must take, and that is to remove redundant test cases (shown in Table 2-5). This is where it might get a little complex. When substituting values for (A OR B) in the final two rows, note that you have a choice whether to use (T OR T), (T OR F), or (F OR T). All three expressions evaluate to TRUE, which is the required neutral value.

Our first try was to use (T OR T) in all four cells where we could make that substitution. When we started to remove redundant test cases, we found that we could remove only two, leaving six test cases that we must run to get MC/DC coverage. Since the theory of MC/DC says we will often end up with $(N + 1)$, or five test cases, we wondered if we could try alternate values, which might allow us to remove the sixth test case and get down to the theoretical number of five. By substituting the logical equivalent of (T OR F) in cells (4,2) and (5,2), we were able to remove three redundant test cases, as can be seen in Table 2-5.

Table 2-5 Test cases for non-trivial predicate

(A OR B) AND (C AND D)	T	F
A	T F T T	F F T T
B	F T T T	F F T T
C	T F T T	T T F T
D	T F T T	T T T F

This leaves us five ($N + 1$) tests, as follows:

- Test 1: A = TRUE, B = FALSE, C = TRUE, D = TRUE
- Test 2: A = FALSE, B = TRUE, C = TRUE, D = TRUE
- Test 3: A = FALSE, B = FALSE, C = TRUE, D = TRUE
- Test 4: A = TRUE, B = TRUE, C = FALSE, D = TRUE
- Test 5: A = TRUE, B = TRUE, C = TRUE, D = FALSE

2.4.1 Complicating Issues: Short-Circuiting

These two issues affect both MC/DC and multiple condition testing (which we will discuss next). In both of these design techniques, we are trying to test a complex predicate with multiple test cases. However, the actual number of test cases needed may be complicated by two different issues: short-circuiting and multiple occurrences of an individual atomic condition.

First, we will look at short-circuiting. Well, you might ask, what does that term mean?

Some programming languages are defined in such a way that Boolean expressions may be resolved by the compiler without evaluating every subexpression, depending on the Boolean operator that is used. The idea is to reduce execution time by skipping some nonessential calculations.

Table 2–6 Truth table for $A \parallel B$

Value of A	Value of B	$A \parallel B$
T	T	T
T	F	T
F	T	T
F	F	F

Consider what happens when a runtime system, that has been built using a short-circuiting compiler, resolves the Boolean expression ($A \parallel B$) (see Table 2–6). If A by itself evaluates to TRUE, then it does not matter what the value of B is. At execution time, when the runtime system sees that A is TRUE, it does not even bother to evaluate B . This saves execution time.

C++ and Java are examples of languages whose compilers may exhibit this behavior. Some flavors of C compilers short-circuit expressions. Pascal does not

ISTQB Glossary

short-circuiting: A programming language/interpreter technique for evaluating compound conditions in which a condition on one side of a logical operator may not be evaluated if the condition on the other side is sufficient to determine the final outcome.

provide for short-circuiting, but Delphi (object-oriented Pascal) has a compiler option to allow short-circuiting if the programmer wants it.

Note that both the OR and the AND short-circuit in different ways. Consider the expression $(A \parallel B)$. When the Boolean operator is an OR (\parallel), it will short-circuit when the first term is a TRUE because the second term does not matter. If the Boolean operator was an AND ($\&\&$), it would short circuit when the first term was FALSE because no value of the second term would prevent the expression from evaluating as FALSE.

Short-circuiting may cause a serious defect condition when it is used. If an atomic condition is supplied by the output of a called function and it is short-circuited out of evaluation, then any side effects that might have occurred through its execution are lost. Oops! For example, assume that the actual predicate, instead of

$$(A \parallel B)$$

is

$$(A \parallel funcB())$$

Further suppose that a side effect of $funcB()$ is to initialize a data structure that is to be used later (not necessarily a good programming custom). In the code, the developer could easily assume that the data structure has already been initialized since they knew that the predicate containing the function had been used in a conditional that was expected to have already executed. When they try to use the non-initialized data structure, it causes a failure at runtime. To quote Robert Heinlein, for testers, “there ain’t no such thing as a free lunch.”

On the other hand, short-circuiting may be used to help avoid serious defects. Consider a pointer p that may or may not be NULL. Short-circuiting allows this common code to work:

```
if ((p != NULL) && (*p>0))
```

If short-circuiting was not used and p actually evaluated to NULL, then this code would crash. Jamie has always been wary of such code, but it is widely used.

Because the short-circuiting is going to be dependent on the compiler writer, Jamie would avoid this dependency by writing this code:

```
if (p) {  
    if (*p > 0) {  
    }  
}
```

If the B expression is never evaluated, the question then becomes, Can we still achieve MC/DC coverage? The answer seems to be maybe. Our research shows that a number of organizations have weighed in on this subject in different ways; unfortunately, it is beyond the scope of this book to cover all of the different opinions. If your project is subject to regulatory statutes, and the development group is using a language or compiler that short-circuits, our advice is to research exactly how that regulatory body wants you to deal with that short-circuiting.

2.4.2 Complicating Issues: Coupling

The second issue to be discussed is when there are multiple occurrences of a condition in an expression. Consider the following pseudo-code predicate:

$$A \parallel (!A \&\& B)$$

In this example, A and $\neg A$ are said to be coupled. They cannot be varied independently as the MC/DC coverage rule says they must. So how does a test analyst actually need to test this predicate and still achieve the required coverage? As with short-circuiting, there are (at least) two approaches to this problem.

One approach is called *unique cause MC/DC*. In this approach, the term *condition* in the definition of MC/DC is deemed to mean *uncoupled condition* and the question becomes moot.

The other approach is called *masking MC/DC*, and it permits more than one atomic condition to vary at once but requires analyzing the logic of the predicate on a case-by-case basis to ensure that only one atomic condition of interest influences the decision. Once again, our suggestion is to follow whatever rules are imposed upon you by the regulatory body that can prevent the release of your system.

Wow! This is really down in the weeds. Should you test to the MC/DC level? Well, if your project needed to follow the FAA DO-178C standard, and the particular software was of Level A criticality, the easy answer is yes, you would need

to test to this level. Level A criticality means that, if the software were to fail, catastrophic results would likely ensue. If you did not test to this level, you would not be able to sell your software or incorporate it into any module for the project. We will discuss this standard and others that might affect your level of coverage later in the chapter.

As always, context matters. The amount of testing should be commensurate with the amount of risk.

2.5 Multiple Condition Coverage

Learning objectives

TTA-2.5.1 (K3) Write test cases by applying the multiple condition testing test design technique to achieve a defined level of coverage.

In this section, we come to the natural culmination of all of the previously discussed control flow coverage techniques. Each one gave us a little higher level of coverage, usually by adding more tests. The final control flow coverage level to discuss is called multiple condition coverage. The concept is to test every possible combination of atomic conditions in a decision! This one is truly exhaustive testing as far as the combinations of atomic conditions that a predicate can take on. Go through every possible permutation—and test them all.

Creating a set of tests does not take much analysis. Create a truth table with every combination of atomic conditions, TRUE and FALSE, and come up with values that test each one. The number of tests is proportional to the number of atomic conditions, using the formula $2n$, where n is the number of atomic conditions (assuming no short-circuiting).

The bug hypothesis is really pretty simple also—bugs could be anywhere, lurking in some esoteric combinations of atomic conditions that are possible but unlikely!

Once again, we could conceivably measure the amount of multiple condition coverage by calculating the theoretical possible numbers of permutations we could have and dividing those into the number that we actually test. However, instead we will simply discuss the hypothetical perfect: 100 percent coverage.

ISTQB Glossary

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).

A simple predicate can be tested without creating a truth table. For example, to get multiple condition coverage of

if (A AND B) then

we can calculate the number of tests needed (2²) and simply enumerate through them as follows, giving us four tests to run:

$A = \text{TRUE}, B = \text{TRUE}$

$A = \text{TRUE}, B = \text{FALSE}$

$A = \text{FALSE}, B = \text{TRUE}$

$A = \text{FALSE}, B = \text{FALSE}$

Let's use a non-trivial predicate to see how many different test cases we are looking at, assuming no short-circuiting. Here is a conditional decision that might be used:

if (A&& B && (C || (D && E))) then ...

We can devise the truth table shown in Table 2-7. Since there are five atomic conditions, we would expect to see 25, or 32, unique rows in the table.

Table 2-7 Truth table

Test	A	B	C	D	E
1	T	T	T	T	T
2	T	T	T	T	F
3	T	T	T	F	T
4	T	T	T	F	F
5	T	T	F	T	T
6	T	T	F	T	F
7	T	T	F	F	T
8	T	T	F	F	F

9	T	F	T	T	T
10	T	F	T	T	F
11	T	F	T	F	T
12	T	F	T	F	F
13	T	F	F	T	T
14	T	F	F	T	F
15	T	F	F	F	T
16	T	F	F	F	F
17	F	T	T	T	T
18	F	T	T	T	F
19	F	T	T	F	T
20	F	T	T	F	F
21	F	T	F	T	T
22	F	T	F	T	F
23	F	T	F	F	T
24	F	T	F	F	F
25	F	F	T	T	T
26	F	F	T	T	F
27	F	F	T	F	T
28	F	F	T	F	F
29	F	F	F	T	T
30	F	F	F	T	F
31	F	F	F	F	T
32	F	F	F	F	F

Theoretically, we would then create a unique test case for every row.

Why theoretically? Well, we might have the same issues that popped up earlier, the concept of short-circuiting and coupling. Depending on the order of the Boolean operators, the number of test cases that are interesting (i.e., achievable in a meaningful way) will differ when we have a short-circuiting compiler.

Let's assume that we are using a compiler that does short-circuit predicate evaluation. Again, here is our predicate:

$$(A \&\& B \&\& (C \parallel (D \&\& E)))$$

Necessary test cases, taking into consideration short-circuiting, are seen in Table 2–8. Note that we started off with 32 different rows depicting all of the possible combinations that five atomic conditions could take, as shown in Table 2–7.

Table 2–8 Tests (1)

Test	A	B	C	D	E	Decision
1	F	-	-	-	-	F
2	T	F	-	-	-	F
3	T	T	F	F	-	F
4	T	T	F	T	F	F
5	T	T	F	T	T	T
6	T	T	T	-	-	T

The first thing we see is that 16 of those rows start with A being set to FALSE. Because of the AND operator, if A is FALSE, the other four terms will all be short-circuited out and never evaluated. We must test this once, in test 1, but the other 15 rows (18–32) are dropped as being redundant testing.

Likewise, if B is set to FALSE, even with A TRUE, the other terms are short-circuited and dropped. Since there are eight rows where this occurs (9–16), seven of those are dropped and we are left with test 2.

In test 3, if D is set to FALSE, the E is never evaluated.

In tests 4 and 5, each term matters and hence is included.

In test 6, once C evaluates to TRUE, D and E no longer matter and are short-circuited.

Out of the original possible 32 tests, only 6 of them are interesting.

For our second example, we have the same five atomic conditions arranged in a different logical configuration, as follows:

$$((A \parallel B) \&\& (C \parallel D)) \&\& E$$

In this case, we again have five different atomic conditions and so we start with the same 32 rows possible as shown in Table 2–7. Notice that anytime E is

FALSE, the expression is going to evaluate to false. Since there are 16 different rows where E is FALSE, you might think that we would immediately get rid of 15 of them. While a smart compiler may be able to figure that out, most won't because the short-circuiting normally goes from left to right.

This predicate results in a different number of tests, as seen in Table 2–9.

Table 2–9 Tests (2)

Test	A	B	C	D	E	Decision
1	F	F	-	-	-	F
2	F	T	F	F	-	F
3	F	T	F	T	F	F
4	F	T	F	T	T	T
5	F	T	T	-	F	F
6	F	T	T	-	T	T
7	T	-	F	F	-	F
8	T	-	F	T	F	F
9	T	-	F	T	T	T
10	T	-	T	-	F	F
11	T	-	T	-	T	T

For test 1, we have both A and B being set to FALSE. That makes the C and D and E short-circuit because the AND signs will ensure that they do not matter. We lose seven redundant cases right there.

For test 2, we lose a single test case because with C and D both evaluating to FALSE, the entire first four terms evaluate to FALSE, making E short-circuit.

With tests 3 and 4, we must evaluate both expressions completely because the expression in the parentheses evaluates to TRUE, making E the proximate cause of the output expression.

For both tests 5 and 6, the D atomic condition is short-circuited because C is TRUE; however, we still need to evaluate E because the entire parentheses evaluates to TRUE. You might ask how can we short-circuit D but not E ? Remember that the compiler will output code to compute values in parentheses first; therefore, the calculation inside the parentheses can be short-circuited, but the expressions outside the parentheses are not affected.

Likewise, for tests 7, 8, 9 10, and 11, we short circuit B since A is TRUE. For test 7, we can ignore E because the subexpression inside the parentheses evaluates to FALSE. Tests 8 and 9 must evaluate to the end because the calculation in the parentheses evaluates to TRUE. And finally, 10 and 11 also short-circuit D .

Essentially, when we're testing to multiple condition coverage with a short-circuiting language, each subexpression must be considered to determine whether short-circuiting can be applied or not. Of course, this is an exercise that can be done during the analysis and design phase of the project when we are not on the critical path (before the code is delivered into test). Every test case that can be thrown out because of this analysis of short-circuiting will save us time when we are on the critical path (i.e., actually executing test cases).

Finally, as promised, we will give you a real-life comparison of the difference between MC/DC coverage and multiple condition coverage. The following table, Table 2–10, is drawn from a NASA paper⁷ that discusses MC/DC techniques. The specific software discussed is written in Ada; it shows all the Boolean expressions with n conditions in five different line replaceable units (LRUs) that, for DO-178C purposes, are rated to be Level A. The five LRUs came from five different airborne systems from 1995.

Table 2–10 Comparing MC/DC with MC coverage

	Number of Conditions in Expression, n					
	1	2	3	4	5	6–10
Number of Expressions with n conditions	16,491	2,262	685	391	131	219
Multiple condition tests needed: (n^2)	2	4	8	16	32	64–1024
MC/DC tests needed: $(n+1)$	2	3	4	5	6	7–11

7. *A Practical Tutorial on Modified Condition/Decision Coverage* by Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson, <http://shemesh.larc.nasa.gov/fm/papers/Hayhurst-2001-tm210876-MCDC.pdf>

	Number of Conditions in Expression, n			
	11–15	16–20	21–35	36–76
Number of Expressions with n conditions	35	36	4	2
Multiple condition tests needed: (n^2)	2,048–32,768	65,536–1,048,576	2,097,152–34,359,738,368	68,719,476,736–75,557,863,725,914,323,419,136
MC/DC tests needed: $(n+1)$	12–16	17–21	22–36	37–77

The first row is split into columns, each representing the number of atomic conditions that occur in the expression. To keep the table reasonably sized, larger numbers of atomic conditions are put into ranges; for example, the last column shows expressions that had between 36 and 76 unique atomic conditions in them. Airborne systems can obviously get rather complicated!

The second row shows the number of times (in the five LRU systems) that each different-sized expression was seen. For example, there were 16,491 times that expressions were simple (i.e., containing only a single atomic condition). On the other hand, there were 219 times that an expression contained between 6 and 10 atomic conditions inclusive.

The third row shows the number of test cases that would be required to achieve multiple condition coverage considering the number of atomic conditions in the expression. So, if there were 16 atomic conditions, it would require 216 (65,536) tests to achieve coverage, and if there were 20, it would require 1,048,576 separate test cases to achieve coverage. Luckily, there are only 36 separate expressions requiring that much testing!

The last row estimates how many tests would be needed to achieve MC/DC coverage for those same predicates. Since we can usually achieve MC/DC coverage with $n+1$ number of tests, the given values use that calculation. In some cases, it might take one or two more tests to achieve coverage.

As you can see in the table, expressions with up to five atomic conditions are the most common (19,960 of 20,256 predicates), and they could conceivably be easily tested using multiple condition testing coverage. For more complex predicates, however, the number of test cases required to achieve multiple condition coverage increases logarithmically. Since the likelihood of defects probably is much higher in more complex predicates, the value of MC/DC coverage testing should be apparent.

We will further discuss the DO-178C standard and others that require specific coverage later in this chapter.

We are not sure when multiple condition testing might actually be used in today's world. MC/DC seems to be the method used when safety and/or mission are critical. We do know that, at one time, multiple condition testing was used for systems that were supposed to work for decades without fail; telephone switches were so tested back in the 1980s.

2.5.1 Control Flow Exercise

In Table 2–11 is a snippet of Delphi code that Jamie wrote for a test management tool. This code controlled whether to allow a drag-and-drop action of a test case artifact to continue or whether to return an error and disallow it.

- Tests are leaf nodes in a hierarchical feature/requirement/use case tree.
- Each test is owned by a feature, a requirement, or a use case.
- Tests cannot own other tests.
- Tests can be copied, moved, or cloned from owner to owner.
- If a test is dropped on another test, it will be copied to the parent of the target test.
- This code was designed to prevent a test from being copied to its own parent, unless the intent was to clone it.
- This code was critical ... and buggy

Using this code snippet:

1. Determine the total number of tests needed to achieve decision condition testing.
2. Determine the total number of tests required to achieve MC/DC coverage (assuming no short-circuiting).
3. Determine the total number of tests needed for multiple condition coverage (assuming no short-circuiting).
4. If the compiler is set to short-circuit, which of those multiple condition tests are actually needed?

Table 2-11 Code snippet

```
// Don't allow test to be copied to its own parent when dropped on test
If (      (DDSourceNode.StringData2 = 'Test')
    AND (DDTgtNode.StringData2 = 'Test')
    AND (DDCtrlPressed OR DDSShiftPressed)
    AND (DDSourceNode.Parent = DDTgtNode.Parent)
    AND (NOT DoingDuplicate)) Then Begin
    Raise TstBudExcept.Create("You may not copy test to its own parent.");
End;
```

The answers are shown in the following section.

2.5.2 Control Flow Exercise Debrief

We find it is almost always easier to rewrite this kind of predicate using simple letters as variable names than to do the analysis using the long names. The long names are great for documenting the code—just hard to use in an analysis.

```
If (      (DDSourceNode.StringData2 = 'Test')
    AND (DDTgtNode.StringData2 = 'Test')
    AND (DDCtrlPressed OR DDSShiftPressed)
    AND (DDSourceNode.Parent = DDTgtNode.Parent)
    AND (NOT DoingDuplicate)) Then Begin
```

This code translates to

if ((A) AND (B) AND (C OR D) AND (E) AND (NOT F⁸)) then

Where

```
A = DDSourceNode.StringData2 = 'Test'
B = DDTgtNode.StringData2 = 'Test'
C = DDCtrlPressed
D = DDSShiftPressed
E = DDSourceNode.Parent = DDTgtNode.Parent
F = DoingDuplicate
```

8. Note that this is the Delphi way of saying (!F)

1. Determine the tests required for decision condition coverage.

Remember, to achieve decision condition level coverage, we need two separate rules to apply:

1. Each atomic condition must be tested both ways.
2. Decision coverage must be satisfied.

It is a matter then of making sure we have both of those covered.

Table 2–12 Decision condition coverage

Test #	A	B	C	D	E	F	Result
1	T	T	T	T	T	F	T
2	F	F	F	F	F	T	F

Tests 1 and 2 in Table 2–12 evaluate each of the atomic conditions both ways and give us decision coverage. If this looks strange, remember that the last atomic condition that will be calculated is actually $\neg F$. But when we pass in the actual variable value in the test case, it will be just F (as represented in the table).

2. Determine the total number of tests required to achieve MC/DC coverage (assuming no short-circuiting) and define them.

There are three rules to modified condition/decision coverage:

1. Each atomic condition must be evaluated both ways.
2. Decision coverage must be satisfied.
3. Each atomic condition must affect the outcome independently when the other conditions are held steady: a TRUE atomic condition results in a TRUE output, a FALSE atomic condition results in a FALSE output.

The first step is to create a neutrals table, three columns and $(N + 1)$ rows.

Table 2–13 Initial neutrals table

(A) AND (B) AND (C OR D) AND (E) AND (NOT F)	T	F
A	T -- -- -- -- --	F -- -- -- -- --
B	-- T -- -- -- --	-- F -- -- -- --
C	-- -- T -- -- --	-- -- F -- -- --
D	-- -- -- T -- --	-- -- -- F -- --
E	-- -- -- -- T --	-- -- -- -- F --
F	-- -- -- -- -- T	-- -- -- -- -- F

The following are neutral values for the six atomic conditions in the table:

- A: A is ANDed to B, therefore its neutral value is TRUE.
- B: B is ANDed to A, therefore its neutral value is TRUE.
- C: C is ORed to D, therefore its neutral value is FALSE.
- D: D is ORed to C, therefore its neutral value is FALSE.
- (C OR D): this expression is ANDed to B, therefore its neutral value is TRUE and may be achieved three ways:
(T,T), (T,F), or (F,T)
- E: E is ANDed to the expression (C OR D), therefore its neutral value is T.
- F: (*NOT F*) is ANDed to E, but it is inverted inside the parentheses, therefore its neutral value is FALSE.

Substituting these values into Table 2–13 creates Table 2–14. Then, simply eliminate the duplicates.

Table 2–14 Completed MC/DC Table

(A) AND (B) AND (C OR D) AND (E) AND (<i>NOT F</i>)	T	F
A	T T F T T F	F T T T T F
B	T T T F T F	T F T T T F
C	T T T F T F	T T F F T F
D	T T F T T F	T T F F T F
E	T T F T T F	T T T T F F
NOT F	T T F T T F	T T T T T T

There are two details that will help you understand these results.

1. The theory says that we need to have a test case for each atomic condition such that if the atomic condition goes TRUE, it makes the entire predicate go TRUE, and that if it toggles FALSE, then the predicate goes FALSE. Looking at the F atomic condition (row 7), this appears to not be fulfilled. When F goes FALSE, the predicate goes TRUE and vice versa. That is because the F variable is negated with a unary operator: that is, (*NOT F*). The actual atomic condition in this case is (*NOT F*), so the rule still is met.
2. For rows 2 (A), 3 (B), 6 (E), and 7 (F), the (C OR D) expression must evaluate to TRUE. We have the choice of three separate value sets that can be used to

achieve a TRUE value: (T,T), (T,F), or (F,T). In order to reduce the number of test cases to $(N + 1)$, you often need to try different combinations. In this case, it took us seven minutes to find combinations that allowed us to reduce the 12 possible test cases to 7.

3. Determine the total number of tests needed for multiple condition coverage.

With 6 different atomic conditions, we can build a truth table to determine the number of multiple condition tests that we would need to test without short-circuiting. Table 2-15 contains the truth tables to cover the six variables; hence 64 tests would be needed for multiple condition coverage.

Table 2-15 Truth tables for 6 values

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T		
B	T	T	T	T	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F		
C	T	T	T	T	T	T	T	F	F	F	F	F	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F	F	F			
D	T	T	T	T	F	F	F	T	T	T	T	F	F	F	T	T	T	T	F	F	F	T	T	T	T	F	F	F	F			
E	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	F			
F	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F			

	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
A	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F		
B	T	T	T	T	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F		
C	T	T	T	T	T	T	T	F	F	F	F	F	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F	F	F			
D	T	T	T	T	F	F	F	T	T	T	T	F	F	F	T	T	T	T	F	F	F	T	T	T	T	F	F	F	F			
E	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	F			
F	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F			

4. If the compiler is set to short-circuit, which of those tests are actually needed?

If the compiler generated short-circuiting code, the actual number of test cases would be fewer. We try to do this in a pattern based on the rules of short-circuiting. That is, the compiler generates code to evaluate from left to right subject to the general rules of scoping and parentheses. As soon as the outcome is determined, it stops evaluating.

Table 2-16 Short-circuited test cases

	A	B	C	D	E	F	A AND B AND (C OR D) AND E AND NOT F
1	F	-	-	-	-	-	F
2	T	F	-	-	-	-	F
3	T	T	F	F	-	-	F
4	T	T	F	T	F	-	F
5	T	T	T	-	F	-	F
6	T	T	F	T	T	T	F
7	T	T	T	-	T	F	T
8	T	T	T	-	T	T	F
9	T	T	F	T	T	F	T

Therefore, we would actually need to run 9 test cases to achieve multiple condition coverage when the compiler short-circuits.

2.6 Path Testing

Learning objectives

TTA-2.6.1 (K3) Write test cases by applying the path testing test design technique.

We have looked at several different control flow coverage schemes. But there are other ways to design test cases using the structure of the system as the test basis. We have covered three main ways of approaching white-box test design so far:

1. Statement testing where the statements themselves drove the coverage
2. Decision/branch testing where the decision predicate (as a whole) drove the coverage
3. Condition, decision condition, modified condition/decision coverage, and multiple condition coverage, all of which looked at subexpressions and atomic conditions of a particular decision predicate to drive the coverage

Now we are going to discuss path testing. Rather than concentrating merely on control flows, path testing is going to try to identify interesting paths through the code that we may want to test. Frankly, in some cases we will get some of the

ISTQB Glossary

path testing: A white-box test design technique in which test cases are designed to execute paths.

same tests we got through control flow testing. On the other hand, we might come up with some other interesting tests.

To try to get the terminology right, consider the following definition from Boris Beizer's book *Software Testing Techniques*:

A path through the software is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit. A path may go through several junctions, processes, or decisions one or more times.

When we discuss path testing, we start out by dismissing one commonly held but bad idea. Brute-force testing—that is, testing every independent path through the system. Got infinity? That's how long it would take for any non-trivial system. Loops are the predominant problem here. Every loop is a potential black hole where each different iteration through the loop creates twice as many paths. We would need infinite time and infinite resources. Well, we said it was a bad idea...

Perhaps we can subset the infinite paths to come up with a smaller number of possible test cases that are interesting and useful. There are several ways of doing this. We will concentrate on two.

2.6.1 Path Testing via Flow Graphs

The first method that we will discuss comes from Boris Beizer; in his book *Software Testing Techniques*, he posits that statement and branch coverage of the source code, by itself, is a very poor indicator of how thoroughly the code has actually been tested.

Assume that you have outlined a set of tests that achieve statement coverage of the source code for a module. Now run that code through the compiler to generate object code that is executable on the platform of choice.

It is logical to assume that running those defined tests against the generated code would achieve statement-level coverage—that is, every generated statement of object code would be executed. However, that does not necessarily follow. In fact, depending on the number of decisions in the source code, the testing done with those tests might end up covering less than 75 percent of the

actual executable statements in the object code. Depending on the programming language and the compiler used, decision/branch coverage of the source code might not generate even statement coverage at the object level. Or, as Beizer puts it:

The more we learn about testing, the more we realize that statement and branch coverage are minimum floors below which we dare not fall, rather than ceilings to which we should aspire.

We will not discuss why this disparity of coverage exists when dealing with compiled code. It is well beyond the scope of what we hope to achieve with this book. If you are interested in this topic, we suggest investigating compiler theory and design books that are available.

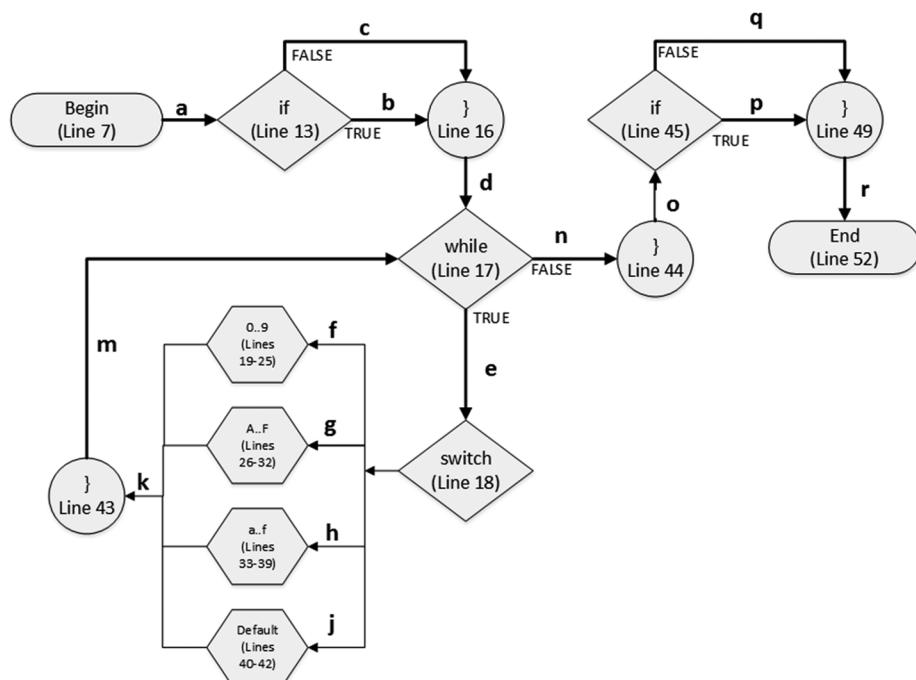


Figure 2–12 Control flow graph for hexadecimal converter

Beizer's answer to this coverage issue is to use control flow graphs to visually pick interesting (and perhaps some uninteresting) paths through the code. There must be sufficient paths selected to supply both statement and decision/branch coverage (remember, he considers that the absolute minimum testing to be

done). Beizer is unconcerned and does not discuss the number of those paths that should be selected for testing (over and above full branch coverage). Instead, he states two general rules for selecting the paths:

1. Many simple paths are better than fewer complicated paths.
2. There is no conceptual problem executing the same code more than once.

The first step in this path coverage technique is to create a control flow graph for the code we want to test. In the interest of simplicity, we will use the code for the hexadecimal converter, already introduced earlier in this chapter (Figure 2-11).

We have fashioned this control flow graph a bit differently than previous flow graphs in this chapter to allow us to better explain how we can use it for path testing.

Note that each code segment between objects in the flow graph is lettered such that we can denote a path by stringing those letters together. The test case would require input data to force the decisions to take the paths desired. For example, a possible path that a tester could test, from beginning to end, might be

abdegkmehkmejkmnoqr

[a]	This path would start at the beginning (line 7)	--
[b]	Decide TRUE at the <i>if()</i> statement (line 13)	Executing in foreground ⁹
[d]	Proceed out of the <i>if()</i> statement (line 16)	--
[e]	Decide TRUE for the <i>while()</i> predicate (line 17)	C
[g]	Go into handler that deals with (A..F)	--
[k]	Proceed out of handler (line 43)	--
[m]	Proceed back to <i>while</i> loop (line 17)	--
[e]	Decide TRUE for the <i>while()</i> predicate (line 17)	d
[h]	Go into handler that deals with (a..f)	--
[k]	Proceed out of handler (line 43)	--
[m]	Proceed back to <i>while()</i> loop (line 17)	--
[e]	Decide TRUE for the <i>while()</i> predicate (line 17)	R
[j]	Go into handler that deals with non-hex character	--

9. This code is designed to run in UNIX. The *if()* statement in line 13 will always evaluate to TRUE when the application is running in the foreground. To test the FALSE branch, it would require that the application be running in the background (with input coming from a device or file other than a keyboard). Because this is already way down in the weeds, we will leave it there.

[k]	Proceed out of handler (line 43)	--
[m]	Proceed back to <i>while()</i> loop (line 17)	--
[n]	Decide FALSE for the <i>while()</i> predicate	EOF
[o]	Proceed to end of <i>while()</i> (line 44)	--
[q]	Decide FALSE at the <i>if ()</i> statement (line 45)	There are hex digits
[r]	Proceed to end of function	--

This set of path segments would be driven by the inputs (or states of the system) as seen in the right-hand column. The expected output would be as follows:

“Got 2 hex digits: cd”

Once we have a control flow diagram, we still need to have some way to determine coverage. To that end, we can create a table to store the paths and facilitate the analysis as to which tests we find interesting and want to run. Our example is shown in Table 2–17. The table should include the paths we decide to test, the decisions that are made in those tests, and the actual path segments that are covered in each test.

The decision columns help us determine whether we have decision coverage given the paths we have chosen. After filling in the paths we decide are interesting, we know that we have decision coverage if each decision column has at least one TRUE (T) and one FALSE (F) entry in it. Notice that we have a switch statement in line 18 that has four possible paths through it (f, g, h, and j). Each of those columns must have at least one entry showing that the path segment has been traversed or we have not achieved decision coverage through that construct.

Likewise, we can ensure that we have statement coverage by ensuring that each code segment has been checked (with an x in this case) in at least one path instance.

For example, path number one in Table 2–17 has been entered to show the route as illustrated earlier. In this case, decision 13 has been marked T, decision 17 has been marked T (even though it eventually goes F to fall out of the *while()* loop), decision 45 has been marked F (because we did have hex digits), and the switch statement has been marked showing three segments covered, so we have partial coverage of the switch statement at this point. The fact that some segments have been traversed multiple times is not recorded and does not matter for the purposes of this analysis.

Table 2–17 Control flow for hex converter

Paths	Decisions							Process Link															
			Switch Statement																				
	13	17	0-9	A-F	a-f	Def	45	a	b	c	d	e	f	g	h	j	k	m	n	o	p	q	r
1	abdegkmehkmejkmnqr	T	T		x	x	x	F	x	x		x	x		x	x	x	x	x	x	x	x	x
2																							
3																							

Despite the fact that we have a candidate test, we don't know whether it is a "good" test based on Boris Beizer's theories. Beizer recommends an algorithm that takes into account his two general rules of selecting paths. Because his algorithm is salient and succinct, we will quote it completely.

1. *Pick the simplest, functionally sensible entry/exit path.*
2. *Pick additional paths as small variations from previous paths. Pick paths that do not have loops rather than paths that do. Favor short paths over long paths, simple paths over complicated paths, and paths that make sense over paths that don't.*
3. *Pick additional paths that have no obvious functional meaning only if it's necessary to provide coverage. But ask yourself first why such paths exist at all. Why wasn't coverage achieved with functionally sensible paths?*
4. *Be comfortable with your chosen paths. Play your hunches and give your intuition free reign as long as you achieve C1 + C2. (100 percent statement and branch coverage)*
5. *Don't follow rules slavishly—except for coverage.*

Note that we likely break rule 1 with our first path as shown in Table 2–17. It is not the shortest possible path; while it is functionally sensible, it loops several times which also breaks rule 2.

For actually testing this module, we have filled out what we believe are a reasonable set of paths to start to test in Table 2–18, using Beizer's algorithm.

Table 2-18 Completed control flow table

Paths		Decisions							Process Link												
				Switch Statement																	
		13	17	0..9	A..F	a..f	Def	45	a	b	c	d	e	f	g	h	j	k	m	n	o
1	Abdefkmnoqr	T	T	x				F	x	x		x	x	x				x	x	x	x
2	abdegkmnoqr	T	T		x			F	x	x		x	x		x			x	x	x	x
3	abdehkkmnoqr	T	T			x		F	x	x		x	x			x		x	x	x	x
4	Abdejkkmnopr	T	T				x	T	x	x		x	x				x	x	x	x	x
5	Acdnopr	F	F					T	x		x	x						x	x	x	x

Test 1: Test a single digit (0..9) and print it out.

Test 2: Test a single capitalized hex digit (A..F) and print it out.

Test 3: Test a single noncapitalized digit (a..f) and print it out.

Test 4: Test a non-hex digit and show there are no hex digits to convert.

Test 5: Test entering EOF as first input while running the application in the background.

Are these enough tests? We would say not, but they are a good start because they do give us both statement and branch coverage. Test cases 2, 3, and 4 are each a small variation in the previous test case. Test 5 is there only to give us the final coverage that we need.

Note that tests running the application in the background would be interesting, feeding input via a device or file. We know this because we see specific provisions for handling signals in the module. Tests selecting non-trivial strings to test would also be interesting. The five tests just shown should be seen as only the start.

One test that is not included in this basic set is one that tests the signal action itself. The first `if()` statement (line 13) is setting up the ability for the system to respond to a `Ctrl^C` asynchronously. When a `Ctrl^C` is inputted through the keyboard, and the signal is not set to be ignored (i.e., the application is running in the foreground), it removes the last hex digit that was inputted. This allows a user to correct a mistake if they entered an incorrect value. Interestingly enough, a serious defect exists with this signal-handling code. This points out very well that you cannot blindly perform white-box testing based solely on our ideas of coverage. You must understand all of the possible ways the code may be executed, both synchronously and asynchronously.

Of course, this particular defect should have been found—before we ever got to dynamic testing—through a code inspection, but that is covered in Chapter 5.

Beizer recommends a low-tech, graphical way to come up with the interesting paths to test. Create the control graph and then print off copies of it and use a highlighter to mark a single path through each copy. Enter each of those paths into the table. As you do so, the next path you create could cover some of the decisions or path segments that were not already covered. Continue until comfortable with the amount of testing—which should be governed by the context of the project, the application, and the test plan.

Using this approach, it is certainly conceivable that some of the paths will be impossible or uninteresting or have no functional meaning. Beizer addresses that issue in rule 3 of his algorithm.

As long-time developers and testers, we have often written and tested code that sometimes requires unusual looking tests to achieve a level of coverage. Some of those times, the structure was necessary to be able to achieve the needed functionality in an elegant way. Other times, we find that we have locked in on a bone-headed structure that was not needed and could have (should have) been rewritten.

In general, having test analysts criticizing the programmers' choice of structure is not a recommendation that we are always comfortable making. However, in today's world of Agile testers working side by side with programmers and mission- and safety-critical software, perhaps a discussion of that attitude might be in order.

2.6.2 Basis Path Testing

The second method we will look at comes from Thomas McCabe and his theory of cyclomatic complexity. His entirely sensible suggestion was to test the structure of the code to a reasonable amount, and he used mathematics and mapping theory to suggest how.

McCabe called his technique *basis path testing*. In December 1976, he published his paper “A Complexity Measure”¹⁰ in the *IEEE Transactions on Software Engineering*. He theorized that any software module has a small number of unique, independent paths through it (if we exclude iterations). He called these *basis paths*.

10. <http://portal.acm.org/citation.cfm?id=800253.807712> or http://en.wikipedia.org/wiki/Cyclomatic_complexity and click on the second reference at the bottom.

His theory suggests that the structure of the code can be tested by executing through this small number of paths and that all the infinity of different paths actually consists of simply using and reusing the basis paths.

A basis path is defined as a linearly independent path through the module. No iterations are allowed for these paths. When you're dealing with looping structures, the loop predicate will generally be visited twice: once TRUE (to travel through the loop once) and once FALSE (to skip the loop). The basis path set is the smallest number of basis paths that cover the structure of the code.

The theory states that creating a set of tests that cover these basis paths, the basis set, will guarantee us both statement and decision coverage of testing. The basis path has also been called the minimal path for coverage.

Cyclomatic complexity is what Thomas McCabe called this theory of basis paths. The term *cyclomatic* is used to refer to an imaginary loop from the end of a module of code back to the beginning of it. How many times would you need to cycle through that outside loop until the structure of the code has been completely covered? This cyclomatic complexity number is the number of loops needed and—not coincidentally—the number of test cases we need to cover the set of basis paths.

The cyclomatic complexity depends not on the size of the module but on the number of decisions that are in its structure.

McCabe, in his paper, pointed out that the higher the complexity, the more likely there will be a higher bug count. Some subsequent studies have shown such a correlation; modules with the highest complexity tend to also contain the highest number of defects.¹¹ Since the more complex the code, the harder it is to understand and maintain, a reasonable argument can be made to keep the level of complexity down. McCabe's suggestion was to split larger, more-complex modules into more and smaller, less-complex modules. This complexity issue is not quite as straightforward as we just said; there are nuances to this that we will discuss in Chapter 3 when we discuss static analysis using control flows. For now, however, let's discuss the nuts and bolts of measuring cyclomatic complexity.

We can measure cyclomatic complexity by creating a directed control flow graph. This works well for small modules; we will show an example later. Realistically, however, tools will generally be used to measure module complexity.

11. For example, NIST Special Publication 500-235, *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*, found at <http://www.itl.nist.gov/lab/specpubs/sp500.htm>

In our directed control flow graph, we have nodes (bubbles) to represent entries to the module, exits from the module, and decisions made in the module. Edges (arrows) represent non-branching code statements which do not add to the complexity. It is essential to understand that, if the code does not branch, it has a complexity of one. That is, even if we had a million lines of code with no decisions in it, it could be tested with a single test case.

In general, the higher the complexity, the more test cases we need to cover the structure of the code. If we cover the basis path set, we are guaranteed 100 percent of both statement and decision coverage.

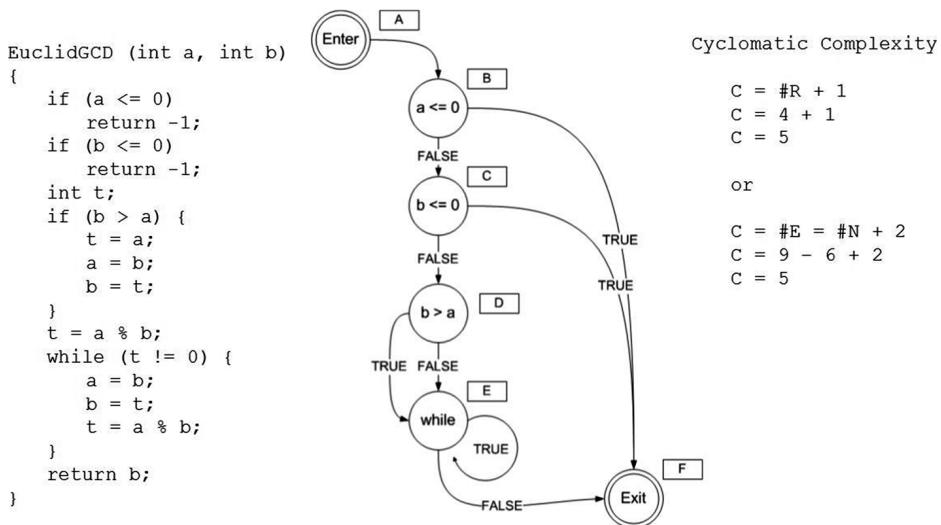


Figure 2-13 Cyclomatic complexity example

On the left side of Figure 2-13 we have a function to calculate the greatest common divisor of two numbers using Euclid's algorithm.

On the right side of the figure, you see the two methods of calculating McCabe's cyclomatic complexity metric. Seemingly the simplest is perhaps the “enclosed region” calculation. The four enclosed regions (R_1, R_2, R_3, R_4), represented by R in the upper equation, are found in the diagram by noting that each decision (bubble) has two branches that—in effect—enclose a region of the graph. We say “seemingly” the simplest because it really depends on how the graph is drawn as to how easy it is to see the enclosed regions.

The other method of calculation involves counting the edges (arrows) and the nodes (bubbles) and applying those values to the calculation $E - N + 2$, where E is the number of edges and N is the number of nodes.

Now, this is all simple enough for a small, modest method like this. For larger functions, drawing the graph and doing the calculation from it can be really painful. So, a simple rule of thumb is this: Count the branching and looping constructs and add 1. The *if* statements, *for*, *while*, and *do/while* constructs, each count as one. For the *switch/case* constructs, each *case* block counts as one. In *if* and *ladder if* constructs, the final *else* does not count. For *switch/case* constructs, the *default* block does not count. This is a rule of thumb, but it usually seems to work. In the sample code, there are three *if* statements and one *while* statement. $3 + 1 + 1 = 5$.

Note that entry and exit nodes are not counted when calculating cyclomatic complexity.

When we introduced McCabe's theory of cyclomatic complexity a bit ago, we mentioned basis paths and basis tests. Figure 2–14 shows the basis paths and basis tests on the right side.

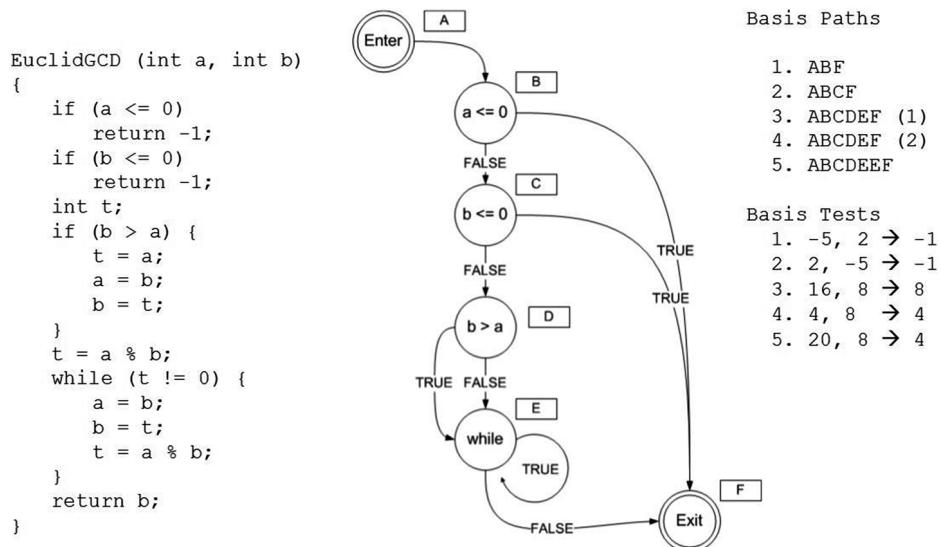


Figure 2–14 Basis tests from directed flow graph

The number of basis paths is equal to the cyclomatic complexity. You construct the basis paths by starting with the most obvious path through the diagram, from enter to exit. Beizer claims that this likely corresponds to the normal path. Then add another path that covers a minimum number of previously uncovered edges, repeating this process until all edges have been covered at least once.

Following the IEEE and ISTQB definition that a test case substantially consists of input data and expected output data, the basis tests are the inputs and expected results associated with each basis path. The basis tests will correspond with the tests required to achieve 100 percent branch coverage. This makes sense, since complexity increases anytime more than one edge leaves from a node in a directed control flow diagram. In this kind of a control flow diagram, a situation where there is “more than one edge coming from a node” represents a branching or looping construct where a decision is made.

What use is that tidbit of information? Well, suppose you were talking to a programmer about their unit testing. You ask how many different sets of inputs they used for testing. If they tell you a number that is less than the McCabe cyclomatic complexity metric for the code they are testing, it’s a safe bet they did not achieve branch coverage.

One more thing on cyclomatic complexity. Occasionally, you might hear someone argue that the actual formula for McCabe’s cyclomatic complexity is as follows:

$$C = E - N + P$$

or

$$C = E - N + 2P$$

We list it as this:

$$C = E - N + 2$$

A careful reading of the original paper does show all three formulae. However, the directed graph that accompanies the former version of the formula ($E - N + P$) also shows that there is an edge drawn from the exit node to the entrance node that is not there when McCabe uses the latter equation ($E - N + 2P$). This edge is drawn differently than the other edges—as a dashed line rather than a solid line, showing that while he is using it theoretically in the math proof, it is not actually there in the code. This extra edge is counted when it is shown, and it must be added when not shown (as in our example). The P value stands for the number of connected components.¹² In our examples, we do not have any connected components, so by definition, $P = 1$. To avoid confusion, we abstract out the P and simply use 2 (P equal to 1 plus the missing theoretical line, connecting the exit to the entrance node). The mathematics of this is beyond the

12. A connected component, in this context, would be a called a subroutine.

scope of this book; suffice it to say, unless an example directed graph contains an edge from the exit node to the enter node, the formula that we used is correct.

We'll revisit cyclomatic complexity later when we discuss static analysis in Chapter 3.

2.6.3 Cyclomatic Complexity Exercise

The following C code function loads an array with random values. Originally, this was a called function in a real program; we have simplified it as a main program to make it easier to understand. As it stands, it is not a very useful program.

Table 2-19 Example code

```

1. int main (int MaxCols, int Iterations, int MaxCount)
2. {
3.     int count = 0, totals[MaxCols], val = 0;
4.
5.     memset (totals, 0, MaxCols * sizeof(int));
6.
7.     count = 0;
8.     if (MaxCount > Iterations)
9.     {
10.         while (count < Iterations)
11.         {
12.             val = abs(rand()) % MaxCols;
13.             totals[val] += 1;
14.             if (totals[val] > MaxCount)
15.             {
16.                 totals[val] = MaxCount;
17.             }
18.             count++;
19.         }
20.     }
21.     return (0);
22. }
```

1. Create a directed control flow graph for this code.
2. Using any of the methods given in the preceding section, calculate the cyclomatic complexity.
3. List the basis tests that could be run.

The answers are in the following section.

2.6.4 Cyclomatic Complexity Exercise Debrief

The directed control flow graph should look like Figure 2–15. Note that the edges D1 and D2 are labeled; D1 is where the *if* conditional in line 14 evaluates to TRUE, and D2 is where it evaluates to FALSE.

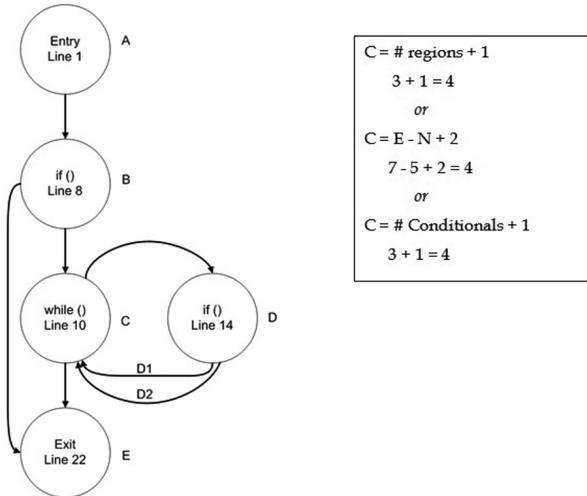


Figure 2–15 Cyclomatic complexity exercise

We could use three different ways to calculate the cyclomatic complexity of the code, as shown in the box on the right in Figure 2–15.

First, we could calculate the number of test cases by the region methods. Remember, a region is an enclosed space. The first region can be seen on the left side of the image. The curved line goes from B to E; it is enclosed by the nodes (and edges between them) B-C-E. The second region is the top edge that goes from C to D and is enclosed by line D1. The third is the region with the same top edge, C to D, and is enclosed by D2. Here is the formula:

$$C = \# \text{ Regions} + 1$$

$$C = 3 + 1$$

$$C = 4$$

The second way to calculate cyclomatic complexity uses McCabe's cyclomatic complexity formula. Remember, we count up the edges (lines between bubbles) and the nodes (the bubbles themselves), as follows:

$$C = E - N + 2$$

$$C = 7 - 5 + 2$$

$$C = 4$$

Finally, we could use our rule-of-thumb measure, which usually seems to work. Count the number of places where decisions are made and add 1. So, in the code itself, we have line 8 (an *if()* statement), line 10 (a *while()* loop), and line 14 (an *if()* statement).

$$C = \# \text{ decisions} + 1$$

$$C = 3 + 1$$

$$C = 4$$

In each case, our basis path is equal to 4. That means our basis set of tests would also number 4. The following test cases would cover the basis paths.

1. ABE
2. ABCE
3. ABCD(D1)CE
4. BCD(D2)CE

2.7 API Testing

Learning objectives

TTA-2.7.1 (K2) Understand the applicability of API testing and the kinds of defects it finds.

From the Advanced Technical Test Analyst syllabus:

An Application Programming Interface (API) is code which enables communication between different processes, programs and/or systems. APIs are often utilized in a client/server relationship where one process supplies some kind of functionality to other processes.

In the old days of computing, APIs did not exist because they were not needed. A central mainframe computer did the processing using punched paper and cards and tape for input and storage. The idea that a single person might sit down and interact with the system in real time was ludicrous.

Then came dumb terminals, allowing multiple individuals to interact with the system in limited ways concurrently. All of the actual processing was still done on the central processor, so there was no need (or place) to offload processing. Mini computers allowed organizations that could not afford the big iron to buy some limited processing power; dumb terminals still ruled the day with all processing done in the central CPU.

As processors started to get smaller and more powerful, the demand for more than time sharing began to develop. Jamie remembers listening to a rant by a real old-time computer programmer complaining that “today’s computer users are spoiled. Someday everyone will want their own powerful computer on their own desk.”

Word processing, data analysis, and decentralized data access: people did indeed want access to their “stuff” that time sharing alone could not satisfy.

As UNIX and other similar mini-computer operating systems were introduced, the possibility of personal computing became a reality. You still did not have a computer on your desk at home; that came a lot later. A small group could have access to a computer network that was not tethered directly to a mainframe; each node in the network had its own processor.

This distributed computing came with its own issues, however. Processing large amounts of data or performing large, complex calculations could not be done on these smaller processors without loading them down such that they no longer supported users interactively. One popular solution that was devised was the ability to request that some of the processing be done on remote processors. This was the advent of the remote procedure call (RPC), which is the direct progenitor of the API.

A number of networks were created that allowed a person working on a single CPU to spread some of their processing to other CPUs that were not currently busy. The ability to distribute some of the processing tasks was not without its own difficulties; if someone turned off their local computer while it was doing remote processing, or wanted to interact with it directly, the remote process might get kicked off without completing its work.

There are really two different kinds of APIs that software engineers deal with. Local APIs are often used to interact with the operating system, databases, and I/O or other processes performing work on the local processor. For example, if a programmer wants to open a file, allocate some heap memory, start a transaction on the database, show some data on screen, or pick up a mouse movement, they make API calls in their programming language to perform the

task. Often, the programmer does not even know they are calling various APIs because the programming language often hides the direct calls.

While those APIs are certainly useful (and sometimes really complex), in this book we are more interested in an alternate definition. The implementation of requests for remote processing can also be called APIs. One phrase we have seen when dealing with this kind of an API is *object exchange protocol*. Essentially, the API implements a protocol that specifies a way for a local process to request that a task be done from a remote location and defines the way the results of that task are returned to the local process. Included in the protocol are techniques for specifying the task to be done, the way to marshal the data to pass to the remote process, the way that information is actually conveyed to the remote location, the way the results are passed back, and the way those results should be interpreted.

As technical test analysts, it is not required that we understand all of the technical details in the implementation of these APIs. We do, however, have to understand how to test them in representative environments and in realistic ways.

APIs offer organization-to-organization connections that promote integration between them. Rather than having every group reinvent the wheel programmatically, cooperation between some aspects of different organizations can save money and resources for all involved. In some cases, a particular business may supply some kind of up-to-date information to other organizations to leverage its own business. For example, Amazon might leverage APIs from UPS, FedEx, and USPS, each of which allows immediate access to shipping rates and delivery time frames. This helps Amazon service its customers and helps the shipping companies compete more effectively for the business. In the old days, Amazon would likely use a printed rate schedule—updated periodically by the shipping companies—to review shipping information. With the use of APIs, a shipping company can run sales and give special rates to good customers, in effect trying to maximize its market share.

As noted earlier, there are no free lunches. Using APIs means that your organization is assuming all risks pertaining to those APIs. Should they fail, the damage is to your organization because the user does not care if it was a remote API that failed. They are going to blame the organization that they were dealing with directly. You can protest and point your finger at the real source of the original error, but it will not matter to the damaged user.

Those same connections that can provide synergy between organizations can also provide pain for all groups involved. In our example, an Amazon cus-

tomer who is charged twice as much as expected, or does not receive the two-day delivery they expected, is not going to be aware that the real problem was caused by failure of the shipper's API; they are going to blame Amazon.

Over the last 15 years the use of APIs has increased astronomically among business organizations, especially on the Internet. As originally envisioned, the Internet was predominately a static environment. A user requested a particular site by typing in a URL (or clicking a link) in their browser and the website would send a static screen of information to that browser. In January 1999, Darcy DiNucci, an electronic information design consultant, prognosticated:

The Web we know now, which loads into a browser window in essentially static screenfuls, is only an embryo of the Web to come. The first glimmerings of Web 2.0 are beginning to appear, and we are just starting to see how that embryo might develop. The Web will be understood not as screenfuls of text and graphics but as a transport mechanism, the ether through which interactivity happens. It will [...] appear on your computer screen, [...] on your TV set [...] your car dashboard [...] your cell phone [...] hand-held game machines [...] maybe even your microwave oven.”¹³

Most would say he understated the evolution of the web. Because an image is worth a thousand words, here is the preceding quote presented graphically:

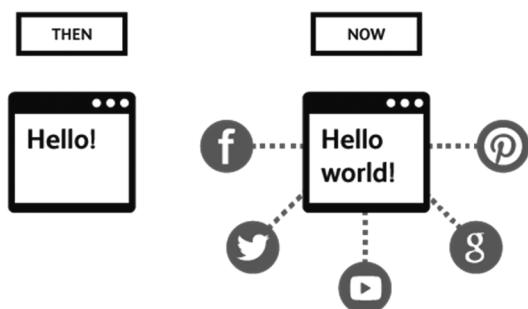


Figure 2–16 The evolution of the Web

An organization is likely to use private, partner, and public APIs. Very often a business organization will use APIs to tie together the disparate systems after merging with another company. This can reduce the amount of work needed—rather than trying to merge all of the systems together physically.

13. *The Golden Age of APIs*, <http://www.soapui.org/The-World-Of-API-Testing/the-golden-age-of-apis.html>

Large organizations led the way in creating APIs. Some large enterprises have so many complex and interlocking APIs that they have to develop a way to manage them, developing a new architecture called the enterprise service bus (ESB).

In Figure 2–17, the exponential growth of public APIs can be seen.¹⁴

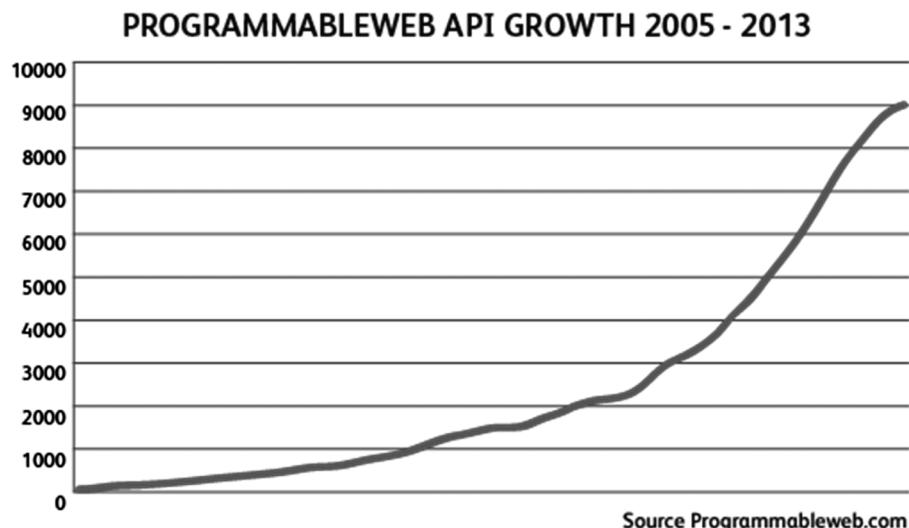


Figure 2–17 The growth of public APIs

In 2013, it was estimated that there were over 10,000 publically available APIs published, more than twice as many as in 2011. It was also estimated that businesses worldwide have millions of private APIs. Looking forward, it is only going to get more intense, with estimates of 30,000 public APIs by 2016 and even estimates of 1,000,000 by 2017.¹⁵ We may be coming to a time when most communication between machine and machine or human and machine will involve APIs. And they all need to be tested.

These figures are not included to startle readers. For testers, this should be seen as a huge opportunity. APIs are not easy to test, which means there is a growing need for skilled technical test analysts (since the heavy use of tools and automation is required to test APIs effectively). The fastest growing segment of computing worldwide is mobile computing. It should not be a surprise to find that mobile environments are substantially run by multiple layers of APIs.

14. Ibid.

15. Ibid.

When an API fails, an organization can be exposed to high levels of risk. Often APIs reveal security, performance, reliability, portability, and many other types of risks. Risks accrue to both the organization providing an API and the organizations consuming it.

Research performed by tool vendor Parasoft along with the Gartner group in 2014¹⁶ found the following:

- Over 80 percent of those responding to the survey claimed they had stopped using one or more APIs because they were too buggy.
- Top impacts of API issues reported:
 - Development delay
 - Increased customer support service costs
 - Time to market delays
 - Testing delays
 - Lost business
- 90 percent of respondents report that one or more APIs failed to meet their expectations.
- 68 percent encountered reliability or functional issues.
- 42 percent encountered security issues.
- 74 percent encountered performance issues.
- The most severe integrity issues encountered with APIs:
 - 61 percent reliability
 - 22.2 percent security
 - 16.7 percent performance

Clearly there may be problems with APIs that require intensive testing. The following issues are among those that must be considered when preparing to test APIs:

- Security issues. This can be a huge problem with the use of APIs because they often have great access to the core data and functional areas of an organization's computing center. The following problems can occur:
 - Worm injections and other payload-based attacks.
 - Ineffective authentication.
 - Insufficient level encryption.

16. API Integrity: An API Economy Must-Have, <http://alm.parasoft.com/api-testing-gartner>

- Unauthorized access control.
- Cloud issues. If the APIs are located in the cloud, security issues might even be worse since there is often little effective control over the physical network security.
- Unexpected misuse. This may often be innocent due to poor documentation or programming errors by a consumer of the API. However, it might also be malicious, which leads us back to security testing.
- Performance issues. There is a wide range of problems that may be detected through proper performance testing:
 - How is the API projected to be used?
 - What will happen if it suddenly becomes popular?
 - Does the API exhibit high availability for worldwide 24/7 usage?
 - Are there times when the usage might peak (e.g., a sports website delivering on-demand statistics during the final game of the World Cup)?
- Portability issues. Your APIs are liable to need to work on a variety of layers. They might be called directly, or they might be called by other APIs, which in turn are called by other APIs in turn controlled by an ESB.
- Documentation. This must be tested for correctness. Since a wide range of developers may need to interact with your APIs, the documentation must provide sufficient information for all of them. And the quality of the documentation is critical. Data order, data types, formatting, even capitalization in the documentation is critical and must be correct.
- Usability. The consumers of APIs are expected to be software developers; can they actually learn and understand how to use the APIs successfully? Clearly, this will partially depend on the documentation mentioned above.
- Testing. Those using your API will likely want to test it with their own layers of APIs. This may require special environments (sandboxes) for them to play in. Do those work correctly without interfering with the production environment?
- Timing issues. APIs are often developed by separate groups. Trying to make sure the schedules of all the participants are coherent is often like herding cats. That means that change becomes a constant during the development phase. One group makes some changes, and then those changes ripple through all of the development teams like a tsunami.
- Discovery. The API is liable to manipulate resources, creating, listing, updating, and deleting them. Are those operations available and correct for all of the ways they may be performed?

- Compliance. Many industries require that software be compliant with various regulations and standards. Regulations may come from governmental and quasi-governmental entities such as the FDA or EPA. Standards may come from international organizations or industry groups. For example, if the software processes credit or debit cards, it must be PCI compliant. Systems that handle health information in any way are subject to HIPAA compliance. APIs may be subject to service-level agreements (SLAs) or other technical agreements. Failure to meet such agreements may cost the organization a great deal in penalties or fines.
- Error handling. Have you tested all the possible error conditions that may occur? Often errors are generated by arguments being passed in incorrectly (e.g., out of order) or out of range.
- Change management. Does your API depend on other APIs? When those change, how will you know? In addition, the data schema for an API is critical. Even a small change to a schema—for example, changing an integer into a real—may cause defects and failures to ripple far downstream. And changes are often rampant.
- Failure management. If (and when) your API fails in production, how will you know it? When (and if) outside organizations stop using your API, will you be able to figure out why and when it occurred?
- Asynchronicity. Because APIs are very loosely coupled, timing glitches and lost and duplicated transactions are a very common failure mode.
- Combinatorial testing. Because APIs are often used in layers—both horizontally and vertically—there turns out to be many different ways to call them. Have you tested all of the boundary values concurrently?
- Protocols used.

Experts note that the amount of time and resources needed to adequately test APIs is often the same as needed to analyze and develop them. This can cause problems with management who are used to much smaller test efforts relative to the development effort.

A recent study performed by a popular automation tool vendor¹⁷ found that the typical application under test had an average of 30 dependencies. Those dependencies include third-party applications, mainframe and mini-computers, enterprise resource planning packages (ERPs), databases, and APIs (private, partner, and public). In this interconnected world, the rules of testing are chang-

17. *Testing in the API Economy: Top 5 Myths*, by Wayne Ariola and Cynthia Dunlop, 2014. A white paper available from Parasoft.

ing. Testing no longer can simply be done during the SDLC and then dropped, waiting for the next cycle.

Many enterprise applications are becoming increasingly interdependent using APIs from private, partner, and public entities. The success of an end-to-end transaction becomes increasingly a function of the working of the composite gestalt with its integrity dependent on its weakest links.

Given all of the risk described above, how does an organization survive or maybe even thrive testing APIs?

Automation of testing is an absolute requirement. Manual techniques simply do not allow enough testing to mitigate the risks noted above, given that a single API by itself may take hundreds of tests that might need to be repeated daily (or more often) as environments evolve, changes to data structures occur, and dependent APIs from partner and public sources mutate. This is especially true since often the APIs must be monitored in production to solve issues there as soon as they crop up (continuous regression testing in production).

Not only must each API be tested alone (often using simulators and/or mock objects), they also needed to be tested together in end-to-end test scenarios. While this can be a huge effort, the alternative is to have untested scenarios actually cause failure in production.

The automation tools that are needed for API testing must have intuitive interfaces, automating as much of the test development as possible as well as the execution of the tests. The protocols used to define the APIs tend to be well developed and documented (if not fully mature). That allows many of the tools in the market today to deliver facilities for creating multiple tests to cover the protocols. This helps the technical test analyst create more and better tests in less time.

Tests on many platforms are required. The automation tools must be able to test across ESBs, databases, architectures, protocols and messaging types. In order to service so many different platforms, virtualization is also an important requirement. Many APIs have limits to the amount of times you can use them in a given period, or you are charged for their use. That means a tool must be able to simulate those resources that would not be directly testable on the fly (or at least with minimal manual actions).

In many ways, what we are describing here sounds a lot like unit testing. However, API testing is not unit testing. Despite the technical details we have discussed, API testing is still essentially black-box testing. Setting up the tests certainly requires technical skills, but at the end of the setup, a figurative button is pushed and the test runs, passing or failing. The reason for a failure is usually not discernable by running a debugger as a developer would in a unit test.

Unit tests are often trying to ensure that a chunk of code works in isolation. API testing must be designed to test the full functionality of the system—although, as noted earlier, we may need to use stubs and drivers as part of our test harness to allow for missing pieces. That makes the API tests much more thorough in their testing of broad scenarios and data flows.

The data used for API testing is a critical part of the overall picture. The data source should be static to the point that using it for testing should not be able to change it. For example, a common issue with regression testing via automation is that the data used for the testing tends to evolve as the tests run. If a regression set runs only partially, or must be reset, the data has changed and must be refreshed. In API testing, allowing the data to change during the testing would place an undue amount of pressure on the test analysts to maintain it. Consider that some APIs may call other APIs, themselves called by yet other APIs. Trying to make sure that a changing data store is coherent would quickly become an impossible task.

To conclude this topic, we have heard API testing—for all of the reasons mentioned—described as similar to juggling chainsaws while seated on a unicycle that is standing on a medicine ball. Attention to detail and great intestinal fortitude is required to succeed. All in a day's work for most technical test analysts...

2.8 Selecting a Structure-Based Technique

Learning objectives

TTA-2.8.1 (K4) Select an appropriate structure-based technique according to a given project situation.

After looking at all of these structure-based techniques, we need to consider one more topic. How much of this stuff do we need to do on our project?

As with so many other topics in software testing, the correct answer is, It depends! The context of the software—how, where, when, how much, and who—will determine how deep we need to get into structural testing.

The more important the software, the more an organization will likely lose when a failure occurs.

*The higher the risk—and cost—of failure, the more testing is needed.
The more testing needed, the higher the cost in time and resources.*

Somewhere, these three related but disjoint imperatives will come to a single sweet spot that will define the amount of testing we need to do. At least that was what we were told when we started testing. Often we find that illumination comes only in the rear-view mirror.

Sometimes the amount of testing, and the techniques we use, is defined not by the testers but by the legal standards or guidelines to which we are required to adhere. In this section, we will look at some standards that may force our choices.

The British Standards Institute produces the BS 7925/2 standard. It has two main sections, test design techniques and test measurement techniques. For test design, it reviews a wide range of techniques, including black-box, white-box, and others. It covers the following black-box techniques that were also covered in the Foundation syllabus:

- Equivalence partitioning
- Boundary value analysis
- 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.

It covers the following white-box techniques that were also covered in the Foundation syllabus:

- Statement testing
- Branch and decision testing

It also covers some additional white-box techniques that were covered only briefly or not at all in the Foundation syllabus:

- Data flow testing
- Branch condition testing
- Branch condition combination testing (similar to decision condition testing, discussed earlier)
- Modified condition decision testing
- Linear Code Sequence and Jump (LCSAJ) testing¹⁸

18. ISTQB no longer requires this particular white-box technique and—partially because we could find nowhere that it is actually performed—we have dropped it from this book.

Rounding out the list are two sections, “Random Testing” and “Other Testing Techniques.” Random testing was not covered in the Foundation syllabus, but we’ll talk about the use of randomness in relation to reliability testing in a later chapter. The section on other testing techniques doesn’t provide any examples but merely talks about rules on how to select them.

You might be thinking, “Hey, wait a minute: that was too fast. Which of those do I need to know for the Advanced Level Technical Test Analyst exam?” The answer is in two parts. First, you need to know any test design technique that was covered in the Foundation syllabus. Such techniques may be covered on the Advanced Level Technical Test Analyst exam. Second, we have covered all the white-box test design techniques that might be on the Advanced Level Test Analyst exam in detail earlier in this chapter.

BS 7925/2 provides one or more coverage metrics for each of the test measurement techniques. These are covered in the measurement part of the standard. The choice of organization for this standard 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-178C. (We touched on that standard when we covered MC/DC coverage earlier.) So, during test design, we’d employ the required 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. To map that process to the ISTQB fundamental test process, we can say the following:

- 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 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 corresponds roughly to the ISTQB's evaluating test criteria and reporting.

Finally, as promised, let's complete our discussion about the DO-178C standard. This standard is promulgated by the United States Federal Aviation Administration (US FAA). As noted, it covers avionics software systems. In Europe, it's called ED-12B. The standard assigns a criticality level, based on the potential impact of a failure. Based on the criticality level, a certain level of white-box test coverage is required, as shown in Table 2–20.

Table 2–20 *FAA DO-178C 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

Let us explain Table 2–20 a bit more thoroughly:

Criticality level A, or Catastrophic, applies when a software failure can result in a catastrophic failure of the system. A catastrophic failure is defined as:

Failure conditions which would result in multiple fatalities, usually with the loss of the airplane.

For software with such criticality, the standard requires modified condition/ decision, decision, and statement coverage. Interestingly, for this level of criticality, structural coverage may only be applied to the source code if it can be shown that the source maps to the compiled object code. We discussed the possibility of the compiled code not having a one-to-one relationship earlier when we discussed path 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. The standard defines this level of criticality as:

Failure conditions which would reduce the capability of the airplane or the ability of the flight crew to cope with adverse operating conditions to the extent that there would be:

- (1) *A large reduction in safety margins or functional capabilities,*
- (2) *Physical distress or excessive workload such that the flight crew cannot be relied upon to perform their tasks accurately or completely, or*
- (3) *Serious or fatal injuries to a relatively small number of the occupants other than the flight crew.*

For software with such criticality, the standard requires decision and statement coverage.

Criticality level C, or Major, applies when a software failure can result in a major failure of the system. The standard defines this as:

Failure conditions which would reduce the capability of the airplane or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a significant reduction in safety margins or functional capabilities, a significant increase in crew workload or in conditions impairing crew efficiency, or discomfort to flight crew, or physical distress to passengers or cabin crew, possibly including injuries.

For software with such criticality, the standard requires only statement coverage.

Criticality level D, or Minor, applies when a software failure can only result in a minor failure of the system. The standard defines this as:

Failure conditions which would not significantly reduce airplane safety, and which would involve crew actions that are well within their capabilities. Minor failure conditions may include, for example, a slight reduction in safety margins or functional capabilities, a slight increase in crew workload, such as, routine flight plan changes, or some physical discomfort to passengers or cabin crew.

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 cannot have an effect on the system. The standard defines this as:

Failure conditions that would have no effect on safety; for example, failure conditions that would not affect the operational capability of the airplane or increase crew workload.

These categories make a certain amount of sense. You should be more concerned about software that affects flight safety, such as rudder and aileron control modules, than about software that doesn't, such as video entertainment systems. However, there is a risk of using a one-dimensional white-box measuring 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. Rest assured that DO-178C does require that requirements-based black-box testing be done in addition to the structural testing we have called out here. It also includes a raft of other QA techniques, including requirements and design reviews, unit testing, and static analysis.

Just to make sure we have hit all of the terminology correctly, the United States FAA DO-178C standard bases the extent of testing—measured in terms of white-box code coverage—on the potential impact of a failure. That makes DO-178C a risk-aware testing standard.

Another interesting example of how risk management, including quality risk management, plays into the engineering of complex and/or safety-critical systems is found in the ISO/IEC standard 61508, which is mentioned in the Advanced syllabus. This standard applies to 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 is very much focused on risks. Risk analysis is required. It considers two primary factors as determining the level of risk: likelihood and impact. During a project, we are 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. Thus we must take defect-preventing actions like requirements, design, and code reviews.

As we were working on the second edition of this book, a news article was published* claiming that hackers could indeed “hack the satellite communications equipment on passenger jets through the WiFi and inflight entertainment systems.” The cyber security researcher, Ruben Santamarta, claims that the entertainment systems are wide open and vulnerable. Representatives for the company that manufactures the satellite equipment claim that such a hacker must have physical access to the equipment, so corruption is unlikely since “...there are strict requirements restricting access to authorized personnel only.” This does go to show, however, that even with the mandated coverage such as shown by Table 2-20, security is rarely as good as we might hope, and technical test analysts need to be ever vigilant for the possibility of a breach.

* <http://www.reuters.com/article/2014/08/04/us-cybersecurity-hackers-airplanes-idUSKBN0G40WQ20140804>

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 specifications, software test results, software safety validation, verification report, and software functional safety report.

The standard addresses the author-bias problem. As discussed in the Foundation syllabus, this is the problem with self-testing, the fact that you bring the same blind spots and bad assumptions to testing your own work that you brought to creating that work. So the standard calls for tester independence, indeed insisting on it for those performing any safety-related tests. And since 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 (or 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 test design techniques we have already discussed in this chapter, others fall under the Advanced Test Analyst purview and are discussed in Rex's book *Advanced Software Testing - Vol. 1: Guide to the ISTQB Advanced Certification as an Advanced Test Analyst*. A few of the topics we will cover in upcoming chapters, such as dynamic analysis, performance testing, static analysis, and test automation.

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 of testing is required, the standard states that it should be documented and independently verified. In other words, the standard can require auditing or outside reviews of testing activities. In addition, continuing on with that theme 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 standard requires the use of structural test design techniques such as discussed in this chapter. Structural coverage requirements are implied, again based on the safety integrity level. (This is similar to DO-178C.) 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.

Structure-Based Testing Exercise

Using the code in Figure 2–18, answer the following questions:

1. How many test cases are needed for basis path coverage?
2. If we wanted to test this module to the level of multiple condition coverage (ignoring the possibility of short-circuiting), how many test cases would we need?
3. If this code were in a system that was subject to FAA/DO178C and was rated at Level A criticality, how many test cases would be needed for the first *if()* statement alone?
4. To achieve only statement coverage, how many test cases would be needed?

```
1. void ObjTree::AddObj(const Obj& w) {
2.     // Make sure we want to store it
3.     if (!(isReq() && isBeq() && (isNort() || (isFlat() && isFreq())))) {
4.         return;
5.     }
6.     // If the tree is currently empty, create a new one
7.     if (root == 0) {
8.         // Add the first obj.
9.         root = new TObjNode(w);
10.    } else {
11.        TObjNode* branch = root;
12.        while (branch != 0) {
13.            Obj CurrentObj = branch->TObjNodeDesig();
14.            if (w < CurrentObj) {
15.                // Obj is new or lies to left of the current node.
16.                if (branch->TObjNodeSubtree(LEFT) == 0) {
17.                    TObjNode* NewObjNode = new TObjNode(w);
18.                    branch->TObjNodeAddSubtree(LEFT, NewObjNode);
19.                    break;
20.                } else {
21.                    branch = branch->TObjNodeSubtree(LEFT);
22.                }
23.            } else if (CurrentObj < w) {
24.                // Obj is new or lies to right of the current node.
25.                if (branch->TObjNodeSubtree(RIGHT) == 0) {
26.                    TObjNode* NewObjNode = new TObjNode(w);
27.                    branch->TObjNodeAddSubtree(RIGHT, NewObjNode);
28.                    break;
29.                } else {
30.                    branch = branch->TObjNodeSubtree(RIGHT);
31.                }
32.            } else {
33.                // Found match, so bump the counter and end the loop.
34.                branch->TObjNodeCountIncr();
35.                break;
36.            }
37.        } // while
38.    } // if
39.    return;
40. }
```

Figure 2–18 Code for structure-based exercise

2.8.1 Structure-Based Testing Exercise Debrief

1. How many test cases are needed for basis path coverage?

The control graph should appear as shown in Figure 2–19.

The number of test cases we need for cyclomatic complexity can be calculated in three different ways. Well, maybe. The region method clearly is problematic because of the way the graph is drawn. This is a common problem when drawing directed control flow graphs; there is a real art to it, especially when working in a timed atmosphere.

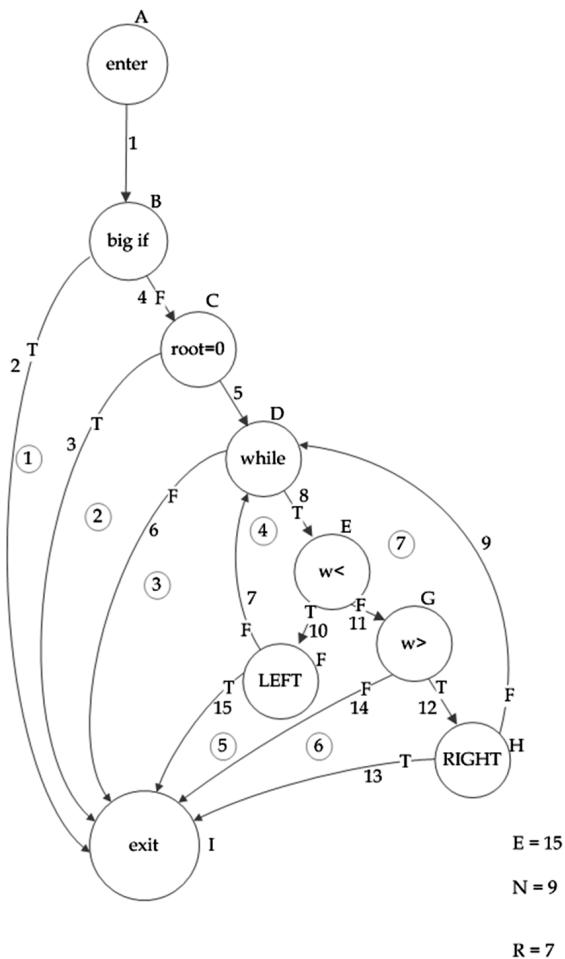


Figure 2–19 Directed control flow graph

Let's try McCabe's cyclomatic complexity formula:

$$C = E - N + 2$$

$$C = 15 - 9 + 2$$

$$C = 8$$

Alternately, we could try the rule of thumb method for finding cyclomatic complexity: count the decisions and add one. There are decisions made in the following lines:

1. Line 3: *if*
2. Line 7: *if* (remember, the *else* in line 10 is not a decision)
3. Line 12: *while*
4. Line 14: *if*
5. Line 16: *if*
6. Line 23: *else if* (this is really just an *if*—this is often called a *ladder if* construct)
7. Line 25: *if*

Therefore, the calculation is as follows:

$$C = 7 + 1$$

$$C = 8$$

All this means we have eight different test cases, as follows:

1. ABI
2. ABCI
3. ABCDI
4. ABCDEFI
5. ABCDEGI
6. ABCDEGHI
7. ABCDEFDI
8. ABCDEGHDI

At this point, you'll need to develop test cases that will force execution through those paths. At this point in the process of developing tests from McCabe's technique, you can often find that some paths are not achievable. For example, path ABCDI, in which the body of the *while()* loop is never executed, should be impossible, since the value of branch is initialized to root, which we already

know is not equal to zero at this point in the code. You can try to use a debugger to force the flow to occur, but no ordinary set of input values will work.

2. If we wanted to test this module to the level of multiple condition coverage (ignoring the possibility of short-circuiting), how many test cases would we need?

The first conditional statement looks like this:

```
!(isReq() && isBeq() && (isNort() || (isFlat() && isFreq()))))
```

Each of these functions would likely be a private method of the class we are working with (`ObjTree`). We can rewrite the conditional to make it easier to understand.

```
!(A && B && (C || (D && E)))
```

A good way to start is to come up with a truth table that covers all the possibilities that the atomic conditions can take on. With five atomic conditions, there are 32 possible combinations, as shown in Table 2–21. Of course, you could just calculate that by calculating 2^5 . But since we want to discuss how many test cases we would need if this code was written in a language that did short-circuit Boolean evaluations (it is!), we'll show it here.

Table 2–21 Truth table for five atomic conditions

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F		
B	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	
C	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	T	T	T	T	F	F	F	F	T	T	T	F	F	F			
D	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	F			
E	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	F			

Thirty-two separate test cases would be needed. Note that 16 of these test cases evaluate to TRUE. This evaluation would then be negated (see the negation operator in the code). So 16 test cases would survive and move to line 7 of the code. We would still need to achieve decision coverage on the other conditionals in the code (more on that later).

3. If this code were in a system that was subject to FAA/DO178C and was rated at Level A criticality, how many test cases would be needed for the first *if()* statement alone?

The standard that we must meet mentions five different levels of criticality.

Table 2-22 FAA/DO178C software test levels

Criticality	Required Coverage
Level A: Catastrophic	Modified Condition/Decision, Decision and Statement
Level B: Hazardous/Severe	Decision and Statement
Level C: Major	Statement
Level D: Minor	None
Level E: no effect	None

This means that we must achieve MC/DC level coverage for the first *if()* statement. Note that there are no other compound conditional statements in the other decisions, so we can ignore them; decision coverage will cover them.

In order to achieve MC/DC coverage, we must ensure the following:

- A. Each atomic condition is evaluated both ways (T/F).
- B. Decision coverage must be satisfied.
- C. Each atomic condition must be able to affect the outcome independently while other atomic conditions are held without changing.

The expression we are concerned with follows:

$(! (A \&& B \&& (C || (D \&& E)))$

We will start out by ignoring the first negation, shown by the exclamation mark at the beginning. We don't know about you, but inversion logic always gives us a headache. Since we are inverting the entire expression (check out the parentheses to see that), we can just look at the expression. First, let's figure out how to make sure each atomic condition can affect the outcome independently. We will create a template table showing the individual atomic conditions and then work with that to identify neutral values for the tests.

Table 2-23 Template for MC/DC test creation

$(A \ \&\& B \ \&\& (C \ \ (D \ \&\& E))$	T	F
A	T - - - -	F - - - -
B	- T - - -	- F - - -
C	- - T - -	- - F - -
D	- - - T -	- - - F -
E	- - - - T	- - - - F

Let's find the neutral values we will use.

$$(A \ \&\& B \ \&\& (C \ || \ (D \ \&\& E)))$$

A: A is ANDed with B, therefore its neutral value is T.

B: B is ANDed with A, therefore its neutral value is T.

C: C is ORed with $(D \ \&\& E)$, therefore its neutral value is F.

D: D is ANDed with E, therefore its neutral value is T.

E: E is ANDed with D, therefore its neutral value is T.

Note that when we want $(C \ || \ (D \ \&\& E))$ to equal T, we can use the following combinations:

T F F
 T T T
 T T F
 T F T
 F T T

Likewise, if we want to make it equal to F, we can use the following combinations:

F T F
 F F T
 F F F

We may need to substitute these variations to get down to the theoretical minimum number of tests which is 6 ($n+1$). Substituting these values into Table 2-23 gives us Table 2-24.

Table 2-24 Test values for MC/DC

$(A \&& B \&& (C (D \&& E)))$	T	F
A	T T T F F	F T T F F
B	T T T F F	T F T F F
C	T T T F F	T T F T F
D	T T F T T	T T F F T
E	T T F T T	T T F T F

It took us less than 4 minutes to come up with these values. To double check:

- Cell (2,2) → T. But if A is changed to F → F
- Cell (2,3) → F. But if A is changed to T → T
- Cell (3,2) → T. But if B is changed to F → F
- Cell (3,3) → F. But if B is changed to T → T
- Cell (4,2) → T. But if C is changed to F → F
- Cell (4,3) → F. But if C is changed to T → T
- Cell (5,2) → T. But if D is changed to F → F
- Cell (5,3) → F. But if D is changed to T → T
- Cell (6,2) → T. But if E is changed to F → F
- Cell (6,3) → F. But if E is changed to T → T

4. To achieve only statement coverage, how many test cases would be needed?

Okay, so this one is relatively simple! You could trace through the code and determine the number. Or, you could notice that the way it is written, statement coverage is going to be the same as decision coverage. How do we know that?

The first *if()* (line 3) needs to be tested both ways to get statement coverage. When TRUE, it does a quick return, which must be executed. When FALSE, we get to go further in the code.

The second *if()* (line 7), third (line 14), fourth (line 16), and sixth (line 25) each has a statement in both the TRUE and FALSE directions.

The *else if()* (line 23) is the *else* for the *if()* (line 14) and has an *else* of its own, both of which have code that needs to execute for statement coverage.

The only confusing piece to this answer is the *while()* (line 12). The way the code is written, it should not allow it to ever evaluate to FALSE. That means we would have to use a debugger to change the value of *branch* to test it.

Since we have already gone through getting decision coverage in answer 3, we will accept that answer.

2.9 A Final Word on Structural Testing

Before we leave structural testing, Boris Beizer wrote a sharp argument seemingly *against* doing white-box structure-based testing in his book *Software Testing Techniques*; we used his second edition, published in 1990. We think that he can express it better than we can, so it is included here:

Path testing is more effective for unstructured rather than structured code.

Statistics indicate that path testing by itself has limited effectiveness for the following reasons:

- *Planning to cover does not mean you will cover—especially when there are bugs contained.*
- *It does not show totally wrong or missing functionality*
- *Interface errors between modules will not show up in unit testing*
- *Database and data-flow errors may not be caught*
- *Incorrect interaction with other modules will not be caught in unit testing*
- *Not all initialization errors can be caught by control flow testing*
- *Requirements and specification errors will not be caught in unit testing.*

After going through all of those problems with structural testing, Beizer goes on to say:

Creating the flowgraph, selecting a set of covering paths, finding input data values to force those paths, setting up the loop cases and combinations—it's a lot of work. Perhaps as much work as it took to design the routine and certainly more work than it took to code it. The statistics indicate that you will spend half your time testing it and debugging—presumably that time includes the time required to design and document test cases. I would rather spend a few quiet hours in my office doing test design than twice those hours on the test floor debugging, going half deaf from the clatter of a high-speed printer that's producing massive dumps, the reading of which will make me half blind. Furthermore, the act of careful, complete, systematic test design will catch as many bugs as the act of testing....The test design process, at all levels, is at least as effective at catching bugs as is running the test designed by that process.

2.10 Sample Exam Questions

1. Which of the following statements correctly represents the relative strength of different structural levels of testing?
 - A. 100 percent condition coverage guarantees 100 percent statement coverage.
 - B. 100 percent statement coverage guarantees 100 percent branch coverage but not 100 percent decision coverage.
 - C. 100 percent condition coverage does not guarantee 100 percent branch coverage.
 - D. 100 percent decision condition coverage does not guarantee 100 percent statement coverage.
2. Given the following predicate, which test would achieve decision condition coverage for this construct of code with a minimum number of tests? Assume a test contains 3 factors—values for A, B, and C. T == TRUE, F == FALSE.

```
while ((A || B) && (C < 100)) {  
    D++  
}
```

 - A. (T,T,98); (F,T,101); (T,F,100)
 - B. (T,T,99); (F,F,0)
 - C. (T,F,579); (F,T,-63)
 - D. (T,T,0); (F,F,50); (T,F,100)

3. Using the following neutral table, select which set of tests gives MC/DC level coverage for the illustrated predicate. Assume no short-circuiting by the compiler.

(A AND B) OR (C AND D)	T	F
A	T -- -- --	F -- -- --
B	-- T -- --	-- F -- --
C	-- -- T --	-- -- F --
D	-- -- -- T	-- -- -- F

- A. (T,T,F,F)(F,F,T,T)(F,T,F,T)(T,F,F,F)(F,F,T,F)
 B. (T,T,T,T)(F,F,F,F)(T,T,FF)(F,F,T,T)
 C. (T,T,F,F)(T,F,T,T)(F,T,F,T)(T,T,T,F)(T,T,F,T)
 D. (F,T,F,T)(F,T,F,F)(F,T,T,T)(T,F,T,F)
4. You have been asked to test your C code to either MC/DC or multiple condition-level coverage. You have a specific piece of code, shown below. How many test cases would it take to test to MC/DC-level coverage? How many would it take to test to multiple condition-level coverage? Assume no short-circuiting by the compiler.

```
if ((!(a&&b)) || (c > d)) {
    z = func(a,b,c,d);
}
else {
    z = func(b,a,d,c);
```

- A. MC/DC: 5, multiple condition: 16
 B. MC/DC: 4, multiple condition: 8
 C. MC/DC: 3, multiple condition: 4
 D. MC/DC: 5, multiple condition: 8

5. The following C code function will allow a browser to connect to a given website.

```
#include<windows.h>
#include<wininet.h>
#include<stdio.h>
int main()
{
    HINTERNET Initialize, Connection, File;
    DWORD dwBytes;
    char ch;
    Connection = InternetConnect(Initialize, "www.xxx.com",
        INTERNET_DEFAULT_HTTP_PORT, NULL, NULL,
        INTERNET_SERVICE_HTTP, 0, 0);

    File = HttpOpenRequest(Connection, NULL, "/index.html",
        NULL, NULL, NULL, 0, 0);

    if (HttpSendRequest(File, NULL, 0, NULL, 0))
    {
        while (InternetReadFile(File, &ch, 1, &dwBytes))
        {
            if (dwBytes != 1) break;
            putchar(ch);
        }
    }
    InternetCloseHandle(File);
    InternetCloseHandle(Connection);
    InternetCloseHandle(Initialize);
    return 0;
}
```

What is the minimum number of test cases that would be required to achieve statement coverage for this code?

- A. 1
- B. 2
- C. 4
- D. 6

6. Given the following snippet of code, which of the following values for the variable Counter will give loop coverage with the fewest test cases?

```
...
for (i=0; i<=Counter; i++) {
    Execute some statements;
}
```

- A. (-13, 0,1,795)
 - B. (-1,0,1)
 - C. (0,1,1000)
 - D. (-7,0,500)
7. In a module of code you are testing, you are presented with the following *if()* statement. How many different test cases would you need to achieve multiple condition coverage (assume no short-circuiting by the compiler).

```
if (A && B || (Delta < 1) && (Up < Down) || (Right >= Left)) {
    Execute some statements;
}
Else {
    Execute some statements;
}
```

- A. 24
- B. 32
- C. 48
- D. 128

8. The following code snippet reads through a file and determines whether the numbers contained are prime or not.

Table 2–25 Code snippet

```
1   Read (Val);
2   While NOT End of File Do
3       Prime := TRUE;
4       For Holder := 2 TO Val DIV 2 Do
5           If Val - (Val DIV Holder)*Holder= 0 Then
6               Write (Holder, ` is a factor of', Val);
7               Prime := FALSE;
8           Endif;
9       Endfor;
10      If Prime = TRUE Then
11          Write (Val , ` is prime');
12      Endif;
13      Read (Val);
14  Endwhile;
15  Write('End of run')
```

Calculate the cyclomatic complexity of the code.

- A. 3
- B. 5
- C. 7
- D. 9

9. Given Beizer's theories of path coverage, the flow below (Figure 2–20), and the tests already listed, which of the following paths through the code would best complete the set of tests required?

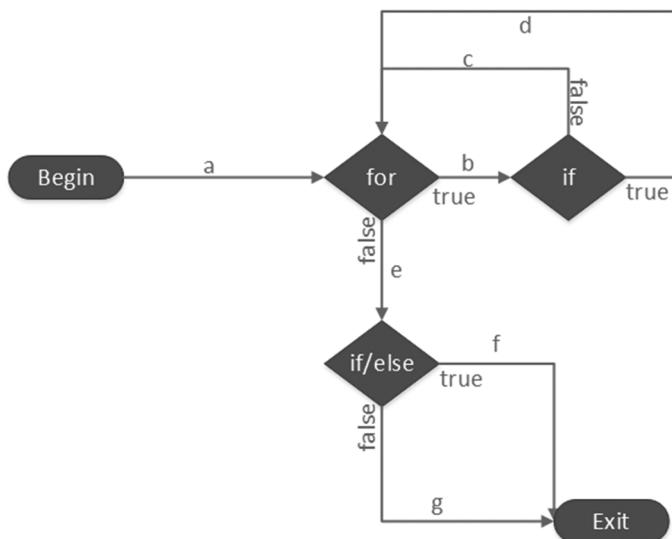


Figure 2–20 Flow chart

Test case 1: a,e,g

Test case 2: a,b,d,e,g

Test case 3: a,b,d,e,f

A. a,b,c,b,d,e,f

B. a,e,f

C. a,b,c,b,c,e,g

D. a,b,c,e,f

10. Which of the following API quality risks is primarily due to loose coupling?
 - A. Security risks
 - B. Lost transactions
 - C. Low availability
 - D. Incorrect parameter marshalling

11. You are testing code that controls the radar system in a Boeing 777. The code is part of the completely redundant backup system and hence is judged to be Level B criticality according to standard ED-12B. To which two of the following levels of structural coverage would it be necessary to test this code?
 - A. 100 percent MC/DC coverage
 - B. 100 percent branch coverage
 - C. 100 percent statement coverage
 - D. 100 percent condition decision coverage
 - E. Multiple condition coverage

3 Analytical Techniques

No product of human intellect comes out right the first time. We rewrite sentences, rip out knitting stitches, replant gardens, remodel houses, and repair bridges. Why should software be any different?

—Ruth Wiener

This third chapter of the Advanced Level Syllabus – Technical Test Analyst is concerned with analytical techniques (which includes both static analysis and dynamic analysis). This chapter contains three sections:

1. Introduction
2. Static Analysis
3. Dynamic Analysis

3.1 Introduction

Learning objectives

Recall of content only.

In the last chapter, you saw a number of different white-box (structure-based) ways to detect defects, all of which require the tester to execute code to force failures to show themselves. In this chapter, we will discuss several analytical techniques that can expose defects in different ways. The first of these is called static analysis.

We examined this topic briefly at the Foundation level; how to apply tools to find defects, anomalies, standards violations, and a whole host of maintainability issues without executing a line of the code. In the Advanced syllabus, ISTQB

opens up the term to apply to more than just tool use. The definition of static analysis, as given in the Advanced Syllabus, is:

Static analysis encompasses the analytical testing that can occur without executing the software. Because the software is not executing, it is examined either by a tool or by a person to determine if it will process correctly when it is executed. This static view of the software allows detailed analysis without having to create the data and preconditions that would cause the scenario to be exercised.

Contrast that with the “official definition” found in the latest ISTQB glossary:

Analysis of software development artifacts, e.g., requirements or code, carried out without execution of these software development artifacts. Static analysis is usually carried out by means of a supporting tool.

It may be that the glossary definition is a bit too restricting. While tools are likely to be used most of the time, there certainly could be times when we do some static analysis manually.

We will examine such topics as control flow analysis, data flow analysis, compliance to standards, certain code maintainability improvement techniques, and call graphing.

Finally, in this chapter, we will discuss dynamic analysis, wherein we utilize tools while executing system code to help us find a range of failures that we might otherwise miss.

Remember to review the benefits of reviews and static analysis from the Foundation; we will try not to cover the same materials. In Chapter 5 we will further discuss reviews.

3.2 Static Analysis

3.2.1 Control Flow Analysis

Learning objectives

TTA-3.2.1 (K3) Use control flow analysis to detect if code has any control flow anomalies.

We discussed control flow testing in the last chapter, mainly as a way to determine structural coverage. One of the ways we discussed control flow graphs was in reference to path testing. In that section, we started looking at cyclomatic

ISTQB Glossary

control flow analysis: A form of static analysis based on a representation of unique paths (sequences of events) in the execution through a component or system. Control flow analysis evaluates the integrity of control flow structures, looking for possible control flow anomalies such as closed loops or logically unreachable process steps.

cyclomatic complexity: The maximum number of linear, independent paths through a program. Cyclomatic complexity may be computed as: $L - N + 2P$, where

- L = the number of edges/links in a graph
- N = the number of nodes in a graph
- P = the number of disconnected parts of the graph
(e.g., a called graph or subroutine)

static analysis: Analysis of software development artifacts, e.g., requirements or code, carried out without execution of these software development artifacts. Static analysis is usually carried out by means of a supporting tool.

complexity and Thomas McCabe's research into what complexity means to software testing.

It is now common wisdom that the more complex the code is, the more likely it is to have hidden defects. This was not always considered, however. In his original 1976 paper, Thomas McCabe cited the (then new) practice of limiting the physical size of programs through modularization. A common technique he cited was writing a 50-line program that consisted of 25 consecutive *if-then* statements. He pointed out that such a program could have as many as 33.5 million distinct control paths, few of which were likely to get tested. He presented several examples of modules that were poorly structured having cyclomatic complexity values ranging from 16 to 64 and argued that such modules tended to be quite buggy. Then he gave examples of developers who consistently wrote low complexity modules—in the 3 to 7 cyclomatic complexity range—who regularly had far lower defect density.

In his paper, McCabe pointed out an anomaly with cyclomatic complexity analysis that we should discuss here.

```
1.     jmp_buf sjbuf;
2.     unsigned long int hexnum;
3.     unsigned long int nhex;
4.
5.     main()
6.     /* Classify and count input chars */
7.     {
8.         int c, gotnum;
9.         void pophdigit();
10.
11.        hexnum = nhex = 0;
12.
13.        if (signal(SIGINT, SIG_IGN) != SIG_IGN) {
14.            signal(SIGINT, pophdigit);
15.            setjmp(sjbuf);
16.        }
17.        while ((c = getchar()) != EOF) {
18.            switch (c) {
19.                case '0':case '1':case '2':case '3':case '4':
20.                case '5':case '6':case '7':case '8':case '9':
21.                    /* Convert a decimal digit */
22.                    nhex++;
23.                    hexnum *= 0x10;
24.                    hexnum += (c - '0');
25.                    break;
26.                case 'a': case 'b': case 'c':
27.                case 'd': case 'e': case 'f':
28.                    /* Convert a lower case hex digit */
29.                    nhex++;
30.                    hexnum *= 0x10;
31.                    hexnum += (c - 'a' + 0xa);
32.                    break;
33.                case 'A': case 'B': case 'C':
34.                case 'D': case 'E': case 'F':
35.                    /* Convert an upper case hex digit */
36.                    nhex++;
37.                    hexnum *= 0x10;
38.                    hexnum += (c - 'A' + 0xA);
39.                    break;
40.                default:
41.                    /* Skip any non-hex characters */
42.                    break;
43.            }
44.        }
45.        if (nhex == 0) {
46.            fprintf(stderr, "hxcvt: no hex digits to convert!\n");
47.        } else {
48.            printf("Got %d hex digits: %x\n", nhex, hexnum);
49.        }
50.
51.        return 0;
52.    }
53.    void pophdigit()
54.    /* Pop the last hex input out of hexnum if interrupted */
55.    {
56.        signal(SIGINT, pophdigit);
57.        hexnum /= 0x10;
58.        nhex--;
59.        longjmp(sjbuf, 0);
60.    }
61.
```

Figure 3–1 Hexadecimal converter code

Refer to Figure 3–1, our old friend the hexadecimal converter code, which we introduced in Chapter 2. Following Boris Beizer's suggestions, we created a control flow graph, which we re-create here in Figure 3–2.

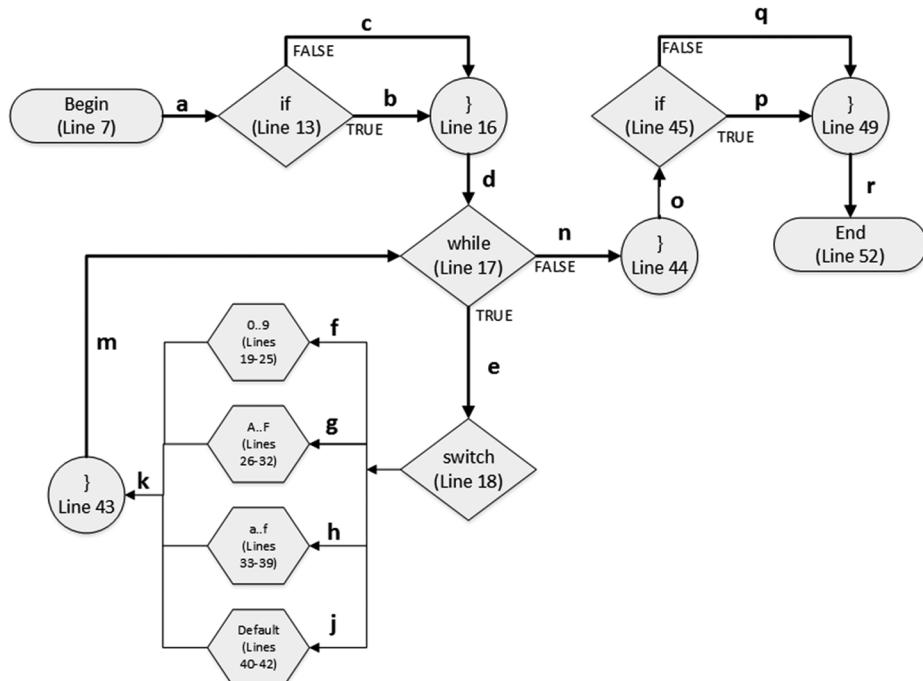


Figure 3–2 Control flow graph (following Beizer)

Let's look at a different kind of flow graph of that same code to analyze its cyclomatic complexity. See Figure 3–3.

Once we have the graph, we can calculate the cyclomatic complexity for the code.

There are 9 nodes (bubbles) and 14 edges (arrows). So using the formula we discussed in Chapter 2:

$$E - N + 2 \rightarrow 14 - 9 + 2 \rightarrow 7$$

That is, using McCabe's cyclomatic complexity analysis to determine the basis set of tests we need for minimal testing of this module, we come up with seven tests. Those seven tests will guarantee us 100 percent of both statement and decision coverage.

Hmmmmmm.

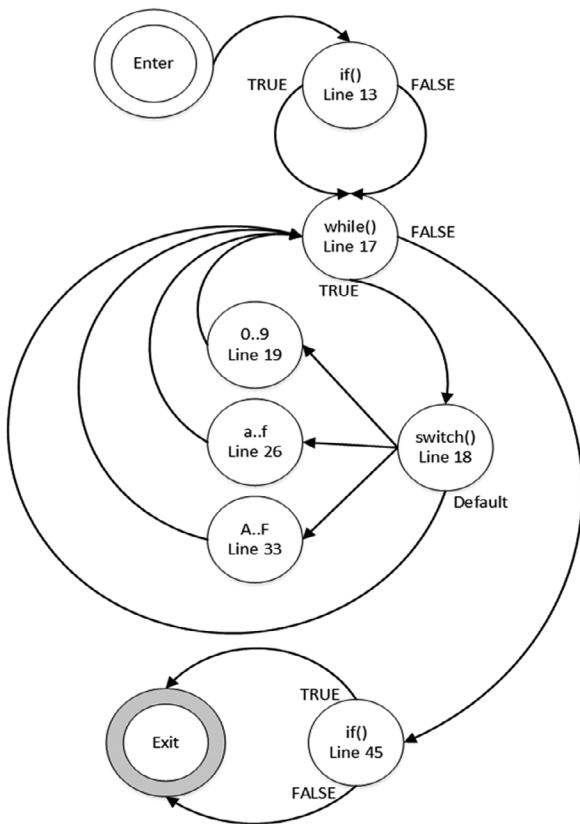


Figure 3–3 Cyclomatic flow graph for hex converter code

If you look back at Beizer's method of path analysis in Chapter 2, we seemed to show that we could achieve 100 percent statement and decision coverage with five tests. Why should a McCabe cyclomatic directed control graph need more tests to achieve 100 percent branch coverage than one suggested by Beizer?

It turns out that the complexity of switch/case statements can play havoc with complexity discussions. So we need to extend our discussion of complexity. McCabe actually noted this problem in his original paper:

The only situation in which this limit (cyclomatic complexity limit of 10) has seemed unreasonable is when a large number of independent cases followed a selection function (a large case statement), which was allowed.¹

1. "A Complexity Measure," Thomas J McCabe, *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976, <http://www.literateprogramming.com/mccabe.pdf>

What McCabe said is that, for reasons too deep to discuss here, switch/case statements maybe should get special dispensation when dealing with complexity. So, let's look at switch statements and see why they are special.

The code we have been using for our hexadecimal converter code (Figure 3–1) contains the following switch statement (we have removed the non-interesting linear code):

```
switch (c) {
    case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9':
        xxxxx;
        break;
    case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':
        xxxxx;
        break;
    case 'A': case 'B': case 'C': case 'D': case 'E': case 'F':
        xxxxx;
        break;
    default:
        break;
}
```

Because of the way this code is written, there are four different paths through the switch statement, as follows:

One path if the inputted character is a digit ('0'..'9')

One path if the inputted character is an uppercase hex-legal character ('A'..'F')

One path if the inputted character is a lowercase legal hex character ('a'..'f')

One path if the inputted character is any other legal (but non-hex) character

We certainly could rewrite this code in a myriad of ways, each of which might appear to have a different number of paths through it. For example, in Delphi, we might use the characteristic of sets to make it easier to understand:

```
if (c IN ('0'..'9')) then begin
    xxxx
end;
else if (c IN ('A'..'F')) then begin
    xxxx
end;
else if (c IN ('a'..'f')) then begin
    xxxx
end;
```

This code appears to only have three paths through it, but of course that is illusory. We can take any one of the three paths, or we can still take the fourth by not having a legal hex value.

If the programmer was a novice, they might use an *if/else* ladder, which would get very ugly with a huge cyclomatic complexity hit:

```
if (c=='0') {  
    xxxxx;  
} else if (c=='1'):  
    xxxxx;  
} else if (c=='2'):  
    xxxxx;  
}  
etc.
```

This particular mess would add 22 new paths to the rest of the module. Of course, any real programmer who saw this mess in a review would grab the miscreant by the suspenders and instruct them on good coding practices, right?

Interestingly enough, with the way the original code was written

```
case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8':  
case '9'...  
case 'a': case 'b': case 'c': case 'd': case 'e': case 'f'...  
case 'A': case 'B': case 'C': case 'D': case 'E': case 'F'...
```

there is a really good chance that the compiler will actually generate a unique path for every “case” token in each string anyway. That means that the source code would be deemed to have four paths, but the generated object code would still contain 22—or more—unique paths through it. The Delphi example would likely generate object code for only the four paths.

It would also be possible to use a complex data structure such as a hash or dictionary to represent this code rather than using the switch statement. That option would hide the complexity in an external call to the data-structure code making the choice.

From the preceding discussion, we can see that there are many different ways to code this specific functionality. All of those coding styles can have different cyclomatic complexity measures. But at the end of the day, all of the different ways of writing this code are going to take just about the same effort to test. This would appear to mean that cyclomatic complexity might not be the

best measure to determine the amount of effort it actually takes to test a module.

So, is excessive cyclomatic complexity truly the enemy? In Chapter 2 we cited a NIST study that seemed to imply that.

We must understand that there are actually two different ways that we can look at complexity. One is the way we have been looking at it: how many tests do we need to get adequate coverage of the structure of the code? This conversation has much to do with the problems built into the design of the human brain.

In 1956, George A. Miller, a cognitive psychologist, wrote a paper called “The Magical Number Seven, Plus or Minus Two.”² In it he discussed the limits of human memory and specifically referred to short-term memory. While many of the individual details have been challenged over the years, it does appear that most people have problems keeping more than seven or eight chunks of information, sometimes called a working set, in their short-term memory for any appreciable time.

How does this affect our discussion of cyclomatic complexity? Our human brains appear to be spectacularly unsuited to keeping many complex concepts in our head at once. We tend to forget some chunks when we try to input too many others. Programmers have long recognized this concept and even have an acronym for it: KISS (keep it simple, stupid). The more complex the code in a module, the more stuff a test analyst needs to keep in their head when designing test cases. A cyclomatic complexity of 10 is about the outside limit for a tester to be able to grasp all of the nuances of the code that they want to test. We are not saying this is a definitive hard limit; just a common-sense rule of thumb (we will discuss this in more detail later).

The second way to look at complexity in software is much more intricate; McCabe called it *essential complexity*. Beyond the basis number of tests we need to test a module to a particular level, what is the reliability, maintainability, understandability, and so on of the module? Essential complexity looks for coding techniques that are not well structured; examples include entering or breaking out of a loop at more than one place, having multiple return statements in the module, and throwing an exception inside a function.

Certain programming languages can make testing much harder based on their features. For example, the programming language Scheme actually has—as

2. <http://www.musanim.com/miller1956/> (This link was selected because there are several different language translations here.)

a feature—the ability to traverse nondeterministic paths (via use of the operator *amb*) to facilitate searching problem spaces without having to program the search directly. Testing such code, however, becomes much more difficult because there is no guarantee which path is being traversed when.

A simplified definition we can use is that essential complexity measures how much complexity we have left after we have removed the well-structured basis paths. According to McCabe, we can work through the structured paths of a module (i.e., those paths that have a single entry and single exit point) and essentially ignore their complexity (since they could be replaced with a call to an external subroutine). All such structured paths in the module now are seen to have an essential complexity of one. Note that a switch/case statement has a single entry and exit, hence McCabe's dismissal of it as contributing to cyclomatic complexity more than any if/else statement.

At this point, any paths not yet counted are, by definition, nonstructured. Each nonstructured path then adds to the essential complexity value.

Sometimes, unstructured code makes a lot of sense to the understandability of the code. For example, suppose we are analyzing a function that accepts arguments when called. If you were to code the module such that the first thing you do is check to make sure each argument is correct—and, if not, immediately return an error to the caller—this would be considered unstructured. But, in our opinion, that extra complexity would actually make the code clearer.

The point of this discussion is to illustrate that cyclomatic complexity is not the be-all and end-all of how difficult it is to test a module. If the essential complexity is low (i.e., the code is well structured), it will be relative easy to refactor into smaller modules that have lower cyclomatic complexity. If, on the other hand, the essential complexity is high, the module will be harder to refactor successfully and will likely be less reliable and harder to maintain and understand.

Is this an argument for using well-structured code whenever possible? The way we look at it, if you smell smoke, you don't have to wait to actually see the flames before getting worried. We believe that there is likely enough evidence for us to err on the conservative side and try to avoid *unnecessary* complexity. There may be certain techniques—such as noted above—where *slightly* unstructured code might make sense.

Of course, in software engineering, there are no free lunches. Every decision we make will have trade-offs, some of which may be undesirable. As someone who once wrote operating system code, Jamie can attest that sometimes high complexity is essential. When his organization tried to write some critical OS modules using a low-complexity approach, the trade-off was that the speed

of execution was too slow—by a factor of 10. When they got rid of the nice structure and just wrote highly complex, tightly coupled code, it executed fast enough for their needs. To be sure, maintenance was highly problematic; it took much longer to bring the code to a high level of quality than it took for earlier, less complex modules they had written. As a wise man once said, “Ya pays your money and ya takes your choice.”

McCabe’s original paper suggested that 10 was a “reasonable but not magic upper limit” for cyclomatic complexity. Over the years, a lot of research has been done on acceptable levels of cyclomatic complexity; there appears to be general agreement with McCabe’s recommended limit of 10. The National Institute of Standards and Technology (NIST) agreed with this complexity upper end, although it noted that certain modules should be permitted to reach an upper end of 15, as follows:

The precise number to use as a limit, however, remains somewhat controversial. The original limit of 10 as proposed by McCabe has significant supporting evidence, but limits as high as 15 have been used successfully as well. Limits over 10 should be reserved for projects that have several operational advantages over typical projects, for example experienced staff, formal design, a modern programming language, structured programming, code walkthroughs, and a comprehensive test plan. In other words, an organization can pick a complexity limit greater than 10, but only if it is sure it knows what it's doing and is willing to devote the additional testing effort required by more complex modules.³

To make this analysis worthwhile, every organization that is worried about complexity should also measure the essential complexity and limit that also.

To complete this discussion on complexity we will look at the output of an automation tool that measures complexity, a much more realistic way to deal with it—instead of using control flow graphs.

The cyclomatic complexity graphs in Figure 3–4 come from the McCabe IQ tool, showing the difference between simple and complex code. Remember that code that is very complex is not necessarily wrong, any more than simple code is always correct. But all things being equal, if we can make our code less complex, we will likely have fewer bugs and certainly should gain higher maintainability. By graphically seeing exactly which regions are the most complex, we

3. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*, NIST Special Publication 500-235, Arthur H. Watson, and Thomas McCabe, <http://www.mccabe.com/pdf/mccabe-nist235r.pdf>

can focus both testing and reengineering on the modules that are most likely to need it. We could conceivably have spent the time to map this out manually; however, tools are at their most useful when they allow us to be more productive by automating the low-level work for us.

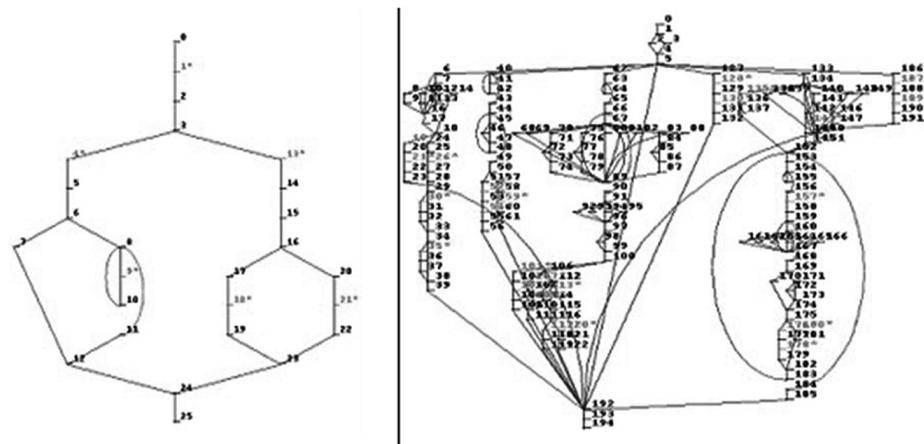


Figure 3-4 Simple vs. complex graphs

Figure 3–4 was a close-up view of two different modules; Figure 3–5 is a more expansive view of our system. This image shows the modules for a section of the system we are testing, both for how they are interconnected and at individual module complexity. Green modules here have low complexity ratings, yellow are considered somewhat complex, and red means highly complex. A highly complex module as a “leaf” node, not interconnected with other modules, might be of less immediate concern than a somewhat complex module in the middle of everything. Without such a static analysis tool to help do the complexity analysis, however, we have very little in the way of options to compare where we should put more time and engineering effort.

While these tools tend to be somewhat pricey, their cost should be compared with the price tag of releasing a system that might collapse under real usage.

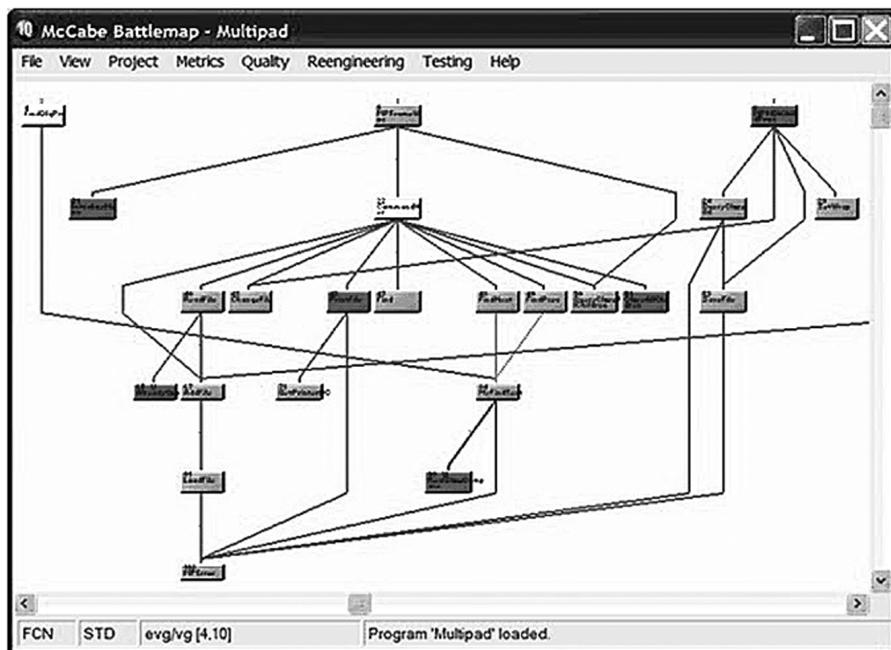


Figure 3-5 Module complexity view

3.2.2 Data Flow Analysis

Learning objectives

TTA-3.2.2 (K3) Use data flow analysis to detect if code has any data flow anomalies.

Our next static analysis topic is data flow analysis; this covers a variety of techniques for gathering information about the possible set of values that data can take during the execution of the system. While control flow analysis is concerned with the paths that an execution thread may take through a code module, data flow analysis is about the life cycle of the data itself.

If we consider that a program is designed to create, set, read, evaluate (and make decisions on), and destroy data, then we must consider the errors that could occur during those processes.

Possible errors include performing the correct action on a data variable at the wrong time in its life cycle, doing the wrong thing at the right time, or the trifecta, doing the wrong thing to the wrong data at the wrong time.

ISTQB Glossary

data flow analysis: A form of static analysis based on the definition and usage of variables.

data flow testing: A white-box test design technique in which test cases are designed to execute definition and use pairs of variables.

definition-use pair: The association of a definition of a variable with the subsequent use of that variable. Variable uses include computational (e.g., multiplication) or to direct the execution of a path ("predicate" use).

In Boris Beizer's book *Software Testing Techniques*, when discussing why we might want to perform data flow testing, he quoted an even earlier work as follows:

It is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, one should not feel confident about a program without having seen the effect of using the value produced by each and every computation.⁴

Here are some examples of data flow errors:

- Assigning an incorrect or invalid value to a variable. These kinds of errors include data-type conversion issues where the compiler allows a conversion but there are side effects that are undesirable.
- Incorrect input results in the assignment of invalid values.
- Failure to define a variable before using its value elsewhere.
- Incorrect path taken due to the incorrect or unexpected value used in a control predicate.
- Trying to use a variable after it is destroyed or out of scope.
- Setting a variable and then redefining it before it is used.
- Side effects of changing a value when the scope is not fully understood. For example, a global or static variable's change may cause ripples to other processes or modules in the system.
- Treating an argument variable passed in by value as if it were passed in by reference (or vice versa).

4. The earlier work was a paper by S. Rapps and E. J. Weyuker named "Data flow analysis techniques for test data selection." Sixth International Conference on Software Engineering, Tokyo, Japan. September 13–16, 1982.

Many data flow issues are related to the programming languages being used.

Since some languages allow a programmer to implicitly declare variables simply by naming and using them, a misspelling might cause a subtle bug in otherwise solid code. Other languages use very strong data typing, where each variable must be explicitly declared before use, but then allow the programmer to “cast” the variable to another data type assuming that the programmer knows what they are doing (sometimes a dodgy assumption, we testers think).

Different languages have data scoping rules that can interact in very subtle ways. Jamie often finds himself making mistakes in C++ because of its scoping rules. A variable may be declared global or local, static or stack-based; it could even be specified that the variables be kept in registers to increase computation speed. A good rule of thumb for testers is to remember that, when giving power to the programmer by having special ways of dealing with data, they will sometimes make mistakes.

Complex languages also tend to have ‘gotchas.’ For example, C and C++ have two different operators that look much the same. The single equal sign (=) is an assignment operator, while the double equal sign (==) is a Boolean operator. When it’s used in a Boolean expression, you would expect that the equal-equal (“is equivalent to” operator) sign would be legal and the single equal sign (assignment operator) would not be. But, the output of an assignment, for some arcane reason, evaluates to a Boolean TRUE. So it is really easy to change the value of a variable unexpectedly by writing ($X = 7$) when the programmer meant ($X == 7$). As previously mentioned, this particular bug is a really good reason to perform static analysis using tools.

The fact is that not all data anomalies are defects. Clever programmers often do exotic things, and sometimes there are even good reasons to do them. Written in a certain way, the code may execute faster. A good technical test analyst should be able to investigate the way data is used, no matter how good the programmer; even great programmers generate bugs.

Unfortunately, as we shall see, data flow analysis is not a universal remedy for all of the ways defects can occur. Sometimes the static code will not contain enough information to determine whether a bug exists. For example, the static data may simply be a pointer into a dynamic structure that does not exist until runtime. We may not be able to tell when another process or thread is going to change the variable—race conditions are extremely difficult to track down even when testing dynamically.

In complex code using interrupts to guide control flow, or when there are multiple levels of prioritization that may occur, leaving the operating system to

decide what will execute when, static testing is pretty much guaranteed not to find all of the interesting bugs that can occur.

It is important to remember that testing is a filtering process. We find some bugs with this technique, some with that, some with another. There are many times that data flow analysis will find defects that might otherwise ship. As always, we use the techniques that we can afford, based on the context of the project we are testing. Data flow analysis is one more weapon in our testing arsenal.

3.2.2.1 Define-Use Pairs

Data flow notation comes in a few different flavors; we are going to look at one of them here called *define-use pair* notation. In the glossary, this is called “definition-use pair,” and we have seen it as “set-use.”

To use this notation, we split the life cycle of a data variable into three separate patterns:

- d:** This stands for the time when the variable is created, defined, initialized, or changed.
- u:** This stands for used. The variable may be used in a computation or in a decision predicate.
- k:** This stands for killed or destroyed, or the variable has gone out of scope.

These three atomic patterns are then combined to show a data flow. A ~ (tilde) is often used to show the first or last action that can occur.

Table 3-1 Data flow combinations

	Anomaly		Explanation
1.	~d	first define	Allowed.
2.	du	define-use	Allowed, normal case.
3.	dk	define-kill	Potential bug; data was never used.
4.	~u	first use	Potential bug; data was used without definition. It may be a global variable, defined outside the routine.
5.	ud	use-define	Allowed: data used and then redefined.
6.	uk	use-kill	Allowed.
7.	~k	first kill	Potential bug; data is killed before definition.
8.	ku	kill-use	Serious defect; data is used after being killed.
9.	kd	kill-define	Usually allowed. Data is killed and then re-defined. Some theorists believe this should be disallowed.

10.	dd	define-define	Declare-define may be normal for language. Potential bug: double definition without intervening use.
11.	uu	use-use	Allowed; normal case. Some do not bother testing this pair since no redefinition occurred.
12.	kk	kill-kill	Likely bug.
13.	d~	define last	Potential bug; dead variable? May be a global variable that is used in other context.
14.	u~	use last	Allowed but possibly dodgy. Variable was used in this routine but not killed off. Could be possible memory leak.
15.	k~	kill last	Allowed; normal case.

Referring to Table 3–1, there are 15 potential combinations of the atomic actions we are concerned with, as follows:

1. ~d or first define. This is the normal way a variable is originated; it is declared. Note that this particular data flow analysis methodology is somewhat hazy as to whether at this point the value is defined or not. A variable is declared to allocate space in memory for it; at that point, however, the value the variable holds is unknown (although some languages have a default value that is assigned, often zero or NULL). The variable needs to be set before it is read (or in other words, must be used in the left side of an assignment statement before it is used in the right side of an assignment statement, as an argument to a function, or in a decision predicate). Other data flow schemes have a special state for a ~d; when first declared, it holds a value of “uninitialized.”
2. du or define-use. This is the normal way a variable is used. Defined first and then used in an assignment or decision predicate. Of course, this depends on using the correct data type and size—which should be checked in a code review.
3. dk or define-kill. This is a likely bug; the variable was defined and then killed off. The question must be asked as to why it was defined. Is it dead? Or was there a thought of using it but the wrong variable was actually used in later code? If creating the variable caused a desirable side effect to occur, this might be done purposefully.
4. ~u or first use. This is a potential bug because the data was used without a known definition. It may not be a bug because the definition may have occurred at a different scope. For example, it may be a global variable, set in a

different routine. This should be considered dodgy and should be investigated by the tester. After all, race conditions may mean that the variable never does get initialized in some cases. In addition, the best practice is to limit the use of global variables to a few very special purposes.

5. ud or use-define. This is allowed where the data is read and then set to a different value. This can all occur in the same statement where the use is on the right side of an assignment and the definition is the left side of the statement, such as $X = X + 1$, a simple increment of the value.
6. uk or use-kill. This is expected and normal behavior.
7. ~k or first kill. This is a potential bug, where the variable is killed before it is defined. It may simply be dead code, or it might be a serious failure waiting to happen, such as when a dynamically allocated variable is destroyed before actually being created, which would cause a runtime error.
8. ku or kill-use. This is always a serious defect where the programmer has tried to use the variable after it has gone out of scope or been destroyed. For static variables, the compiler will normally catch this defect; for dynamically allocated variables, it may cause a serious runtime error.
9. kd or kill-define. This is usually allowed where a value is destroyed and then redefined. Some theorists believe this should be disallowed; once a variable is destroyed, bringing it back is begging for trouble. Others don't believe it should be an issue. Testers should evaluate this one carefully if it is allowed.
10. dd or define-define. In some languages, it is standard to declare a variable and then define it in a different line. In other languages, the variable may be declared and defined in the same line. As you will see in our example, there may be times when this is useful. And sometimes, it may indicate a defect.
11. uu or use-use. This is normal and done all of the time.
12. kk or kill-kill. This is likely to be a bug—especially when using dynamically created data. Once a variable is killed, trying to access it again—even to kill it—will cause a runtime error in most languages.
13. d~ or define last. While this is a potential bug—a dead variable that is never used, it might just be that the variable is meant to be global and will be used elsewhere. The tester should always check all global variable use very carefully. If not a global variable, this may result in a memory leak.
14. u~ or use last. The variable was used but not killed on this path. This may be a potential memory leak if the variable was created dynamically and not killed (deallocated) elsewhere in the code.
15. k~ or kill last. This is the normal case.

Following is an example to show how we use define-use pair analysis.

3.2.2.2 Define-Use Pair Example

Assume a telephone company that provides the following cell-phone plan:

If the customer uses up to 100 minutes (inclusive), then there is a flat fee of \$40.00 for the plan. For all minutes used from 101 to 200 minutes (inclusive), there is an added fee of \$0.50 per minute. Any minutes after that used are billed at \$0.10 per minute. Finally, if the bill is over \$100.00, a 10 percent discount on the entire bill is given.

A good tester might immediately ask the question as to how much is billed if the user does not use the cell phone at all. Our assumption—based on what we have seen in the real world—would be that the user would still have to pay the \$40.00. However, the code as given in Figure 3–6 lets the user off completely free if the phone was not used for the month.

```
1.     public static double calculateBill (int Usage) {  
2.         double Bill = 0;  
3.         if (Usage > 0) {  
4.             Bill = 40;  
5.  
6.             if (Usage > 100) {  
7.                 if (Usage <= 200) {  
8.                     Bill = Bill + (Usage - 100) * 0.5;  
9.                 }  
10.            else {  
11.                Bill = Bill + 50 + (Usage - 200) * 0.1;  
12.                if (Bill >= 100) {  
13.                    Bill = Bill * 0.9;  
14.                }  
15.            }  
16.        }  
17.  
18.        return Bill;  
19.    }
```

Figure 3–6 Cell phone billing example

Line 3 looks at the number of minutes billed. If none were billed, the initial \$40.00 is not billed; instead, it evaluates to FALSE and a \$0.00 bill is sent. Frankly, we want to do business with this phone company, and if we worked there, we would flag this as an error.

Assuming a little time was used, however, the bill is set to \$40.00 in line 4. In line 6 we check to see if there were more than 100 minutes used; in line 7 we check if more than 200 minutes were used. If not, we simply calculate the extra minutes over 100 and add \$0.50 for each one. If over 200 minutes, we take the base bill, add \$50.00 for the first extra 100 minutes, and then bill \$0.10 per minute for all extra minutes. Finally, we calculate the discount if the bill is over \$100.00.

For each variable, we create a define-use pattern list that tracks all the changes to that variable through the module. In each case we track the code line(s) that an action takes place in.

For the code in Figure 3–6, here is the data flow information for the variable *Usage*:

Table 3–2 Variable usage define-use pattern list

	Pattern	Explanation
1.	$\sim d$ (1)	normal case
2.	du (1-3)	normal case
3.	uu (3-6)(6-7)(7-8)(7-11)	normal case
4.	uk (6-19)(8-19)(11-19)	normal case
5.	$k\sim$ (19)	normal case

1. The *Usage* variable is created in line 1. It is actually a formal parameter that is passed in as an argument when the function is called. In most languages, this will be a variable that is created on the stack and immediately set to the passed-in value.⁵
2. There is one *du* (define-use) pair at (1-3). This is simply saying that the variable is defined in line 1 and used in line 3 in the predicate for the *if* statement. This is expected behavior.
5. To be a bit more precise, languages that create a copy of the value of the function's parameter when a function is called are said to use call by value parameters. If instead the function receives a pointer to the memory location where the parameter resides, the language is said to use call by reference parameters. Languages using call by value parameters can implement call by reference parameters by passing in a pointer to the parameter explicitly. This can become a source of confusion because, if the parameter is an array, then a pointer is passed even in call by value languages. This aids efficiency because the entire contents of the array need not be copied to the stack, but the programmer must remember that modifications to the values in the array will persist even after the function returns.

3. Each time that *Usage* is used following the previous *du* pair, it will be a *uu* (use-use) pair until it is defined again or killed.
4. So it is used in line 3 and line 6. Then comes (6-7), (7-8), and (7-11). Notice there is no (8-11) pair because under no circumstances can we execute that path. Line 7 is in the TRUE branch of the conditional and line 11 is in the FALSE path.
5. Under *uk* (use-kill), there are three possible pairs that we must deal with.
 - (6-19) is possible when *Usage* has a value of 100 or less. We use it in line 6 and then the next touch is when the function ends. At that time, the stack-frame is unrolled and the variable goes away.
 - (8-19) is possible when *Usage* is between 101 and 200 inclusive. The value of *Bill* is set and then we return.
 - (11-19) is possible when *Usage* is greater than 200.
 - Note that (3-19) and (7-19) are not possible because in each case, we must touch *Usage* again before it is destroyed. For line 3, we must use it again in line 6. For line 7, we must use it in either line 8 or 11.
6. Finally, at line 19 we have a kill last on *Usage* because the stack frame is removed.

It is a little more complicated when we look at the local variable *Bill* as shown in Table 3–3.

Table 3–3 Variable *Bill* define-use pattern list

	Pattern	Explanation
1.	$\sim d$ (2)	normal case
2.	<i>dd</i> (2-4)	suspicious
3.	<i>du</i> (2-18)(4-8)(4-11)(11-12)	normal case
4.	<i>ud</i> (8-8)(11-11)(13-13)	acceptable
5.	<i>uu</i> (12-13)(12-18)	normal case
6.	<i>uk</i> (18-19)	normal case
7.	<i>k~</i> (19)	normal case

1. $\sim d$ signifies that this is the first time this variable is defined. This is normal for a local variable declaration.
2. *dd* (define-define) could be considered suspicious. Generally, you don't want to see a variable redefined before it is used. However, in this case it is

fine; we want to make sure that *Bill* is defined before first usage even though it could be redefined. Note that if we did not set the value in line 2, then if there were no phone minutes at all, we would have returned an undefined value at the end. It might be zero—or it might not be. In some languages it is not permissible to assign a value in the same statement that a variable is declared in. In such a case, the *if()* statement on line 3 would likely be given an *else* clause where *Bill* would be set to the value of 0. The way this code is currently written is likely more efficient than having another jump for the *else* clause.

3. *du* (define-use) has a variety of usages.
 - (2-18) happens when there are no phone minutes and ensures that we do not return an undefined value.
 - (4-8) occurs when *Usage* is between 101 and 200 inclusive. Please note that the “use” part of it is on the right side of the statement—not the left side. The value in *Bill* must be retrieved to perform the calculation and then it is accessed again (after the calculation) to store it.
 - (4-11) occurs when the *Usage* is over 200 minutes. Again, it is used on the right side of the statement, not the left.
 - (11-12) occurs when we reset the value (in the left side of the statement on line 11) and then turn around and use it in the conditional in line 12.
4. *ud* (use-define) occurs when we assign a new value to a variable. Notice that in lines 8, 11, and 13, we assign new values to the variable *Bill*. In each case, we also use the old value for *Bill* in calculating the new value. Note that in line 4, we do not use the value of *Bill* in the right side of the statement. However, because we do not actually use *Bill* before that line, it is not a *ud* pair; instead, as mentioned in (2), it is a *dd* pair. In each case, this is considered acceptable behavior.
5. *uu* (use-use) occurs in lines (12-13). In this case, the value of *Bill* is used in the decision expression in line 12 and then reused in the right side of the statement in line 13. It can also occur at (12-18) if the value of *Bill* is less than \$100.
6. *uk* (use-kill) can only occur once in this code (18-19) since all execution is serialized through line 18.
7. And finally, the *k~* (kill last) occurs when the function ends at line 19. Since the local variable goes out of scope automatically when the function returns, it is killed.

Once the data flow patterns are defined, testing becomes a relatively simple case of selecting data such that each defined pair is covered. Of course, this does not guarantee that all data-related bugs will be uncovered via this testing. Remember the factorial example in Chapter 2? Data flow testing would tell us the code was sound; unfortunately, it would not tell us that when we input a value of 13 or higher, the loop would blow up.

A wise mentor who taught Jamie a great deal about testing once said, “If you want a guarantee, buy a used car!”

If you followed the previous discussion, where we were trying to do our analysis directly from the code, you might guess where we are going next. Code is confusing. Often, it is conceptually easier to go to a control flow diagram and perform the data flow analysis from that.

In Figure 3–7 we see the control flow diagram that matches the code. Make sure you match this to the code to ensure that you understand the conversion. We will use this in an exercise directly. From this, we can build an annotated control flow diagram for any variable found in the code.

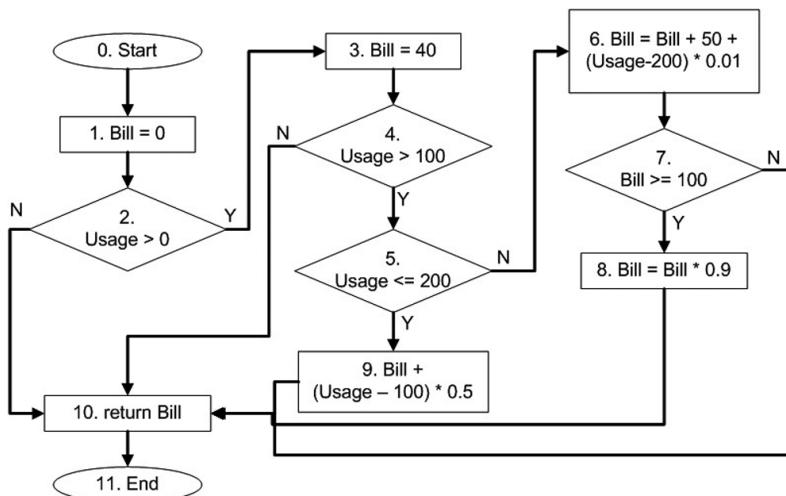


Figure 3–7 Control flow diagram for example

Using the control flow diagram from Figure 3–7, we can build an annotated flow graph for the variable *Bill* as seen in Figure 3–8. It is much easier to find the data connections when laid out this way, we believe. Note that we have simply labeled each element of the flow with information about *Bill* based on d-u-k values.

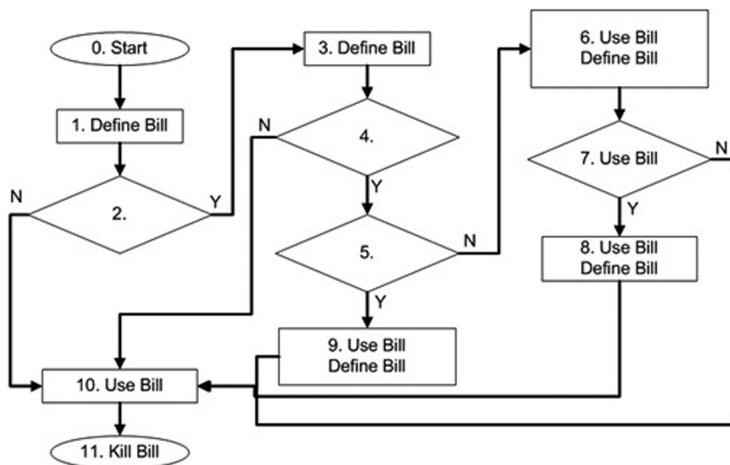


Figure 3–8 Annotated flow graph for variable Bill

Table 3–4 Define-use pairs for the variable Bill

Anomaly		Explanation
~d	1	Normal case
dd	1-2-3	Potential bug
du	1-2-10 3-4-5-6 3-4-5-9 3-4-10 6-7 8-10 9-10	Normal case Normal case Normal case Normal case Normal case Normal case Normal case
ud	6-6 8-8 9-9	Normal case Normal case Normal case
uu	7-8 7-10	Normal case Normal case
uk	10-11	Normal case
k~	11	Normal case

From the annotated flow graph, we can create a table of define-use pairs as shown in Table 3–4. To make it more understandable, we have expanded the normal notation ($x-y$) to show the intervening steps also.

Looking through this table of data flows, we come up with the following:

- One place where we have a first define (line 1)
- One potential bug where we double define (*dd*) in the flow 1-2-3
- Seven separate *du* pairs where we define and then use *Bill*
- Three separate *ud* pairs where we use and then redefine *Bill*
- Two separate *uu* pairs where we use and then reuse *Bill*
- One *uk* pair where we use and then kill *Bill*
- And finally, one place where we kill *Bill* last

Why use an annotated control flow rather than using the code directly? We almost always find more data flow pairs when looking at a control flow graph than at a piece of code.

3.2.2.3 Data Flow Exercise

Using the control flow and the code previously shown, build an annotated control flow diagram for the variable *Usage*.

Perform the analysis for *Usage*, creating a d-u-k table for it.

3.2.2.4 Data Flow Exercise Debrief

Here is the annotated data flow for the code:

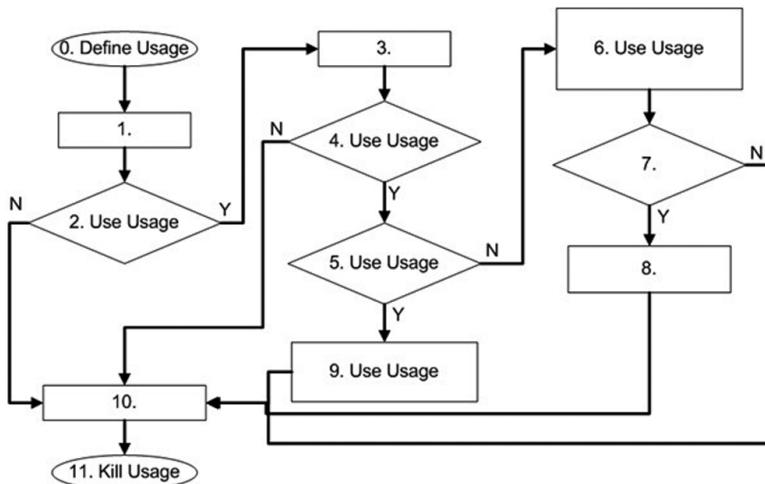


Figure 3–9 Annotated flow graph for *Usage*

Below is the table that represents the data flows.

Table 3–5 Define-use pairs for Usage

Anomaly		Explanation
~d	0	Normal case
du	(0-1-2)	Normal usage
uu	(2-3-4), (4-5), (5-6), (5-9)	Normal usage
uk	(2-10-11), (4-10-11), (6-7-10-11), (6-7-8-10-11), (9-10-11)	Normal usage
k~	11	Normal usage

- The variable is defined in 0.
- It is first used in 2 (0,1,2 is the path).
- There are four use-use relationships, at (2-3-4), (4-5), (5-6), and (5-9).
- There are five use-kill relationships, at (2-10-11), (4-10-11), (6-7-10-11), (6-7-8-10-11), and (9-10-11).
- There is a final kill at 11.

Note that all of the data flows appear normal.

3.2.2.5 A Data Flow Strategy

What we have examined so far for data flow analysis is a technique for identifying data flows; however, this does not rise to the status of being a full test strategy. In this section, we will examine a number of possible data flow testing strategies that have been devised. Each of these strategies is intended to come up with meaningful testing—short of full path coverage. Since we don't have infinite time or resources, it is essential to figure out where we can trim testing without losing too much value.

The strategies we will look at all turn on how variables are used in the code. Remember that we have seen two different ways that variables can be used in this chapter: in calculations (called *c-use*) or in predicate decisions (called *p-use*).

When we say that a variable is used in a calculation, we mean that it shows up on the right-hand side of an assignment statement. For example, consider the following line of code:

$$Z = A + X;$$

Both *A* and *X* are considered *c-use* variables; they are brought into the CPU explicitly to be used in a computation where they are algebraically combined and, in this case, assigned to the memory location, *Z*. In this particular line of

code, Z is a *define* type use, so not of interest to this discussion. Other uses of *c-use* variables include as pointers, as parts of pointer calculations, and for file reads and writes.

For predicate decision (*p-use*) variables, the most common usage is when a variable appears directly in a condition expression. Consider this particular line of code:

```
if (A < X) { }
```

This is an *if()* statement where the predicate is $(A < X)$. Both A and X are considered *p-use* variables. Other *p-use* examples include variables used as the control variable of a loop, used in expressions used to evaluate the selected branch of a case statement, or used as a pointer to an object that will direct control flow.

In some languages a variable may be used as both *p-use* and *c-use* simultaneously; for example a test-and-clear machine language instruction. That type of use is beyond the scope of this book so we will ignore it for now. We will simply assume that each variable usage is one or the other, *p-use* or *c-use*.

The first strategy we will look at is also the strongest. That is, it encompasses all of the other strategies that we will discuss. It is called the *all du* path strategy (*ADUP*). This strategy requires that every single *du* path, from every definition of every variable to every use of that variable (both *p-use* and *c-use*), be exercised. While that sounds scary, it might not be as bad as it sounds since a single test likely exercises many of the paths.

The second strategy, called *all-uses* (*AU*), relaxes the requirement that every path be tested to require that at least one path segment from every definition to every use (both *p-use* and *c-use*) that can be reached by that definition be tested. For *ADUP* coverage, we might have several paths that lead from a definition to a use of the variable, even though those paths do not have any use of the variable in question.

To differentiate the two, consider the following example. Assume a code module wherein a variable X is defined in line 10 and then used in line 100 before being defined again. Further, assume that there is a switch statement with 10 different branches doing something that has nothing to do with the variable X. To achieve *ADUP*, we would have to have 10 separate tests, one through each branch of the switch statement leading to the use of the variable X. *AU* coverage would require only 1 test for the *du* pair, through any single branch of the switch statement.

The *all-definitions (AD)* strategy requires only that every definition of every variable be covered by at least one use of that variable, regardless of whether it is a *p-use* or *c-use*.

The remaining strategies we will discuss differentiate between testing *p-use* and *c-use* instances for *du* pairs.

Two of these strategies are mirror images of each other. These include *all p-uses/some c-uses (APU+C)* and *all c-uses/some p-uses (ACU+P)*. Each of these states that for every definition of a specific variable, you must test at least one path from the definition to every use of it. That is, to achieve *APU+C* coverage, for each variable (and each definition of that variable), test at least one definition-free path (*du*) to each predicate use of the variable. Likewise, for *ACU+P*, we must test every variable (and every definition of that variable) with at least one path to each computational use of the variable. Then, if there are definitions not yet covered, fill in with paths to the off-type of use. While the math is beyond the scope of this book, Boris Beizer claims that *APU+C* is stronger than decision coverage but *ACU+P* may be weaker or even not comparable to branch coverage.

Two more strategies, *all p-uses (APU)* and *all c-uses (ACU)*, relax the requirement that we test paths not covered by the *p-use* or *c-use* tests. Note that this means some definition-use paths will not be tested.

We fully understand that the previous few paragraphs are information dense. Remember, our overriding desire is to come up with a strategy that gives us the right amount of testing for the context of our project at a cost we can afford. There is a sizable body of research that has been done trying to answer the most important question: Which of these strategies should I use for my project? The correct answer, as so often in testing, is it depends.

A number of books have dedicated a lot of space discussing these strategies in greater detail. A very thorough discussion can be found in *The Compiler Design Handbook*.⁶ A somewhat more comprehensive discussion can be found in *Software Testing Techniques*.⁷

6. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*, by Y.N. Srikant and Priti Shankar.

7. *Software Testing Techniques, Second Edition*, by Boris Beizer.

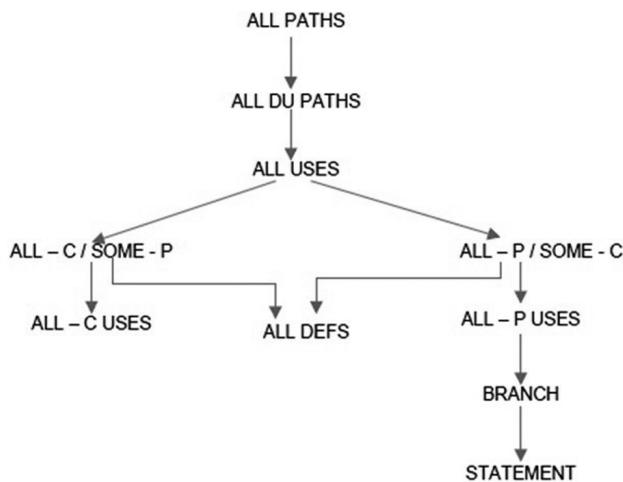


Figure 3–10 Rapps/Weyuker hierarchy of coverage metrics

Figure 3–10 may help because it tries to put the preceding discussion into context. At the very top of the structure is *all paths*—the holy grail of testing which, in any non-trivial system, is impossible to achieve. Below that is *ADUP*, which gives the very best possible coverage at the highest cost in test cases. Next comes *all uses*, which will require fewer tests at the cost of less coverage.

At this point, there is a discontinuity between the testing methods. The *all c-uses* testing (*ACU+P* and *ACU*) are shown down the left-hand side of the figure. These are not comparable with their mirror image shown on the right-hand side; mathematically, they are inherently weaker than the other strategies at their same level. The *all p-uses* (*APU+C* and *APU*) are stronger than branch coverage. The *all defs* (*AD*) strategy is weaker than *APU+C* and *ACU+P* but possibly stronger than *APU* and *ACU*.

Figure 3–10 should make the relative bug detection strengths—and also the costs—of these strategies clearer. Bottom line, a technical tester should understand that there are different strategies, and if the context of your project requires a high level of coverage, one of these strategies may be the place to start.

3.2.3 Static Analysis to Improve Maintainability

Learning objectives

TTA-3.2.3 (K3) Propose ways to improve the maintainability of code by applying static analysis.

Many years ago Jamie was doing some work as a C programmer, writing application code for Windows 3.1. As a voracious reader, he subscribed to a multitude of programming periodicals. In one of the journals, there was an interesting repeating feature article that used to excite him as a programmer but scandalize him as a tester. On the last page of the magazine was an obfuscation challenge: who could stick the most functionality in a single line of C code? Each time he got the magazine from the mailbox, it was always the first place he would look.

To be honest, most of the time Jamie was not able to comprehend exactly what the code was supposed to do.

This would be a funny anecdote and a happy memory if code we had tested didn't so often appear to have been written by a programmer intent on winning the title for obfuscation of the month.

Very often programmers seem to write their code to be elegant—to them. However, this can be problematic if others (who may have to read, change, maintain, or test it) cannot understand it.

We will discuss the quality characteristic of *Maintainability* more thoroughly in Chapter 4. In the following sections, we will discuss the capability of static analysis to help improve maintainability. A great deal of maintainability comes from programmers following a common set of standards and guidelines to ensure that everyone who reads the code can understand it. And, there are static analysis tools that can parse through the code to determine if those standards and guidelines have been followed.

3.2.3.1 Code Parsing Tools

In Figure 3–11, we show the output of a static code parsing tool that is looking for issues that the compiler would not flag because they are not syntactically incorrect.⁸ The code on the left is an example of a very poorly written C language module. The problem is that the code does not necessarily look bad. Many developers figure that as long as the compiler does not throw an error, the code

8. This example comes from the *Wikipedia* article on Splint. This reference was inadvertently left off in the first edition.

probably is correct. Then they spend hours debugging it when it does not do what they expect. On the right is the output of a tool called Splint, an open-source static analysis tool for C that parsed this code.

```
#include <stdio.h>
int main () {
    char c;
    while (c != 'x'); {
        c = getchar();
        if (c == 'x') return 0;
        switch (c) {
            case '\n':
            case '\r':
                printf("Newline\n");
            default:
                printf("%c", c);
        }
    }
    return 0; }
```

Output of **Splint**, an open source static analysis tool

1. Variable c used before definition
2. Suspected infinite loop. No value used in loop test (c) is
3. Assignment of int to char: c = getchar()
4. Test expression for if is assignment expression: c = 'x'
5. Test expression for if not Boolean, type char: c = 'x'
6. Fall through case (no preceding break)

Figure 3–11 Code parsing tools

Notice in item 1, the variable *c* is evaluated before it is set. That is not illegal, just dim-witted. The initial value of variable *c* is set to whatever random value it was when the code initialized. It probably is not equal to the character ‘*x*’, and so it will probably work almost every time. Almost! In those few cases when it does initialize to ‘*x*’, the system will just return, having done nothing. Some programming languages, such as C#, will flag this as an error when compiled; most programming languages allow it. Other programming languages will allow it but also will initialize the value of all variables to zero, which has the effect of preventing this particular problem but creating a certain problem if the value checked in the *while* condition were to be zero.

In item 2, the code is probably not doing what the user wanted; the static analysis tool is warning that there is no definite end to the loop. This is caused by the semicolon following the *while* statement. That signifies an empty statement (perfectly legal) but also represents an infinite loop. Many programmers reading this code do not see that the entire body of the *while* loop consists of the empty statement represented by the semicolon.

Item 3 is a data typing mistake that might cause problems on some architectures and not on others. The output of the common library routine *getchar()* is a data type *int*. It is being assigned to a data type *char*. It probably will work in

most instances; years ago, this kind of assignment was common practice. We can foresee problems working with double-byte character sets and some architectures where an *int* is much larger than a *char*. The programmer is assuming that the assignment will always go correctly. They might be right...

Items 4 and 5 represent a definite bug. This is a common mistake that anyone new to C is going to make again and again. In C, the assignment operator is a single equal sign (=) while the Boolean equivalence operator is a double equal sign (==). Oops! Rather than seeing if the inputted char is an ‘x’, this code is explicitly setting it to ‘x’. Since the output of the assignment by definition is TRUE, we will return 0 immediately. No matter what character is typed in, this code will see it as an ‘x’.

Even if the Boolean expression was evaluated correctly, even if the assignment of the input was correct between the data types, this code would still not do what the programmer expected. Item 6 points out that a *break* reserved word was not used after the *newline* was outputted. That means that the default *printf()* would also be executed each time, rather than only when a *newline* or *carriage return* was entered. That means the formatting of the output would be incorrect.

Many compilers have the ability to return warnings such as we see here with the static analysis tool. Oftentimes, however, the warning messages given by a compiler are obtuse and difficult to understand. Sometimes the warning messages are for conditions that the programmers understand and wrote on purpose. For these and many other reasons, programmers often start ignoring various warning messages from the compiler. Some compilers and static analysis tools allow the user to selectively turn off certain classes of messages to avoid the clutter.

For testers, however, it is likely worth the time to make sure every type of warning is evaluated. The more safety or mission critical the software, the more this is true. Some organizations require that all compiler and static analysis tool warnings be turned over to the test team for evaluation.

Static analysis tools can also be used to check if the tree structure of a website is well balanced. This is important so that the end user can navigate to their intended pages easier, and it simplifies the testing tasks and reduces the amount of maintenance needed on the site by the developers.

3.2.3.2 Standards and Guidelines

Programmers sometimes consider programming standards and guidelines to be wasteful and annoying. We can understand this viewpoint to a limited extent; they are often severely time and resource constrained. Better written and better documented code takes a little more time to write. And, there may be times when standards and guidelines are just too onerous. Project stakeholders, however, should remind programmers that—in the long term—higher maintainability of the system will save time and effort.

Jamie once heard software developers described as the ultimate optimists. No matter how many times they get burned, they always seem to assume that “this time, we’ll get lucky and everything will work correctly.” A lament often heard after a new build turns out to be chock-full of creepy-crawlers: Why is it we never have time to do it right, but we always can take time to do it over?

Perhaps it is time to take the tester’s attitude of professional pessimism seriously and start taking the time to do it right up front.

There are a number of static analysis tools that can be customized to allow local standards and guidelines to be enforced. And, some features might be turned off at certain times if that makes economic or strategic sense. For example, if we are writing an emergency patch, the requirement for following exact commenting guidelines might be relaxed for this session. “Heck, we can fix it tomorrow!” Having said that, in many of the places we have worked, it often seemed that we were always in emergency mode. The operative refrain was always, “We’ll get to that someday!” As Creedence Clearwater Revival pointed out, however, “Someday never comes.”

In the following paragraphs, we have recorded some of the standards and guidelines that can be enforced if we were programming in Java and using a static analysis tool called Checkstyle. This open-source tool was created in 2001 and is currently in use around the world.

- Check for embedded Javadoc comments. Javadoc is a protocol for embedding documentation in code that can be parsed and exhibited in HTML format. Essentially, this is a feature that allows an entire organization to write formal documentation for a system. Of course, it only works when every developer includes it in their code.
- Enforce naming conventions to make code self-documenting. For example, functions might be required to be named as action verbs with the return data type encoded in them (e.g., `strCalculateHeader()`, `recPlotPath()`, etc.). Constants may be required to be all uppercase. Variables might have their data

type encoded (i.e., using Hungarian notation). There are many different naming conventions that have been used; if everyone in an organization were to follow the standards and guidelines, it would be much easier to understand all the code, even if you did not write it.

- Check the cyclomatic complexity of a routine against a specified limit automatically.
- Ensure that required headers are in place. These are often designed to help users pick the right routines to use in their own code rather than rewriting them. Often the headers will enclose parameter lists, return value, side effects, and so on.
- Check for magic numbers. Sometimes developers use constant values rather than named constants in their code. This is particularly harmful when the value changes because all occurrences must also be changed. Using a named constant allows all changes to occur at once by making the change in a single location.
- Check for white space (or lack of same) around certain tokens or characters. This gives the code a standardized look and often makes it easier to understand.
- Enforce generally agreed-upon conventions in the code. For example, there is a document called *Code Conventions for the Java Programming Language*.⁹ This document reflects the Java language coding standards presented in the *Java Language Specification*¹⁰ by Sun Microsystems (now Oracle). As such, it may be the closest thing to a standard that Java has. Checkstyle can be set to enforce these conventions, which include details on how to build a class correctly.
- Search the code looking for duplicate code. This is designed to discourage copy-and-paste operations, generally considered a really bad technique to use. If the programmer needs to do the same thing multiple times, they should refactor it into a callable routine.
- Check for visibility problems. One of the features of object-oriented design is the ability to hide sections of code. An object-oriented system is layered with the idea of keeping the layers separated. If we know how a class is designed, we might use some of that knowledge when deriving from that class. How-

9. The URL that was in the first edition of this book is now dead. Apparently the code conventions document was last updated in 1999, and was somewhat out of date. However, there is a lot of discussion on the Web about coding conventions, so we recommend searching on that phrase—there appears to be a lot of information out in the ether.

10. <http://docs.oracle.com/javase/specs/>

ever, using that knowledge may become problematic if the owner of that class decides to change the base code. Those changes might end up breaking our code. In general, when we derive a new class, we should know nothing about the super class beyond what the owner of it decides to explicitly show us. Visibility errors occur when a class is not correctly designed, showing more than it should.

These are just a few of the problems this kind of static analysis tool can expose.

3.2.4 Call Graphs

Learning objectives

TTA-3.2.4 (K2) Explain the use of call graphs for establishing integration testing strategies.

In the Foundation level syllabus, integration testing was discussed as being either incremental or nonincremental. The incremental strategy was to build the system a few pieces at a time, using drivers and stubs to replace missing modules. We could start at the top level with stubs replacing modules lower in the hierarchy and build downward (i.e., *top down*). We could start at the bottom level and, using drivers, build upward (i.e., *bottom up*). We could start in the middle and build outward (the *sandwich* or the *backbone* method). We could also follow the functional or transactional path.

What these methods have in common is the concept of test harnesses. Consisting of drivers, stubs, emulators, simulators, and all manners of mock objects, test harnesses represent nonshippable code that we write to be able to test a partial system to some level of isolation.

There was a second strategy we discussed called the “big bang,” a nonincremental methodology that required that all modules be available and built together without incremental testing. Once the entire system was put together, we would then test it all together. The Foundation level syllabus is pretty specific in its disdain for the big bang theory: In order to ease fault isolation and detect defects early, integration should normally be incremental rather than “big bang.”¹¹

11. Foundation level syllabus, section 2.2.2

The incremental strategy has the advantage that it is intuitive. After all, if we have a partial build that appears to work correctly and then we add a new module and the partial build fails, we all think we know where the problem is.

In some organizations, however, the biggest roadblock to good incremental integration testing is the large amount of nonshippable code that must be created. It is often seen as wasteful; after all, why spend time writing code that is not included in the build or destined for the final users? Frankly, extra coding that is not meant to ship is very hard to sell a project manager when other testing techniques (e.g., the big bang) are available. In addition, when the system is non-trivial, a great number of builds are needed to test in an incremental way.

In his book *Software Testing, A Craftsman's Approach*, Paul Jorgensen suggests another reason that incremental testing might not be the best choice. Functional decomposition of the structure—and therefore integration by that structure—is designed to fit the needs of the project manager rather than the needs of the software engineers. Jorgensen claims that the whole mechanism presumes that correct behavior follows from individually correct units (that have been fully unit-tested) and correct interfaces. Further, the thought is that the physical structure has something to do with this. In other words, the entire test basis is built upon the logic of the functional decomposition, which likely was done based on the ease of creating the modules or the ability to assign certain modules to certain developers.

3.2.4.1 Call-Graph-Based Integration Testing

Rather than physical structure being paramount, Jorgensen suggests that call-graph-based integration makes more sense. Simply because two units are structurally next to each other in a decomposition diagram, it does not necessarily follow that it is meaningful to test them together. Instead, he suggests that what makes sense is to look at which modules are working together, calling each other. His suggestion is to look at the behavior of groups of modules that cooperate in working together using graph theory.

Two arguments against incremental integration are the large number of builds needed and the amount of nonshippable code that must be written to support the testing. If we can reduce both of those and still do good testing, we should consider it. Of course, there are still two really good arguments for incremental testing: it is early testing rather than late, and when we find failures, it is usually easier to identify the causes of them. Using call graphs for integration testing would be considered a success if we can reduce the former two (number

of builds and cost of nonshippable code) without exacerbating the latter two (finding bugs late and the difficulty of narrowing down where the bugs are).

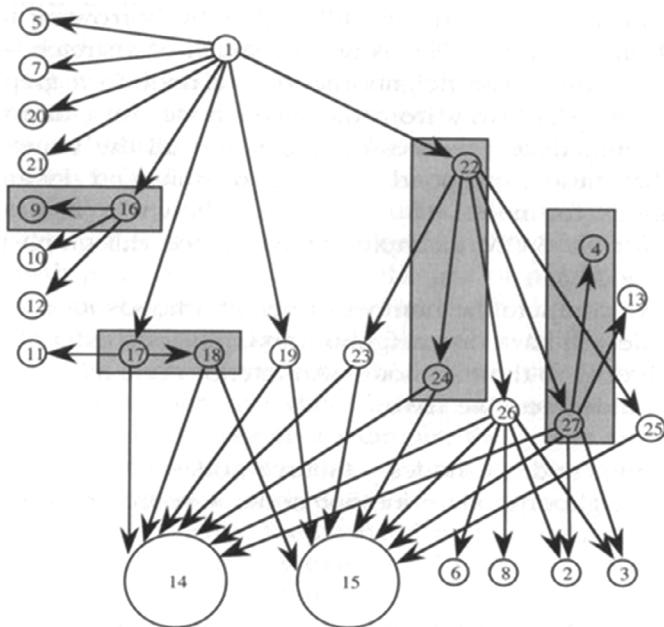


Figure 3–12 Pairwise graph example

The graph from Paul Jorgensen's book in Figure 3–12 is called a pairwise graph. The gray boxes shown on the graph denote pairs of components to test together. The main impetus for pairwise integration is to reduce the amount of test harness code that must be written. Each testing session uses the entire build but targets pairs of components that work together (found by looking at the call graph as shown here). Theoretically, at least, this reduces the problem of determining where the defect is when we find a failure. Since we are not concentrating on system-wide behavior but targeting a very specific set of behaviors, failure root cause should be easier to determine.

The number of testing sessions is not reduced appreciably, but many sessions can be performed on the same build, so the number of needed builds may be minimally reduced. On the other hand, the amount of nonshippable code that is required is reduced to almost nothing since the entire build is being tested.

In the graph of Figure 3–12, four sessions are shown, modules 9–16, 17–18, 22–24, and 4–27.

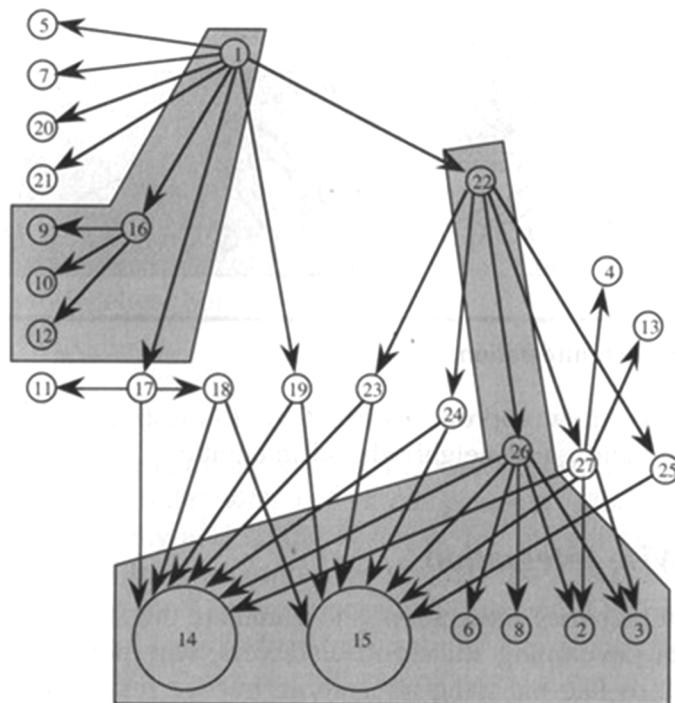


Figure 3–13 Neighborhood integration example

The second technique, as shown in Figure 3–13, is called neighborhood integration. This uses the same integration graph that you saw in Figure 3–12 but groups nodes differently. The neighborhood integration theory allows us to reduce the number of test sessions needed by looking at the neighborhood of a specific node. That means all of the immediate predecessor and successor nodes of a given node are tested together. In Figure 3–13, two neighborhoods are shown: that of node 16 and that of node 26. By testing neighborhoods, we reduce the number of sessions dramatically, though we do increase the possible problem of localizing failures. Once again, there is little to no test harness coding needed. Jorgensen calls neighborhood testing “medium bang” testing.

Clearly we should discuss the good and bad points of call-graph integration testing. On the positive side, we are now looking at the actual behavior of actual code rather than simply basing our testing on where a module exists in the hierarchy. Savings on test harness development and maintenance costs can be appreciable. Builds can be staged based on groups of neighborhoods, so scheduling them can be more meaningful. In addition, neighborhoods can be

merged and tested together to give some incrementalism to the testing (Jorgensen inevitably calls these merged areas villages).

The negative side of this strategy is appreciable. We can call it medium bang or target specific sets of modules to test, but the entire system is still running. And, since we have to wait until the system is reasonably complete so we can avoid the test harnesses, we end up testing later than if we had used incremental testing.

There is another, subtle issue that must be considered. Suppose we find a failure in a node in the graph. It likely belongs to several different neighborhoods. Each one of them should be retested after a fix is implemented; this means the possibility of appreciably more regression testing during integration testing.

In our careers, we have never had the opportunity to perform pairwise or neighborhood integration testing. In researching this course, we have not been able to find many published documents even discussing the techniques. While the stated goals of reducing extraneous coding and moving toward behavioral testing are admirable, one must wonder whether the drawbacks of late testing make these two methods less desirable.

3.2.4.2 McCabe's Design Predicate Approach to Integration

How many test cases do we need for doing integration testing? Good test management practices and the ability to estimate how many resources we need both require an answer to this question. When we were looking at unit testing, we found out that McCabe's cyclomatic complexity could at least reveal the basis set—the minimum number of test cases that we needed.

It turns out that Thomas McCabe also had some theories about how to approach integration testing. This technique, which is called McCabe's design predicate approach, consists of three steps as follows:

1. Draw a call graph between the modules of a system showing how each unit calls and is called by others. This graph has four separate kinds of interactions, as you shall see in a bit.
2. Calculate the integration complexity.
3. Select tests to exercise each type of interaction, not every combination of all interactions.

As with unit testing, this does not show us how to exhaustively test all the possible paths through the system. Instead, it gives us a minimum number of tests

that we need to cover the structure of the system through component integration testing.

We will use the basis path/basis tests terminology to describe these tests.

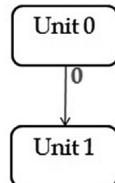


Figure 3–14 Unconditional call

The first design predicate (Figure 3–14) is an unconditional call. As shown here, it is designated by a straight line arrow between two modules. Since there is never a decision with an unconditional call, there is no increase in complexity due to it. This is designated by the zero (0) that is placed at the source side of the arrow connecting the modules.

Remember, it is decisions that increase complexity. Do we go here or there? How many times do we do that? In an unconditional call, there is no decision made—it always happens.

It is important to differentiate the integration testing from the functionality of the modules we are testing. The inner workings of a module might be extremely complex, with all kinds of calculations going on. For integration testing, those are all ignored; we are only interested in testing how the modules communicate and work together.

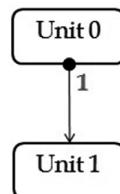


Figure 3–15 Conditional call

We might decide to call another module, or we might not. Figure 3–15 shows the conditional call, where an internal decision is made as to whether we will call the second unit. For integration testing, again, we don't care how the decision is made. Because it is a possibility only that a call may be made, we say that the complexity goes up to one (1). Note that the arrow now has a small filled-in

circle at the tail (source) end. Once again, we show the complexity increase by placing a 1 by the tail of the arrow.

It is important to understand that number. The complexity is not one—it is an increase of one. Suppose this graph represented the entire system. We have an increase of complexity of one—but the question is an increase from what? One way to look at it is to say that the first test is free, as it were. So the unconditional call does not increase the complexity, and we would need one test to cover it. One test is the minimum—would you ever feel free to test zero times? If you have gotten this far in the book, we would hope the answer is a resounding NO!

In this case, we start with that first test. Then, because we have an increase of one, we would need a second test. As you might expect, one test is where we call the module, the other is where we do not.

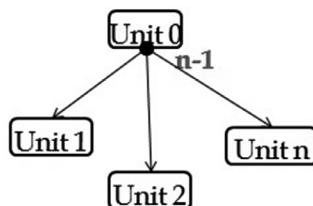


Figure 3–16 Mutually exclusive conditional call

The third design predicate (Figure 3–16) is called a mutually exclusive conditional call. This happens when a module will call one—and only one—of a number of different units. We are guaranteed to make a call to one of the units; which one will be called will be decided by some internal logic to the module.

In this structure, it is clear that we will need to test all of the possible calls at one time or another; that means there will be a complexity increase. This increase in complexity can be calculated based on the number of possible targets; if there are three targets as shown, the complexity increase would be two (2); calculated by the number of possibilities minus one. Note in the graph, a filled-in circle at the tail of the arrows shows that some of the targets will not be called.

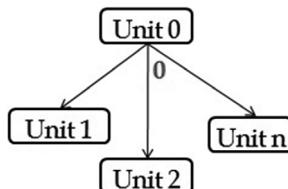


Figure 3–17 Unconditional calls

In the next graph (Figure 3–17), we show something that looks about the same. It is important, however, to see the difference. Without the dot in the tail, there is no conditional. That means that the execution of any test must include Unit 0 to Unit 1 execution as well as Unit 0 to Unit 2 execution and Unit 0 to Unit n at one time or another. This would look much clearer if it were drawn with the three arrows each touching Unit 0 in different places instead of converging to one place. No matter how it is drawn, however, the meaning is the same. With no conditional dot, these are unconditional connections; hence there is no increase in complexity.

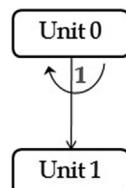


Figure 3–18 Iterative call

In (Figure 3–18) we have the iterative call. This occurs when a unit is called at least once but may be called multiple times. This is designated by the arcing arrow on the source module in the graph. In this case, the increase in complexity is deemed to be one, as shown in the graph.

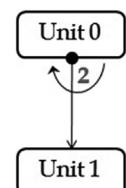


Figure 3–19 Iterative conditional call

Last but not least, we have the iterative conditional call. As seen in this last graph (Figure 3–19), if we add a conditional signifier to the iterative call, it increases the complexity by one. That means Unit 0 may call Unit 1 zero times, one time, or multiple times. Essentially, this should be considered exactly the same as the way we treated loop coverage in Chapter 2.

3.2.4.3 Hex Converter Example

In several different exercises and examples earlier in the book, you saw the hex converter code in Figure 3–20.

```
1.     jmp_buf sjbuf;
2.     unsigned long int hexnum;
3.     unsigned long int nhex;
4.
5.     main()
6.     /* Classify and count input chars */
7.     {
8.         int c, gotnum;
9.         void pophdigit();
10.
11.        hexnum = nhex = 0;
12.
13.        if (signal(SIGINT, SIG_IGN) != SIG_IGN) {
14.            signal(SIGINT, pophdigit);
15.            setjmp(sjbuf);
16.        }
17.        while ((c = getchar()) != EOF) {
18.            switch (c) {
19.                case '0':case '1':case '2':case '3':case '4':
20.                case '5':case '6':case '7':case '8':case '9':
21.                    /* Convert a decimal digit */
22.                    nhex++;
23.                    hexnum *= 0x10;
24.                    hexnum += (c - '0');
25.                    break;
26.                case 'a': case 'b': case 'c':
27.                case 'd': case 'e': case 'f':
28.                    /* Convert a lower case hex digit */
29.                    nhex++;
30.                    hexnum *= 0x10;
31.                    hexnum += (c - 'a' + 0xa);
32.                    break;
33.                case 'A': case 'B': case 'C':
34.                case 'D': case 'E': case 'F':
35.                    /* Convert an upper case hex digit */
36.                    nhex++;
37.                    hexnum *= 0x10;
38.                    hexnum += (c - 'A' + 0xA);
39.                    break;
40.                default:
41.                    /* Skip any non-hex characters */
42.                    break;
43.            }
44.        }
45.        if (nhex == 0) {
46.            fprintf(stderr,"hxcvt: no hex digits to convert!\n");
47.        } else {
48.            printf("Got %d hex digits: %x\n", nhex, hexnum);
49.        }
50.
51.        return 0;
52.    }
53.    void pophdigit()
54.    /* Pop the last hex input out of hexnum if interrupted */
55.    {
56.        signal(SIGINT, pophdigit);
57.        hexnum /= 0x10;
58.        nhex--;
59.        longjmp(sjbuf, 0);
60.    }
61.
```

Figure 3–20 Enhanced hex code converter code

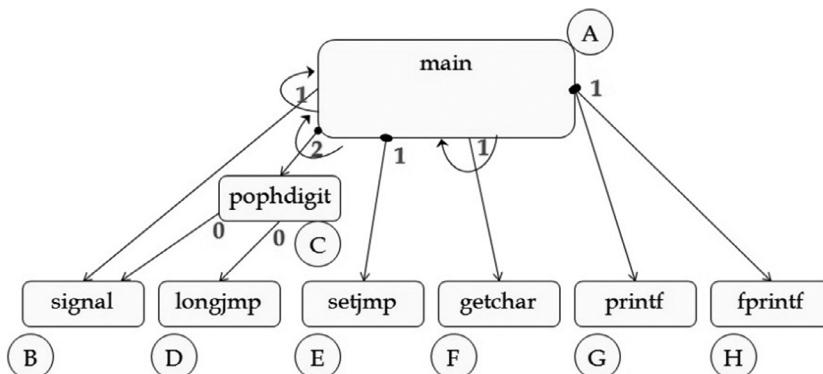
This code is for UNIX or Linux; it will not work with Windows without modification of the interrupt handling. Here is a complete explanation of the code, which we will use for our integration testing example.

When the program is invoked, it calls the *signal()* function. If the return value tells *main()* that **SIGINT** is not currently ignored (i.e., the process is running in the foreground), then *main()* calls *signal* again to set the function *pophdigit()* as the signal handler for **SIGINT**. *main()* then calls *setjmp()* to save the program condition in anticipation of a *longjmp()* back to this spot. Note that when we get ready to graph this, *signal* is definitely called once. After that, it may or may not be called again to set the return value.

main() then calls *getchar()* at least once. If upon first call an **EOF** is returned, the loop is ignored and *fprintf()* is called to report the error. If not **EOF**, the loop will be executed. All legal hex characters (A..F, a..f, 0..9) will be translated to a hex digit and added to the hex number as the next logical digit. In addition, a counter will be incremented for each hex char received. This continues to pick up input chars until the **EOF** is found.

If an interrupt (*Ctrl-C*) is received, it will call the *pophdigit()* routine, which will pop off the latest value and decrement the hex digit count.

What we would like to do with this is figure out the minimum number of tests we need for integration testing of it.



$$\text{Integration Complexity} = \text{Sum}(Design\ Predicates) + 1$$

$$IC = (1 + 2 + 1 + 1 + 1 + 0 + 0) + 1$$

$$IC = 7$$

Figure 3–21 Enhanced hex converter integration graph

Figure 3–21 is the call graph for the enhanced hex converter code. Let's walk through it from left to right.

Module *main* (A) calls *signal* (B) once with a possibility of a second call. That makes it iterative, with an increase in complexity of 1.

The function *popdigit()* (C) may be called any number of times; each time there is a signal (*Ctrl-C*), this function is called. When it is called, it always calls the *signal* (B) function. Since it might be called 0 times, 1 time, or multiple times, the increase in complexity is 2.

The function *setjmp* (E) may occur once in that case when *signal* (B) is called twice. That makes it conditional with an increase in complexity of 1.

The function *getchar* (F) is guaranteed to be called once and could be called any number of times. That makes it iterative with an increase in complexity of 1.

One of the functions, *printf* (G) or *fprintf* (H), will be called but not the other. That makes it a mutually exclusive conditional call. Since there are two possibilities, the increase in complexity is (N - 1), or 1.

Finally, the function *popdigit* (C) always calls *signal* (B) and *longjmp* (D), so each of those has an increase in complexity of 0.

The integration complexity is seven, so we need seven distinct test cases, right? Not necessarily! It simply means that there are seven separate paths that must be covered. If that sounds confusing, well, this graph is different than the directed graphs that we used when looking at cyclomatic complexity. When using a call graph, you must remember that after the call is completed, the thread of execution goes back to the calling module.

So, when we start out, we are executing in *main()*. We call the *signal()* function, which returns back to *main()*. Depending on the return value, we may call *signal()* again to set the handler, and then return to *main()*. Then, if we called *signal()* the second time, we call *setjmp()* and return back to *main()*. This is not a directed graph where you only go in one direction and never return to the same place unless there is a loop.

Therefore, several basis paths can be covered in a single test. You might ask whether that is a good or bad thing. We have been saying all along that fewer tests are good because we can save time and resources. Well, Okay, that was a straw man, not really what we have been saying. The refrain we keep coming back to is fewer tests are better if they give us the amount of testing we need based on the context of the project.

Fewer tests may cause us to miss some subtle bugs. Jamie remembers going to a conference once and sitting through a presentation by a person who was brand-new to requirements-based testing. This person clearly had not really gotten the full story on RBT, because he kept on insisting that as long as there was one test per requirement, then that was enough testing. When asked if some

requirements might need more than one test, he refused to admit that might even be possible.

A good rule of thumb is the more complex the software, the bigger the system, the more difficult the system is to debug, the more test cases you should plan on running—even if the strict minimum is fewer tests. Look for interesting interactions between modules, and plan on executing more iterations and using more and different data.

For now, let's assume that we want the minimum number of tests. We always like starting with a simple test to make sure some functionality works—a “Hello World” type test.

1. We want to input just an “A” while running the application in the foreground to make sure we get an output. We would expect it to test the following path: *ABBEFFG* and give an output of “*a*”. This will test the paths between *main()*, *signal()*, *setjmp()*, *getchar()*, and *printf()*.
2. Our second test is to make sure the system works when no input is given. This test will invoke the program but input an immediate *EOF* without any other characters: *ABFH*. We would expect that it would exercise the *main()*, *signal()*, *getchar()*, and, differently this time, the *fprintf()*. Output should be the no input message.
3. Our third test would be designed to test the interrupt handler. The input would be *F5^CT9a*. The interrupt is triggered by typing in the *Ctrl-C* keys together (shown here by the caret-C). Note that we have also included a non-hex character to make sure it gets sloughed off. The paths covered should be *ABBEFFFBCDFFFFG* and should execute in the following order: *main()*, *signal*, *signal*, *setjmp()*, *getchar()*, *getchar()*, *getchar()*, *popdigit()*, *longjmp()*, *getchar()*, *getchar()*, *getchar()*, *printf()*. We would expect an output of “*f9a*”.

At this point, we have executed each one of the paths at least once. But have we covered all of the design predicates? Notice that the connection from *main* (A) to *popdigit* (C) is an iterative, conditional call. We have tested it 0 times and 1 time, but not iteratively. So, we need a fourth test.

4. This test is designed to test the interrupt handler multiple times. Ideally we would like to send a lot of characters at it with multiple *Ctrl-C* (signal) characters. Our expected output would be all hex characters; the number of them would be the number inputted less the number of *Ctrl-C*s that were inputted.

Would we want to test this further? Sure. For example, we have not tested the application running in the background. That is the only way to get the SIGNAL to be ignored. We would have to use an input file and feed it to the application through redirection.¹²

In working with this example, we found a really subtle bug. Try tracing out what happens if the first inputted character is a Ctrl-C signal. In addition, the accumulator is defined as an unsigned long int; we wonder what happens when we input more characters than it can hold?

Typically, we want to start with the minimum test cases as sort of a smoke test and then continue with interesting test cases to check the nuances. Your mileage may vary!

3.2.4.4 McCabe Design Predicate Exercise

Calculate the integration complexity of the following call graph.

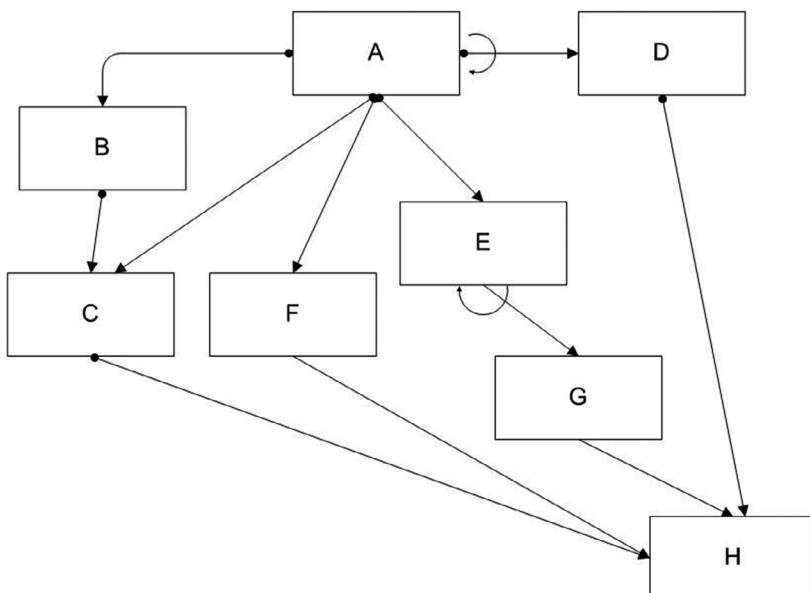


Figure 3–22 Design predicate exercise

12. The command is `hexcvt <file1> >file2 &`. That assumes that the compiled executable is called `hexcvt`, that `file1` contains a string of characters, and that `file2` will be used to catch the output. The ampersand makes it run in the background. In case you wanted to know...

3.2.4.5 McCabe Design Predicate Exercise Debrief

We tend to try to be systematic when going through a problem like this. We have modules A–H, each one has zero to five arrows coming from it. Our first pass through we just try to capture that information. We should have as many predicates as arrows. Then, for each one, we identify the predicate type and calculate the increase of paths needed. Don't forget that the first test is free!

Module A

- A to B: Conditional call (+1)
- A to C, A to F, A to E: Mutually exclusive conditional call ($n-1$, therefore +2)
- A to D: Iterative conditional call (+2)

Module B

- B to C: Conditional call (+1)

Module C

- C to H: Conditional call (+1)

Module D

- D to H: Conditional call (+1)

Module E

- E to G: Iterative call (+1)

Module F

- F to H: Unconditional call (+0)

Module G

- G to H: Unconditional call (+0)

Therefore, the calculation is as follows:

$$IC = (1 + 2 + 2 + 1 + 1 + 1 + 1 + 1) == 10$$

3.3 Dynamic Analysis

In the previous section, we discussed static analysis. In this section, we will discuss dynamic analysis. As the name suggests, this is something that is done while the system is executing. Dynamic analysis always requires instrumentation of some kind. In some cases, a special compiler creates the build, putting special code in that writes to a log. In other cases, a tool is run concurrently with the system under test; the tool monitors the system as it runs, sometimes reporting in real time and almost always logging results.

Dynamic analysis is a solution to a set of common problems in computer programs. While the system is executing, a failure occurs at a point in time, but there are no outward symptoms of the failure for the user to see. *If a tree falls in the woods and no one hears it, does it make a sound?* We don't know if a failure that we don't perceive right away makes a sound, but it almost always leaves damage behind. The damage may be corrupted data, which could be described as a land mine waiting for a later user to step on, a general slowdown of the system, or an upcoming blue screen of death; there are a lot of eventual symptoms possible.

So what causes these failures? Here are some possibilities.

It may be a minor memory leak where a developer forgets to deallocate a small piece of memory in a function that is run hundreds of times a minute. Each leak is small, but the sum total is a crash when the system runs out of RAM a few hours down the road. Historically, even some Microsoft Windows operating system calls could routinely lose a bit of memory; every little bit hurts.

It may be a wild pointer that changes a value on the stack erroneously; the byte(s) that were changed may be in the return address so that when the jump to return to the caller is made, it jumps to the wrong location, ensuring that the thread gets a quick trip to never-never land.

It may be an API call to the operating system that has the wrong arguments—or arguments in the wrong order—so the operating system allocates too small a buffer and the input data from a device is corrupted by being truncated.

The fact is there are likely to be an uncountable number of different failures that could be caused by the smallest defects. Some of those bugs might be in the libraries the programmers use, the programming languages they write in, or the compilers they build with.

Dynamic analysis tools work by monitoring the system as it runs. Some dynamic analysis tools are intrusive; that is, they cause extra code to be inserted right in the system code, often by a special compiler. These types of tools tend

ISTQB Glossary

memory leak: A memory access failure due to a defect in a program's dynamic store allocation logic that causes it to fail to release memory after it has finished using it, eventually causing the program and/or other concurrent processes to fail due to lack of memory.

dynamic analysis: The process of evaluating behavior, e.g., memory performance, CPU usage, of a system or component during execution.

wild pointer: A pointer that references a location that is out of scope for that pointer or that does not exist.

to be logging tools. Every module, every routine gets some extra logging code inserted during the compile. When a routine starts executing, it writes a message to the log, essentially a "Kilroy was here" type message.

The tool may cause special bit patterns to be automatically written to dynamically allocated memory. Jamie remembers wondering what a *DEADBEEF* was when he first started testing, because he kept seeing it in test results. It turns out that *DEADBEEF* is a 32-bit hex code that the compiler generated to fill dynamically allocated memory with; it allowed developers to find problems when the (re-)allocated heap memory was involved in anomalies. This bit pattern made it relatively simple when looking at a memory dump to find areas where the pattern is interrupted.

Other dynamic analysis tools are much more active. Some of them are initialized before the system is started. The system then (essentially) executes in a resource bubble supplied by the tool. When a wild pointer is detected, an incorrect API call is made, or when an invalid argument is passed in to a method, the tool determines it immediately. Some of these tools work with a MAP file (created by the compiler and used in debugging) to isolate the exact line of code that caused the failure. This information might be logged, or the tool might stop execution immediately and bring up the IDE with the module opened to the very line of code that failed.

Logging-type dynamic analysis tools are very useful when being run by testers; the logs that are created can be turned over to developers for defect isolation. The interactive-type tools are appropriate when the developers or skilled technical testers are testing their own code.

These tools are especially useful when failures occur that cannot be replicated, because they save data to the log that indicates what actually happened.

Even if we cannot re-create the failure conditions, we have a record of them. And, by capturing the actual execution profile in logs, developers often glean enough information that they can improve the dynamic behavior of the runtime system.

Dynamic analysis tools can be used by developers during unit and integration testing and debugging. We have found them very useful for testers to use during system testing. Because they may slow down the system appreciably, and because they are sometimes intrusive to the extent of changing the way a system executes, we don't recommend that they be used in every test session. There certainly is a need to execute at least some of the testing with the system configured the way the users will get it. But some testing, especially in early builds, can really benefit from using dynamic analysis tools.

There are some dynamic tools that are not intrusive. These do not change the system code; instead, they sit in memory and extrapolate what the system is doing by monitoring the memory that is being used during execution. While these type of tools tend to be more expensive, they are often well worth the investment.

All of these tools generate reports that can be analyzed after the test run and turned over to developers for debugging purposes.

These tools are not perfect. Because many of them exhibit the probe effect—that is, they change the execution profile of the system—they do force more testing to be done. Timing and profile testing certainly should not be performed with these tools active because the timing of the system can change radically. And, some of these tools require the availability of development artifacts, including code modules, MAP files, and so on.

In our experience, the advantages of these tools generally far outweigh the disadvantages. Let's look at a few different types of dynamic analysis tools.

3.3.1 Memory Leak Detection

Memory leaks are a critical side effect of developer errors when working in environments that allow allocation of dynamic memory without having automatic garbage collection. Garbage collection means that the system automatically recovers allocated memory once the developer is done with it. For example, Java has automatic garbage collection, while standard C and C++ compilers do not.

Memory can also be lost when operating system APIs are called with incorrect arguments or out of order. There have been times when compilers

generated code that had memory leaks on their own; much more common, however, is for developers to make subtle mistakes that cause these leaks.

The way we conduct testing is often not conducive to finding memory leaks without the aid of a dynamic analysis tool. We tend to start a system, run it for a relatively short time for a number of tests, and then shut it down. Since memory leaks tend to be a long-term issue, we often don't find them during normal testing. Customers and users of our systems do find them because they often start the system and run it 24/7, week after week, month after month. What might be a mere molehill in the test lab often becomes a mountain in production.

A dynamic analysis tool that can track memory leaks generally monitors both the allocation and deallocation of memory. When a dynamically allocated block of memory goes out of scope without being explicitly deallocated, the tool notes the location of the leak. Most tools then write that information to a log; some might stop the execution immediately and go to the line of code where the allocation occurred.

All of these tools write voluminous reports that allow developers to trace the root cause of failures.

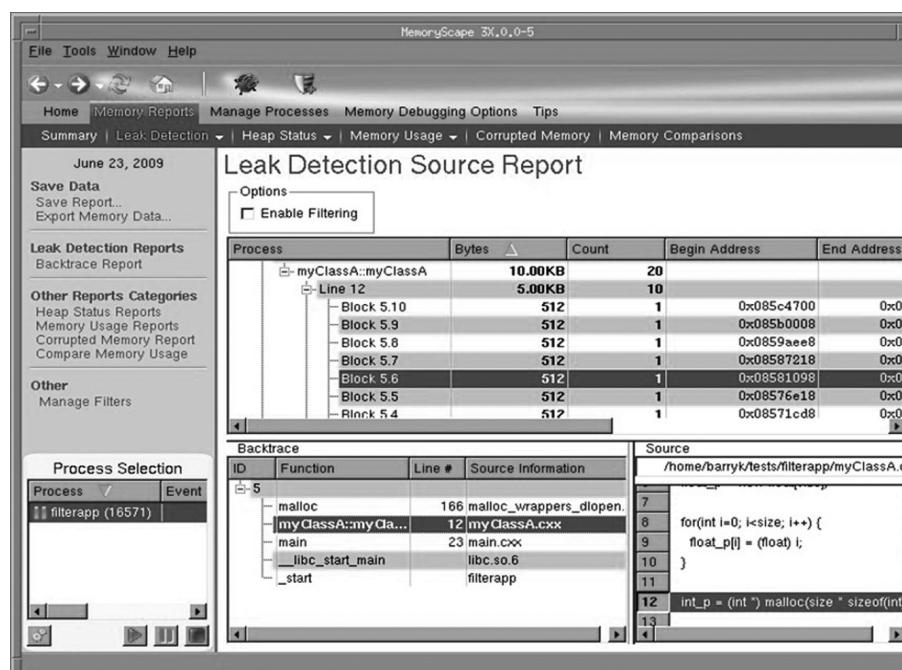


Figure 3–23 Memory leak logging file

Figure 3–23 shows the output of such a tool. For a developer, this is very important information. It shows the actual size of every allocated memory block that was lost, the stack trace of the execution, and the line of code where the memory was allocated. A wide variety of reports can be derived from this information:

- Leak detection source report (this one)
- Heap status report
- Memory usage report
- Corrupted memory report
- Memory usage comparison report

Some of this information is available only when certain artifacts are available to the tool at runtime. For example, a MAP file and a full debug build would be needed to get a report this detailed.

3.3.2 Wild Pointer Detection

Another major cause of failures that occur in systems written in certain languages is pointer problems. A pointer is an address of memory where something of importance is stored. C, C++, and many other programming languages allow users to access these with impunity. Other languages (Delphi, C#, Java) supply functionality that allows developers to do powerful things without explicitly using pointers. Some languages do not allow the manipulation of pointers at all due to their inherent danger.

The ability to manipulate memory using pointers gives a programmer a lot of power. But, with a lot of power comes a lot of responsibility. Misusing pointers causes some of the worst failures that can occur.

Compilers try to help the developers use pointers more safely by preventing some usage and warning about others; however, the compilers can usually be overridden by a developer who wants to do a certain thing. The sad fact is, for each correct way to do something, there are usually many different ways to do it poorly. Sometimes a particular use of pointers appears to work correctly; only later do we get a failure when the (in)correct set of circumstances occurs.

The good news is that the same dynamic tools that help with memory leaks can help with pointer problems.

Some of the consequences of pointer failures are listed here:

1. Sometimes we get lucky and nothing happens. A wild pointer corrupts something that is not used throughout the rest of the test session. Unfortu-

nately, in production we are not this lucky; if you damage something with a pointer, it usually shows a symptom eventually.

2. The system might crash. This might occur when the pointer trashes an instruction of a return address on the stack.
3. Functionality might be degraded slightly—sometimes with error messages, sometimes not. This might occur with a gradual loss of memory due to poor pointer arithmetic or other sloppy usage.
4. Data might be corrupted. Best case is when this happens in such a gross way that we see it immediately. Worst case is when that data is stored in a permanent location where it will reside for a period before causing a failure that affects the user.

Short of preventing developers from using pointers (not likely), the best prevention of pointer-induced failures is the preventive use of dynamic analysis tools.

3.3.3 Dynamic Analysis Exercise

1. Read the HELLOCARMS system requirements document.
2. Outline ways to use dynamic analysis during system testing.
3. Outline ways to use dynamic analysis during system integration testing.

3.3.4 Dynamic Analysis Exercise Debrief

Your solution should include the following main points:

- Memory leak detection tools can be used during both system test and system integration test, on at least two servers on which HELLOCARMS code runs (web server and application server). The database server might not need such analysis.
- Assuming that a programming language that allows pointer usage (e.g., C++) is used on some or all parts of the system, wild pointer tools might be used. These would be deployed during selected cycles of system test and system integration test, especially if any odd behaviors or illegal memory access violations are observed. As with memory leak detection, the main targets would be the web server and application server. The database server would only need such analysis if code that uses pointers is written for it.
- Performance analysis tools should definitely be used during system test and system integration test, and in this case on all three servers.
- API tracking tools could certainly be used during both cycles of test. During system test, these could find anomalies in the way the application works with the local operating system. During system integration, it would be able to

look for irregularities in the API, RPC, and any middleware function calls that occur on the distributed system.

You might also identify additional opportunities for use of dynamic analysis.

3.4 Sample Exam Questions

1. Given the following:

- I. Changing the index variable from the middle of a loop
- II. Using a neighborhood rather than pairwise relationship
- III. Jumping to the inside of a loop using a GOTO
- IV. Non-deterministic target of a Scheme function call
- V. Avoiding the use of a McCabe design predicate
- VI. Cyclomatic complexity of 18
- VII. Essential complexity of 18
- VIII. Use-use relationship of a passed in argument

Which of the possible defects or design flaws above can be detected by control flow analysis?

- A. All of them
 - B. II, IV, V, VII, VIII
 - C. I, III, IV, VI, VII
 - D. I, III, V, VI, VII
2. Your flagship system has been experiencing some catastrophic failures in production. These do not occur often; sometimes a month can go between failures. Unfortunately, to this point the test and support staff have not been able to re-create the failures. Investigating the issue, you have not found any proof that specific configurations have caused these problems; failures have occurred in almost all of the configurations. Which of the following types of testing do you believe would have the most likely chance of being able to solve this problem?
- A. Soak-type performance testing
 - B. Keyword-driven automated testing
 - C. Static analysis
 - D. Dynamic analysis

3. Your organization has hired a new CIO due to several poor releases in the past year. The CIO came from a highly successful software house and wants to make immediate changes to the processes currently being used at your company. The first process change to come down is a hard and fast rule that all code must be run through a static analysis tool and all errors and warnings corrected before system test is complete. As the lead technical test analyst for the current project, which of the following options best describes the explanation you should give to the CIO about why this new rule is a bad idea?
- A. Your staff has no training on these tools.
 - B. There is no budget for buying or training on these tools.
 - C. Changing existing, working code based on warnings from static analysis tools is not wise.
 - D. Given the current work load, the testers do not have time to perform static testing.
4. The following code snippet reads through a file and determines whether the numbers contained are prime or not:

```
1  Read (Val);
2  While NOT End of File Do
3      Prime := TRUE;
4      For Holder := 2 TO Val DIV 2 Do
5          If Val - (Val DIV Holder)*Holder= 0 Then
6              Write (Holder, ` is a factor of', Val);
7              Prime := FALSE;
8          Endif;
9      Endfor;
10     If Prime = TRUE Then
11         Write (Val , ` is prime');
12     Endif;
13     Read (Val);
14 Endwhile;
15 Write('End of run)
```

Calculate the cyclomatic complexity of the code.

- A. 3
- B. 5
- C. 7
- D. 9

5. When you're calculating complexity of a function, which of the following control structures is most likely to give a misleading cyclomatic complexity value?
 - A. Switch/case statement
 - B. Embedded loop inside another loop
 - C. While loop
 - D. For statement
6. A code module is measured and determined to have a cyclomatic complexity of 16. Under which of the following circumstances would the development group be least likely to refactor the module into two different modules?
 - A. They are building a mission-critical application.
 - B. The project time constraints are very tight.
 - C. They are building a real-time critical application.
 - D. They are building a safety-critical application.

7. Consider the following code snippet:

```
1. #include<windows.h>
2. #include<wininet.h>
3. #include<stdio.h>
4. int main()
5. {
6.
7.     HINTERNET Initialize, Connection, File;
8.     DWORD dwBytes;
9.     char ch;
10.    Connection = InternetConnect(Initialize, "www.xxx.com",
11.                                INTERNET_DEFAULT_HTTP_PORT, NULL, NULL,
12.                                INTERNET_SERVICE_HTTP, 0, 0);
13.
14.    File = HttpOpenRequest(Connection, NULL, "/index.html",
15.                           NULL, NULL, NULL, 0, 0);
16.
17.    if(HttpSendRequest(File, NULL, 0, NULL, 0))
18.    {
19.        while(InternetReadFile(File, &ch, 1, &dwBytes))
20.        {
21.            if(dwBytes != 1)break;
22.            putchar(ch);
23.        }
24.    }
25.    InternetCloseHandle(File);
26.    InternetCloseHandle(Connection);
27.    InternetCloseHandle(Initialize);
28.    return 0;
29. }
```

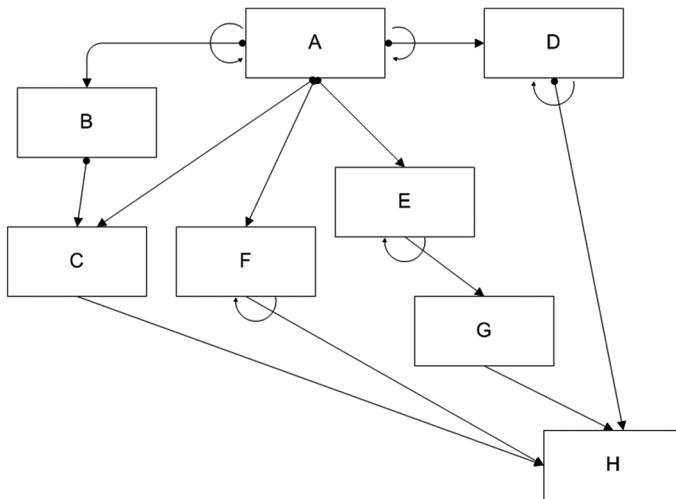
With regards to the variable *Connection* and looking at lines 10–14, what kind of data flow pattern do we have?

- A. du
- B. ud
- C. dk
- D. There is no defined data flow pattern there.

8. Which of the following should always be considered a possible serious defect?
 - A. ~u
 - B. dk
 - C. kk
 - D. dd
9. Your team has decided to perform data flow analysis in the software you are developing to be placed in a satellite system. Which of the following defect would be least likely found using this analysis technique?
 - A. Failure to initialize a variable that is created on the stack as a local variable.
 - B. A race condition occurs where an array value is changed by another thread unexpectedly.
 - C. An allocated heap variable is killed inside a called function because a break statement is missing.
 - D. A passed-by-reference variable is unexpectedly changed before the original value is used.
10. Which of these data flow coverage metrics is the strongest of the four listed?
 - A. All-P uses
 - B. All-C uses
 - C. All Defs
 - D. All DU paths

11. You have been tasked by the QA manager to come up with some common sense standards and guidelines for programmers to follow when they are writing their code. Which of the following possible standards are likely to be included in the standards and guidelines?
- I. Every method requires a header (describing what the method does) and listing parameters.
 - II. Every variable must use Hungarian notation when it is named.
 - III. Comments must be meaningful and not just repeat what the code says.
 - IV. No module may have a cyclomatic complexity over 10 without formal review sign-off.
 - V. Static analysis tool must indicate high coupling and low cohesion of each module.
 - VI. Code that is copied and pasted must be marked by comments and change bars.
 - VII. No magic numbers may appear in the code; instead, named constants must be used.
- A. I, II, III, VI, VII
 - B. I, II, III, IV, VII
 - C. All of them
 - D. I, III, IV, V

12. Given the integration call graph below, and using McCabe's design predicate approach, how many basis paths are there to test?



- A. 12
B. 10
C. 13
D. 14
13. Given the following list, which of these are most likely to be found through the use of a dynamic analysis tool?
- I. Memory loss due to wild pointers
 - II. Profiling performance characteristics of a system
 - III. Failure to initialize a local variable
 - IV. Argument error in a Windows 32 API call
 - V. Incorrect use of equality operator in a predicate
 - VI. Failure to place a break in a switch statement
 - VII. Finding dead code
- A. I, III, IV, and VII
B. I, II, III, IV, and VI
C. I, II, and IV
D. II, IV, and V

4 Quality Characteristics for Technical Testing

Quality in a product or service is not what the supplier puts in. It is what the customer gets out and is willing to pay for. A product is not quality because it is hard to make and costs a lot of money, as manufacturers typically believe. This is incompetence. Customers pay only for what is of use to them and gives them value. Nothing else constitutes quality.

–Peter F. Drucker

The fourth chapter of the Advanced Level Syllabus – Technical Test Analyst is primarily concerned with non-functional testing as defined in ISO 9126. In the previous ISTQB Advanced syllabus, we called these topics *quality attributes*.

In this chapter there are four common learning objectives that we will be covering in each applicable section. Therefore, we list those here to avoid listing them in each section.

Common Learning Objective

TTA-4.x.1 (K2) Understand and explain the reasons for including maintainability, portability, and resource utilization tests in a testing strategy and/or test approach.

TTA-4.x.2 (K3) Given a particular product risk, define the particular non-functional test type(s) which are most appropriate.

TTA-4.x.3 (K2) Understand and explain the stages in an application's life cycle where non-functional tests should be applied.

TTA-4.x.4 (K3) For a given scenario, define the types of defects you would expect to find by using non-functional testing types.

Beyond the common learning objectives, we shall cover seven sections, as follows:

1. Introduction
2. Security Testing
3. Reliability Testing
4. Efficiency Testing
5. Maintainability Testing
6. Portability Testing
7. General Planning Issues

4.1 Introduction

Learning objectives

Recall of content only.

In this chapter, we will discuss testing many of the various quality characteristics with which technical test analysts (TTAs) are expected to be familiar. While most of the topics we will discuss fall under the heading of nonfunctional testing, ISTQB also adds security testing, which is considered—by ISO 9126—as a subcharacteristic of functionality testing. Security testing often falls to the TTA due to its intrinsic technical aspects.

Many of the test design techniques used in nonfunctional testing are discussed in the ISTQB Foundation syllabus. Other design techniques and a deeper discussion of the techniques introduced in the Foundation syllabus can be found in the companion book to this one, *Advanced Software Testing, Vol. 1*. A common element between the test analyst and technical test analyst specialties is that both require understanding the quality characteristics—that really matter to our customers—in order to recognize typical risks, develop appropriate testing strategies, and specify effective tests.

As shown in Table 4–1, the ISTQB Advanced syllabi assign certain topics to test analysts, and others to technical test analysts. Despite the fact that ISO 9126 has been replaced by ISO 25000, ISTQB has decided to continue using ISO 9126 as a reference because the changes do not affect our testing discussion and 9126 is still widely used in the testing community.

Table 4-1 ISO-9126 testing responsibilities

Characteristic	Subcharacteristics	Test Analyst	Technical Test Analyst
Functionality	Accuracy, Suitability, Interoperability, Compliance	X	
	Security		X
Reliability	Maturity (Robustness), Fault-tolerance, Recoverability, Compliance		X
Usability	Understandability, Learnability, Operability, Attractiveness, Compliance	X	
Efficiency	Performance (Time behavior), Resource Utilization, Compliance		X
Maintainability	Analyzability, Changeability, Stability, Testability, Compliance		X
Portability	Adaptability, Installability, Co-existence, Replaceability, Compliance		X

Since we will be covering these topics in this chapter, it is important to understand exactly what the ISO 9126 standard entails. There are four separate sections:

1. The first section, ISO 9126-1, is the quality model itself, enumerating the six quality characteristics and the subcharacteristics that go along with them.
2. ISO 9126 Part 2 references external measurements that are derived by dynamic testing. Sets of metrics are defined for assessing the quality subcharacteristics of the software. These external metrics can be calculated by mining the incident tracking database and/or making direct observations during testing.
3. ISO 9126 Part 3 references internal measurements that are derived from static testing. Sets of metrics are defined for assessing the quality subcharacteristics of the software. These measurements tend to be somewhat abstract because they are often based on estimates of likely defects.
4. ISO 9126 Part 4 is for quality-in-use metrics. This defines sets of metrics for measuring the effects of using the software in a specific context of use. We will not refer to this portion of the standard in this book.

Each characteristic in ISO-9126 contains a number of subcharacteristics. We will discuss most of them. One specific subcharacteristic that appears under each main characteristic is *compliance*. This subcharacteristic is not well explained in ISO 9126 because, in each case, it is a measurement of how compliant the system is with applicable regulations, standards, and conventions. While there are a few metrics to measure this compliance—and we will discuss those—the standard does not describe to which regulations, standards, and conventions it refers. Since this is an international standard, that should not be surprising.

For each compliance subcharacteristic, ISO 9126 has almost exactly the same text (where XXXX is replaced by the quality characteristic of your choice):

Internal (or external) compliance metrics indicate a set of attributes for assessing the capability of the software product to comply to such items as standards, conventions, or regulations of the user organization in relation to XXXX.

It is up to each organization, and to its technical test analysts, to determine which federal, state, county, and/or municipality standards apply to the system they are building. Because these can be quite diverse, we will emulate ISO 9126 and not discuss compliance further beyond the metrics mentioned earlier.

ISO 9126 lists a number of metrics that could be collected from both static (internal) and dynamic (external) testing. Some of the internal metrics are somewhat theoretical and may be problematic for less mature organizations (for example, how many bugs were actually found in a review compared to how many were estimated to be there). Many of these metrics would require a very high level of organizational maturity to track. For example, several of them are based on the “number of test cases required to obtain adequate test coverage” from the requirements specification documentation. What is adequate coverage? ISO 9126 does not say. That is up to each organization to decide. It is Jamie’s experience that the answer to that question is often vaporous.

Question: What is adequate? Answer: Enough!¹

Jamie and Rex have followed different career paths and have different experiences when it comes to static testing. While Jamie has worked for some organizations that did some static testing, they were more the exception than the rule. The ones that did static testing did not track many formal metrics from that test-

1. Usually followed with an expansive arm movement, a knowing wink, and a quick run toward the door.

ing. Rex, on the other hand, has worked with far more organizations that have performed static testing and calculated metrics from the reviews (what ISO 9126 refers to as internal metrics).

After each section, we will list some of the internal and external metrics to which ISO 9126 refers. We are not claiming that these will always be useful to track; each organization must make decisions as to which metrics they believe will give them value. We are listing them because the authors of the ISO 9126 standard believed that they can add *some* value to some organizations *some* times, and we believe they may add something meaningful to the discussion.

4.2 Security Testing

Learning objectives

TTA-4.3.1 (K3) Define the approach and design high-level test cases for security testing.

Security testing is often a prime concern for technical test analysts. Because so often security risks are either hidden, subtle, or side effects of other characteristics, we should put special emphasis on testing for security risks.

Typically, other types of failures have symptoms that we can find, through either manual testing or the use of tools. We know when a calculation fails—the erroneous value is patently obvious. Security issues often have no symptoms, right up until the time a hacker breaks in and torches the system. Or, maybe worse, the hacker breaks in, steals critical data, and then leaves without leaving a trace. Ensuring that people can't see what they should not have access to is a major task of security testing.

The illusion of security can be a powerful deterrent to good testing since no problems are perceived. The system appears to be operating correctly. Jamie was once told to “stop wasting your time straining at gnats” when he continued testing beyond what the project manager thought was appropriate. When they later got hacked, the first question on her lips was how did he miss that security hole?

Another deterrent to good security testing is that reporting bugs and the subsequent tightening of security may markedly decrease perceived quality in performance, usability, or functionality; the testers may be seen as worrying too much about minor details, hurting the project.

Not every system is likely to be a target, but the problem, of course, is trying to figure out which ones will be. It is simple to say that a banking system, a

ISTQB Glossary

security testing: Testing to determine the security of the software product.

university system, or a business system is going to be a target. But some systems might be targeted for political reasons or monetary reasons, and many might just be targeted for kicks. Vandalism is a growth industry in computers; some people just want to prove they are smarter than everyone else by breaking something with a high profile.

4.2.1 Security Issues

We are going to look at a few different security issues. This list will not be exhaustive; many security issues were discussed in *Advanced Software Testing: Volume 1* and we will not be duplicating those. Here's what we will discuss:

- Piracy (unauthorized access to data)
- Buffer overflow/malicious code insertion
- Denial of service
- Reading of data transfers
- Breaking encryption meant to hide sensitive materials
- Logic bombs/viruses/worms
- Cross-site scripting

As we get into this section, please allow us a single global editorial comment: security, like quality, is the responsibility of every single person on the project. All analysts and developers had better be thinking about it, not as an after-thought, but as an essential part of building the system. And every tester at every level of test had better consider security issues during analysis and design of their tests. If every person on the project does not take ownership of security, our systems are just not going to be secure.

4.2.1.1 Piracy

There are a lot of ways that an intruder may get unauthorized access to data.

SQL injection is a hacker technique that causes a system to run an SQL query in a case where it is not expected. Buffer overflow bugs, which we will discuss in the next section, may allow this, but so might intercepting an authorized SQL statement that is going to be sent to a web server and modifying it. For example, a query is sent to the server to populate a certain page, but a hacker modifies the underlying SQL to get other data back.

Passwords are really good targets for information theft. Guessing them is often not hard; organizations may require periodic update of passwords and the human brain is not really built for long, intricate passwords. So users tend to use patterns of keys (q-w-e-r-t-y, a-s-d-f, etc.). Often, they use their name, their birthday, their dog's name, or even just p-a-s-s-w-o-r-d. And, when forced to come up with a hard-to-remember password, they write it down. The new password might be found just underneath the keyboard or in their top-right desk drawer. Interestingly enough, Microsoft published a study that claims there is no actual value and often a high cost for changing passwords frequently.² From their mouths to our sysadmin's ear...

The single biggest security threat is often the physical location of the server. A closet, under the sysadmin's desk, or in the hallway are locations where we have seen servers, leaving access to whoever happens to be passing by.

Temp (and other) files and databases can be targets if they are unencrypted. Even the data in the EXE or DLL files might be discovered using a common binary editor.

Good testing techniques include careful testing of all functionality that can be accessed from outside. When a query is received from beyond the firewall, it should be checked to ensure that it is accessing only the expected area in the database management system (DBMS). Strong password control processes should be mandatory, with testing regularly occurring to make sure they are enforced. All sensitive data in the binaries and all data files should be encrypted. Temporary files should be obfuscated and deleted after use.

4.2.1.2 Buffer Overflow

It seems like every time anyone turns on their computer nowadays, they are requested to install a security patch from one organization or another. A good number of these patches are designed to fix the latest buffer overflow issue. A buffer is a chunk of memory that is allocated by a process to store some data. As such, it has a finite length. The problems occur when the data to be stored is longer than the buffer. If the buffer is allocated on the stack and the data is allowed to overrun the size of the buffer, important information also kept on the stack might also get overwritten.

When a function in a program is called, a chunk of space on the stack, called a stack frame, is allocated. Various forms of data are stored there, including local

2. <http://microsoft-news.tmcnet.com/microsoft/articles/81726-microsoft-study-reveals-that-regular-password-changes-useless.htm>

variables and any statically declared buffers. At the bottom of that frame is a return address. When the function is done executing, that return address is picked up and the thread of execution jumps to there. If the buffer overflows and that address is overwritten, the system will almost always crash because the next execution step is not actually executable. But suppose the buffer overflow is done skillfully by a hacker? They can determine the right length to overflow the buffer so they can put in a pointer to their own code. When the function returns, rather than going back to where it was called from, it goes to the line of code the hacker wants to run. Oops! You just lost your computer.

Anytime you have a data transfer between systems, buffer overflow can be a concern—especially if your organization did not write the connecting code. A security breach can come during almost any DLL, API, COM, DCOM, or RPC call. One well-known vulnerability was in the FreeBSD utility *setlocale()* in the libc module.³ Any program calling it was at risk of a buffer overflow bug.

All code using buffers should be statically inspected and dynamically stressed by trying to overflow it. Testers should investigate all available literature when developers use public libraries. If there are documented failures in the library, extra testing should be put into place. If that seems excessive, we suggest you check out the security updates that have been applied to your browser over the last year.

4.2.1.3 Denial of Service

Denial of service attacks are sometimes simply pranks pulled by bored high schoolers or college students. They all band together and try to access the same site intensively with the intent of preventing other users from being able to get in. Often, however, these attacks are planned and carried out with overwhelming force. For example, recent military invasions around the world have often started with well-planned attacks on military and governmental computer facilities.⁴ Often, unknown perpetrators have attacked specific businesses by triggering denial of service attacks by “bots,” zombie machines that were taken over through successful virus attacks.

The intent of such an attack is to cause severe resource depletion of a website, eventually causing it to fail or slow down to unacceptable speeds.

3. <http://security.freebsd.org/advisories/FreeBSD-SA-00:53.catopen.asc>

4. Just one example: <http://defensetech.org/2008/08/13/cyber-war-2-0-russia-v-georgia/>

A variation on this is where a single HTTP request might be hacked to contain thousands of slashes, causing the web server to spin its wheels trying to decode the URL.

There is no complete answer to preventing denial of service attacks. Validation of input calls can prevent the attack using thousands of slashes. For the most part, the best an organization can do is try to make the server and website as efficient as possible. The more efficiently a site runs, the harder it will be to bring it down.

4.2.1.4 Data Transfer Interception

The Internet consists of a protocol whereby multiple computers transfer data to each other. Many of the IP packets that any given computer is passing back and forth are not meant for that computer itself; the computer is just acting as a conduit for the packets to get from here to there. The main protocol used on the Internet, HTTP, does not encrypt the contents of the packets. That means that an unscrupulous organization might actually save and store the passing packets and read them.

All critical data that your system is going to send over the Internet should be strongly encrypted to prevent peeking. Likewise, the HTTPS protocol should be used if the data is sensitive.

HTTPS is not infallible. It essentially relies on trust in a certification authority (VeriSign, Microsoft, or others) to tell us whom we can trust, and it also contains some kind of encryption. However, it is better and more secure than HTTP.

4.2.1.5 Breaking Encryption

Even with encryption, not all data will be secure. Weak encryption can be beaten through brute force attacks by anyone with enough computing power. Even when encryption is strong, the key may be stolen or accessed, especially if it is stored with the data. Because strong encryption tends to be time intensive, the amount and type used in a system may be subject to performance trade-offs. The technical test analysts should be involved in any discussions as to those design decisions.

Because encryption is mathematically intense, it usually can be beaten by better mathematical capabilities. In the United States, the National Security Agency (NSA) often has first choice of graduating mathematicians for use in their code/cipher breaking operations. If your data is valuable enough, there is

liable to be someone willing to try to break your security, even when the data is encrypted.

The only advice we can give for an organization is to use the strongest (legal) encryption that you can afford, never leave the keys where they can be accessed, and certainly never store the key with the data. Testers should include testing against these points in every project. Many security holes are opened when a “quick patch is made that won’t affect anything.”⁵

4.2.1.6 Logic Bombs/Viruses/Worms

We need to add to this short list of possible security problems the old standbys, viruses, worms, and logic bombs.

A logic bomb is a chunk of code that is placed in a system by a programmer and gets triggered when specific conditions occur. It might be there for a programmer to get access to the code easily (a back door), or it might be there to do specific damage. For example, in June 1992, an employee of the US defense contractor General Dynamics was arrested for inserting a logic bomb into a system that would delete vital rocket project data. It was alleged that his plan was to return as a highly paid consultant to fix the problem once it triggered.⁶

There are a great number of stories of developers inserting logic bombs that would attack in case of their termination from the company they worked for. Many of these stories are likely urban legends. Of much more interest to testers is when the logic bombs are inserted via viruses or worms.

Certain viruses have left logic bombs that were to be triggered on a certain date: April Fools’ Day or Friday the 13th are common targets.

Worms are self-replicating programs that spread throughout a network or the Internet. A virus must attach itself to another program or file to become active; a worm does not need to do that.

For testing purposes, we would suggest the best strategy is to have good antivirus software installed on all systems (with current updates, of course) and a strict policy of standards and guidelines for all users to prevent the possibility of infection.

To prevent logic bombs, all new, impacted, and changed code should be subjected to some level of static review.

5. A quote we have heard enough times to scare us!

6. <http://www.gfi.com/blog/insidious-threats-logical-bomb/>

4.2.1.7 Cross-Site Scripting

This particular vulnerability pertains mainly to web applications. Commonly abbreviated as XSS, this exposure allows an attacker to inject malicious code into the web pages viewed by other consumers of a website. When another user connects to the website, they get a surprise when they download what they thought were safe pages.

This vulnerability is unfortunately widespread; an attack can occur anytime a web application mixes input from a user with the output it generates without validating it or encoding it on its own.

Since the victim of a malicious script is downloading from a trusted site, the browser may execute it blindly. At that point, the evil code has access to cookies, session tokens, and other sensitive information kept by the browser on the client machine.

According to OWASP.org (see section 4.2.1.8 for a discussion), this is one of the most common security holes found on the Web today. Luckily, there are several ways for a development group to protect themselves from XSS; they can be found on the OWASP site.⁷

When you're testing a website, there are several things that should always be tested thoroughly:

- Look closely at GET requests. These are more easily manipulated than other calls (such as POSTs).
- Check for transport vulnerabilities by testing against these criteria:
 - Are session IDs always sent encrypted?
 - Can the application be manipulated to send session IDs unencrypted?
 - What cache-control directives are replied to calls passing session IDs?
 - Are such directives always present?
 - Is GET ever used with session IDs?
 - Can POST be interchanged with GET?

Black-box tests should always include the following three phases:

1. For each web page, find all of the web application's user-defined variables. Figure out how they may be inputted. These may include HTTP parameters, POST data, hidden fields on forms, and preset selection values for check boxes and radio buttons.

7. [https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)

2. Analyze each of the variables listed in step 1 by trying to submit a predefined set of known attack strings that may trigger a reaction from the client-side web browser being tested.
3. Analyze the HTML delivered to the client browser, looking for the strings that were inputted. Identify any special characters that were not properly encoded.

Given the space we can afford to spend on this topic, we can barely scratch the surface of this common security issue. However, the OWASP site has an enormous amount of information for testing this security issue and should be consulted; it always has the latest exploits that have been found on the Web.

4.2.1.8 Timely Information

Many years ago, Jamie was part of a team that did extensive security testing on a small business website. After he left the organization, he was quite disheartened to hear that the site had been hacked by a teenager. It seems like the more we test, the more some cretin is liable to break in just for the kicks.

We recently read a report that claimed a fair amount of break-ins were successful because organizations (and people) did not install security patches that were publicly available.⁸ Like leaving the server in an open, unguarded room, all of the security testing in the world will not prevent what we like to call stupidity attacks.

If you are tasked with testing security on your systems, there are a variety of websites that might help you with ideas and testing techniques. Access to timely information can help you from falling victim to a “me too” hacker who exploits known vulnerabilities.⁹ Far too often, damage is done by hackers because no one thought of looking to see where the known vulnerabilities were.

Figure 4–1 shows an example from the *Common Vulnerabilities and Exposures (CVE)* site.¹⁰ This international website is free to use; it is a dictionary of publically known security vulnerabilities. The goal of this website is to make it easier to share data about common software vulnerabilities that have been found.

8. http://www.sans.org/reading_room/whitepapers/windows/microsoft-windows-security-patches_273

9. Of course, this won’t help your organization if you are targeted by someone who invents the hack.

10. cve.mitre.org

Rank	Score	ID	Name
[1]	346	CWE-79	Failure to Preserve Web Page Structure ('Cross-site Scripting')
[2]	330	CWE-89	Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection')
[3]	273	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	261	CWE-352	Cross-Site Request Forgery ('CSRF')
[5]	219	CWE-285	Improper Access Control (Authorization)
[6]	202	CWE-802	Relyance on Untrusted Inputs in a Security Decision
[7]	197	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[8]	194	CWE-434	Unrestricted Upload of File with Dangerous Type
[9]	188	CWE-78	Improper Sanitization of Special Elements used in an OS Command ('OS Command Injection')
[10]	188	CWE-311	Missing Encryption of Sensitive Data
[11]	176	CWE-798	Use of Hard-coded Credentials
[12]	158	CWE-805	Buffer Access with Incorrect Length Value
[13]	157	CWE-98	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP File Inclusion')
[14]	156	CWE-129	Improper Validation of Array Index
[15]	155	CWE-754	Improper Check for Unusual or Exceptional Conditions
[16]	154	CWE-209	Information Exposure Through an Error Message
[17]	154	CWE-190	Integer Overflow or Wraparound
[18]	153	CWE-131	Incorrect Calculation of Buffer Size
[19]	147	CWE-306	Missing Authentication for Critical Function
[20]	146	CWE-494	Download of Code Without Integrity Check
[21]	145	CWE-732	Incorrect Permission Assignment for Critical Resource
[22]	145	CWE-770	Allocation of Resources Without Limits or Throttling
[23]	142	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[24]	141	CWE-322	Use of a Broken or Risky Cryptographic Algorithm
[25]	138	CWE-362	Race Condition

Figure 4–1 List of top 25 security vulnerabilities from the CVE website

The site contains a huge number of resources like this, a list of the top 25 programming errors of 2010. By facilitating information transfer between organizations, and giving common names to known failures, the organization running this website hopes to make the Internet a safer community.

Name	CWE-ID	Description
Authentication Issues	CWE-287	Failure to properly authenticate users.
Credentials Management	CWE-255	Failure to properly create, store, transmit, or protect passwords and other credentials.
Permissions, Privileges, and Access Control	CWE-264	Failure to enforce permissions or other access restrictions for resources, or a privilege management problem.
Buffer Errors	CWE-119	Buffer overflows and other buffer boundary errors in which a program attempts to put more data in a buffer than the buffer can hold, or when a program attempts to put data in a memory area outside of the boundaries of the buffer.
Cross-Site Request Forgery (CSRF)	CWE-352	Failure to verify that the sender of a web request actually intended to do so. CSRF attacks can be launched by sending a formatted request to a victim, then tricking the victim into loading the request (often automatically), which makes it appear that the request came from the victim. CSRF is often associated with XSS, but it is a distinct issue.
Cross-Site Scripting (XSS)	CWE-79	Failure of a site to validate, filter, or encode user input before returning it to another user's web client.
Cryptographic Issues	CWE-310	An insecure algorithm or the inappropriate use of one; an incorrect implementation of an algorithm that reduces security; the lack of encryption (plaintext); also, weak key or certificate management, key disclosure, random number generator problems.
Path Traversal	CWE-22	When user-supplied input can contain "... or similar characters that are passed through to file access APIs, causing access to files outside of an intended subdirectory.
Code Injection	CWE-94	Causing a system to read an attacker-controlled file and execute arbitrary code within that file. Includes PHP remote file inclusion, uploading of files with executable extensions, insertion of code into executable files, and others.
Format String Vulnerability	CWE-134	The use of attacker-controlled input as the format string parameter in certain functions.
Configuration	CWE-16	A general configuration problem that is not associated with passwords or permissions.
Information Leak / Disclosure	CWE-200	Exposure of system information, sensitive or private information, fingerprinting, etc.
Input Validation	CWE-20	Failure to ensure that input contains well-formed, valid data that conforms to the application's specifications. Note: this overlaps other categories like XSS, Numeric Errors, and SQL Injection.
Numeric Errors	CWE-189	Integer overflow, signedness, truncation, underflow, and other errors that can occur when handling numbers.
OS Command Injections	CWE-78	Allowing user-controlled input to be injected into command lines that are created to invoke other programs, using system() or similar functions.
Race Conditions	CWE-362	The state of a resource can change between the time the resource is checked to when it is accessed.
Resource Management Errors	CWE-359	The software allows attackers to consume excess resources, such as memory exhaustion from memory leaks, CPU consumption from infinite loops, disk space exhaustion, etc.
SQL Injection	CWE-89	When user input can be embedded into SQL statements without proper filtering or quoting, leading to modification of query logic or execution of SQL commands.
Link Following	CWE-59	Failure to protect against the use of symbolic or hard links that can point to files that are not intended to be accessed by the application.
Other	No Mapping	NVD is only using a subset of CWE for mapping instead of the entire CWE, and the weakness type is not covered by that subset.
Not in CWE	No Mapping	The weakness type is not covered in the version of CWE that was used for mapping.
Insufficient Information	No Mapping	There is insufficient information about the issue to classify it; details are unknown or unspecified.
Design Error	No Mapping	A vulnerability is characterized as a "Design error" if there exists no errors in the implementation or configuration of a system, but the initial design causes a vulnerability to exist.

Figure 4–2 CWE cross section

In the United States, the National Vulnerability Database, run by the National Institute of Standards and Technology (NIST) and sponsored by the National Cyber Security Division of the Department of Homeland Security (DHS), has a website to provide a clearinghouse for shared information called Common Weakness Enumeration (CWE).¹¹ See Figure 4–2.

The screenshot shows a web browser window with multiple tabs open. The active tab is titled 'CAPEC-196: Session Credential Falsification through Forging (Version 2.6)'. The page content includes:

- Attack Pattern ID:** 196
- Abstraction:** Standard
- Status:** Draft
- Completeness:** Complete
- Presentation Filter:** Basic
- Summary:** An attacker creates a false but functional session credential in order to gain or usurp access to a service. Session credentials allow users to identify themselves to a service after an initial authentication without needing to resend the authentication information (usually a username and password) with every message. If an attacker is able to forge valid session credentials they may be able to bypass authentication or piggy-back off some other authenticated user's session. This attack differs from Reuse of Session IDs and Session Sidejacking attacks in that in the latter attacks an attacker uses a previous or existing credential without modification while, in a forging attack, the attacker must create their own credential, although it may be based on previously observed credentials.
- Attack Prerequisites:**
 - The targeted application must use session credentials to identify legitimate users. Session identifiers that remains unchanged when the privilege levels change. Predictable session identifiers.
- Solutions and Mitigations:**
 - Implementation: Use session IDs that are difficult to guess or brute-force. One way for the attackers to obtain valid session IDs is by brute-forcing or guessing them. By choosing session identifiers that are sufficiently random, brute-forcing or guessing becomes very difficult.
 - Implementation: Regenerate and destroy session identifiers when there is a change in the level of privilege: This ensures that even though a potential victim may have followed a link with a fixed identifier, a new one is issued when the level of privilege changes.
- Related Attack Patterns:**

Nature	Type	ID	Name	Count
ChildOf		21	Exploitation of Session Variables, Resource IDs and other Trusted Credentials	1000
CanPrecede		384	Application API Message Manipulation via Man-in-the-Middle	1000
ParentOf		59	Session Credential Falsification through Prediction	1000
ParentOf		226	Session Credential Falsification through Manipulation	1000

Figure 4–3 Possible attack example from CAPEC site

A related website (seen in Figure 4–3) is called *Common Attack Pattern Enumeration and Classification* (CAPEC¹²). It is designed to not only name common problems but also to give developers and testers information to detect and fight against different attacks. From the site “about” page:

Building software with an adequate level of security assurance for its mission becomes more and more challenging every day as the size, complexity, and tempo of software creation increases and the number and the skill level of attackers continues to grow. These factors each exacerbate the issue that, to build secure software, builders must ensure that they have protected every relevant potential vulnerability; yet, to attack software, attackers often have to find and exploit only a single exposed vulnerability. To identify and mitigate relevant vulnerabilities in software, the development community needs more

11. <http://nvd.nist.gov/cwe.cfm>

12. capec.mitre.org

than just good software engineering and analytical practices, a solid grasp of software security features, and a powerful set of tools. All of these things are necessary but not sufficient. To be effective, the community needs to think outside of the box and to have a firm grasp of the attacker's perspective and the approaches used to exploit software. An appropriate defense can only be established once you know how it will be attacked.¹³

The objective of the site is to provide a publicly available catalog of attack patterns along with a comprehensive schema and classification taxonomy.

Finally, there is a useful-for-testing website for the Open Web Application Security Project (OWASP).¹⁴ This not-for-profit website is dedicated to improving the security of application software. This is a wiki type of website, so your mileage may vary when using it. However, there are a large number of resources that we have found on it that would be useful when trying to test security vulnerabilities, including code snippets, threat agents, attacks, and other information.

So, there are lots of security vulnerabilities and a few websites to help. What should you do as a technical test analyst to help your organization?

As you might expect, the top of the list has to be static testing. This should include multiple reviews, walk-throughs, and inspections at each phase of the development life cycle as well as static analysis of the code. These reviews should include adherence to standards and guidelines for all work products in the system. While this adds bureaucracy and overhead, it also allows the project team to carefully look for issues that will cause problems later on. Information on security vulnerabilities should be supplied by checklists and taxonomies to improve the chance of finding the coming problems before going live.

What should you look for? Certain areas will be most at risk. Communication protocols are an obvious target, as are encryption methods. The configurations in which the system is going to be used may be germane. Look for open ports that may be accessed by those who wish to breach the software.

Don't forget to look at processes that the organization using the system will employ. What are the responsibilities of the system administrator? What password protocols will be used? Many times the system is secure, but the environment is not. What hardware, firmware, communication links, and networks will be used? Where will the server be located? Since the tester likely will not have the ability to follow up on the software in the production environment, the next

13. <http://capec.mitre.org/about/index.html>

14. OWASP.org

best thing is to ensure that documentation be made available to users of the software, listing security guidelines and recommendations.

Static analysis tools, preferably ones that can match patterns of vulnerabilities, are useful, and dynamic analysis tools should also be used by both developers and testers at different levels of test.

There are hundreds, possibly thousands, of tools that are used in security testing. If truth be known, these are the same tools that the hackers are going to use on your system. We did a simple Google search and came up with more listings for security tools than we could read. The tools seem to change from day to day as more vulnerabilities are found. Most of these tools seem to be open source and require a certain competence to use.

Understanding what hackers are looking for and what your organization could be vulnerable to is important. Where is your critical data kept? Where can you be hurt worst?

If any of these steps are met with, “We don’t know how to do this,” then the engagement (or hiring) of a security specialist may be indicated. Frankly, in today’s world, it is not a question of if you might be hit; it is really a question of when and how hard. At the time we write this, our tips are already out of date. Security issues are constantly morphing, no matter how much effort your organization puts into them.

Helen Keller once said, “Security is mostly superstition. It does not exist in nature.” In software, security only comes from eternal vigilance and constant testing.

4.2.1.9 Internal Security Metrics

Access Auditability: A measure of how auditable the access login is. To calculate, count the number of access types that are being logged correctly as required by the specifications. Compare that number with the number of access types that are required to be logged in the specifications. Calculation is done by using the formula

$$X = A/B$$

where A is the number of access types that are being logged as called for in the specifications and B is the number of access types required to be logged in the specifications. The result will be $0 \leq X \leq 1$. The closer to 1, the more auditable this system would be. This metric is targeted at both the analysts writing the requirements and the developers reviewing them. Obviously, this metric (and others like it) are useful only if you are collecting measurements of this type.

Access Controllability: A measure of how controllable access to the system is. To calculate, count the number of access controllability requirements implemented correctly as called for in the specifications and compare with the number of access controllability requirements actually in the specifications. Calculation is done by using the formula

$$X = A/B$$

where A is the number of access controllability requirements implemented correctly as called for in the specifications and B is the total number of access controllability requirements in the specifications. The result will be $0 \leq X \leq 1$. The closer to one, the more controllable this system would be. This metric is targeted at both analysts writing the requirements and the developers reviewing them.

Data Corruption Prevention: A measure of how complete the implementation of data corruption prevention is. To calculate, count the number of implemented instances of data corruption prevention as specified and compare that with the number of instances of operations or access specified in the requirements as capable of corrupting or destroying data. Calculation is done by using the same formula:

$$X = A/B$$

A is the number of implemented instances of data corruption prevention as called for in the specifications that are actually confirmed in review, and B is the number of instances of operations/accessible identified in the requirements as capable of corrupting or destroying data. The result will be $0 \leq X \leq 1$. The closer to one, the more complete the prevention of data corruption this system would have. This metric is targeted at developers reviewing the specifications. If there are multiple security levels defined, this metric would likely be aimed at the highest levels of security.

Data Encryption: A measure of how complete the implementation of data encryption is. To calculate, count the number of implemented instances of encryptable/decryptable data items as specified and compare with the number of instances of data items requiring data encryption/decryption facilities as defined in the requirements. Calculation uses the same formula:

$$X = A/B$$

A is the number of implemented instances of encryptable/decryptable data items called for in the specifications that are confirmed in review. B is the total number of data items that require data encryption/decryption facility as defined

in the specifications. The result will be $0 \leq X \leq 1$. The closer to 1, the more complete this encryption would be. This metric is targeted at developers reviewing requirements, designs, and source code.

4.2.1.10 External Security Metrics

Access Auditability: A measure of how complete the audit trail concerning “user accesses to the system and to the data” is. To calculate, evaluate the amount of access that the system recorded in the access history database. Use the formula

$$X = A/B$$

where A is the number of user accesses to the system and data that are actually recorded in the access history database and B is the total number of user accesses to the system and data occurring during the evaluation time. For this metric to be meaningful, the tester would need to count the number of times attempts to access the system and data were made during the testing. This testing should be performed by penetration tests to simulate attacks. “User access to the system and data” may include “virus detection record” for virus protection. The result will be $0 \leq X \leq 1$. The closer to 1.0, the better the auditability should be.

Access Controllability: A measure of how controllable access to the system actually is. To calculate, count the number of different detected illegal operations compared to the number of different illegal operations possible as defined in the specifications. Again we use the same formula:

$$X = A/B$$

A is the number of actually detected different types of illegal operations, and B is the number of illegal operations defined in the specifications. Penetration tests should be performed to simulate an attack when performing this measurement. Attacks should include unauthorized persons trying to create, delete, or modify programs or information. The result will be $0 \leq X \leq 1$. The closer to 1.0, the better the access controllability will be.

Data Corruption Prevention: A measure of the frequency of data corruption events. This is done by measuring the occurrences of both major and minor data corruption events. This is calculated by using the formulae

$$X = 1 - A/N$$

$$Y = 1 - B/N$$

$$Z = A/T \text{ or } B/T$$

where A is the number of times a major data corruption event occurred and B is the number of times a minor data corruption event occurred. N is the number of test cases that were run trying to cause a data corruption event. T is the amount of time spent actually testing. This requires intensive abnormal operation testing to be done trying to cause corruption. Penetration¹⁵ tests should be run to simulate attacks on the system. For X and Y, the closer the measurement is to 1.0, the better. For Z, the closer the measurement to 0, the better (i.e., a longer period of operation is measured).

4.2.1.11 Exercise: Security

Using the HELLOCARMS System Requirements document, analyze the risks and create an informal test design for security.

4.2.1.12 Exercise: Security Debrief

We picked system requirement 010-040-040, which states, “Support the submission of applications via the Internet, providing security against unintentional and intentional security attacks.”

As soon as we open this system up to the Internet, security issues (and thus testing) come to the forefront.

During our analysis phase, we would try to ascertain exactly how much security testing had already been done on HELLOCARMS itself and the interoperating systems. Since up until now the systems had been reasonably closed, we would expect to find some untested holes. These holes would prompt an estimate for testing them, to make sure we have the resources we need.

Next, as part of our analysis, we would investigate the most common web security holes on sites mentioned earlier in this chapter: CVE (Common Vulnerabilities and Exposures), NVD (National Vulnerability Database), CAPEC (Common Attack Pattern Enumeration and Classification), and OWASP (Open Web Application Security Project). We would want all the help we could find.

We would ensure that TTAs were active in static testing at every level as the design and code were being developed.

15. Note that penetration tests may bring their own baggage. As suggested by commenter Bernard Homès, “Such tests (1) are difficult to implement, (2) need to be authorized by upper management as they can be interpreted as hacking attempts, (3) need reformed hackers to implement (white hat vs black hats)”

Our test suite would likely contain tests to cover the following:

- Injection flaws where untrusted data is sent to our site trying to trick us into executing unintended commands
- Cross-site scripting where the application takes untrusted data and sends it to the web server without proper validation
- Testing the authentication and session management functions to make sure unauthorized users are not allowed to log in
- Testing HELLOCARMS code to make sure direct objects (files, directories, database keys, etc.) were not available from the browsers
- Ensuring that no unencrypted or lightly encrypted data was sent to browsers (including making sure the keys are not sent with the data)
- Checking to make sure all certificates are tested correctly to avoid corrupted or invalid certificate acceptance
- Testing any links on our pages to ensure that we use only trusted data in our links (would not want a reputation for forwarding our customers to malware sites)

4.3 Reliability Testing

Learning objectives

TTA-4.3.1 (K3) Define the approach and design high-level test cases for the reliability quality characteristic and its corresponding ISO 9126 subcharacteristics.

While we believe that software reliability is always important, it is essential for mission-critical, safety-critical, and high-usage systems. As you might expect, reliability testing can be used to reduce the risk of reliability problems. Frequent bugs underlying reliability failures include memory leaks, disk fragmentation and exhaustion, intermittent infrastructure problems, and lower-than-feasible time-out values.

ISO 9126 defines reliability as “the ability of the software product to perform its required functions under stated conditions for a specified period of time, or for a specified number of operations.” While some of our information can come from evaluating metrics collected from other testing, we can also test for reliability by executing long suites of tests repeatedly. Realistically, this will require automation to get meaningful data.

ISTQB Glossary

maturity: (1) The capability of an organization with respect to the effectiveness and efficiency of its processes and work practices. See also Capability Maturity Model Integration, Test Maturity Model integration. (2) The capability of the software product to avoid failure as a result of defects in the software.

operational acceptance testing: Operational testing in the acceptance test phase, typically performed in a (simulated) operational environment by operations and/or system administration staff 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 be executed in noncontiguous time segments.

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.

robustness: The degree to which a component or system can function correctly in the presence of invalid inputs or stressful environmental conditions.

A precise, mathematical science has grown up around the topic of reliability. This involves the study of patterns of system use, sometimes called *operational profiles*. The operational profile will often include not only the set of tasks but also the users who are likely to be active, their behaviors, and the likelihood of certain tasks occurring. One popular definition describes an operational profile as being a “quantitative characterization of how a system may be used.”

Operational profiles can help allocate resources for architectural design, development, testing, performance analysis, release, and a host of other activities that may occur during the software development life cycle (SDLC). For reliability testing, operational profiles can help the organization understand just how reliable the system has to be in the performance of its mission, which tasks are in most need of reliability testing, and which functions need to be tested the most.

Overt reliability testing is really only meaningful at later stages of testing, including system, acceptance, and system integration testing. However, as you shall see, we can use metrics to calculate reliability earlier in the SDLC.

It is interesting to compare electronic hardware reliability to that of software. Hardware tends to wear out over time; in other words, there are usually physical faults that occur to cause hardware to fail. Software, on the other hand, never wears out. Limitations in software reliability over time almost always can be traced to defects originating in requirements, design, and implementation of the system.¹⁶

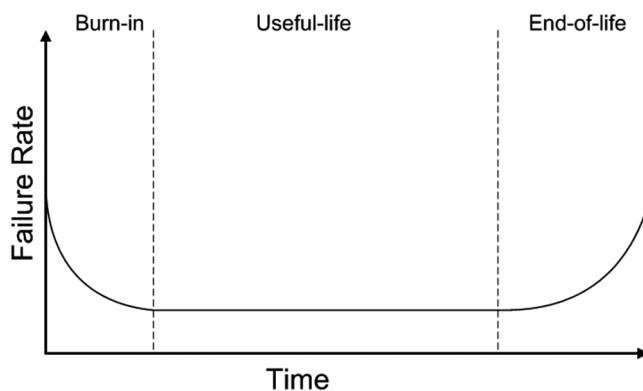


Figure 4–4 *Hardware reliability graph*

In Figure 4–4, you see a graph that shows—in general—the reliability of electronic hardware over time. Most failures will occur in the first few weeks of operation. If it survives early life, odds are very good that the electronic hardware will continue working for a long time—often past the point that it is targeted to be replaced by something faster/smarter/sexier.

16. There's a further complication that must be considered when moving beyond considering purely the electronic circuitry to looking at the physical device as a whole. When looking at physical objects, there are two different types of failures. One type is due to wear, which is a result of mechanical and chemical processes and can be exacerbated by misuse (e.g., failure to properly lubricate a bearing). Wear happens normally over time, except where accelerated by misuse. Another type of failure results from a defect in the components, such as the problems revealed recently with the bolts in the new San Francisco–Oakland Bay Bridge (<http://www.sfxaminer.com/sanfrancisco/bay-bridge-bolt-problem-arose-from-quality-control-lapses-officials-say/Content?oid=2336143>). Defects will result in premature (and often abrupt) failure, while wear is a gradual and usually detectable process.

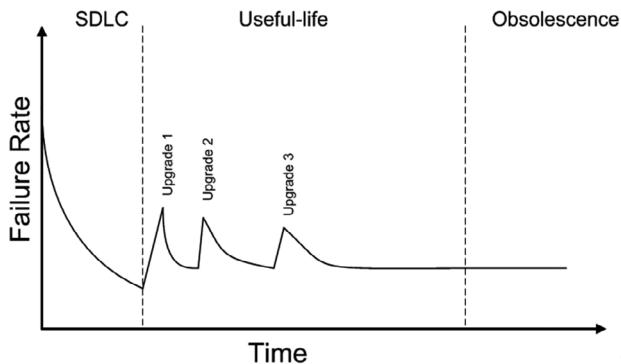


Figure 4–5 Software reliability graph

Compare that with the software reliability graph of Figure 4–5. Software has a relatively high failure rate during the testing and debugging period; during the SDLC, the failure rate will trend downwards (we hope!). While hardware tends to be stable over its useful life, however, software tends to start becoming obsolete fairly early in its life, requiring upgrades. The cynical might say that those upgrades are often not needed but pushed by a software industry that is not making money if it is not upgrading product. The less cynical might note that there are always features missing that require upgrades and enhancements. Each upgrade tends to bring a spike of failures (hence a lowering of reliability), which then tails off over time until the next upgrade.

Finally, as can be seen in Figure 4–5, as we get upgrades, complexity also tends to increase, which tends to lower reliability somewhat.

While there are techniques for accelerated reliability testing for hardware—so-called Highly Accelerated Life Tests (or HALT)—software reliability tests usually involve extended duration testing. The tests can be a small set of pre-specified tests, run repeatedly, which is fine if the workflows are very similar. (For example, an ATM could be tested this way.) The tests might be selected from a pool of different tests, selected randomly, which would work if the variation between tests were something that could be predicted or limited. (For example, an e-commerce system could be tested this way.) The test can be generated on the fly, using some statistical model (called stochastic testing). For example, telephone switches are tested this way, because the variability in called number, call duration, and the like is very large. The test data can also be randomly generated, sometimes according to a model.

In addition, standard tools and scripting techniques exist for reliability testing. This kind of testing is almost always automated; we've never seen an example of manual reliability testing that we would consider anything other than an exercise in self-deception.

Given a target level of reliability, we can use our reliability tests and metrics as exit criteria. In other words, we continue until we achieve a level of consistent reliability that is sufficient. In some cases, service-level agreements and contracts specify what "sufficient" means.

Many organizations use reliability growth models in an attempt to prognosticate how reliable a system may be at a future time. As failures are found during testing, the root causes of those failures should be determined and repaired by developers. Such repairs should cause the reliability to be improved. A mathematical model can be applied to such improvements, measuring the given reliability at certain steps in the development process and extrapolating just how much better reliability should be in the future.

A variety of different mathematical models have been suggested for these growth models, but the math behind them is beyond the scope of this book. Some of the models are simplistic and some are stunningly complex. One point we must make, however: whichever reliability growth model an organization chooses to use, it should eventually be tuned with empirical data; otherwise, the results are liable to be meaningless. We have seen a lot of resources expended on fanciful models that did not turn out to be rooted anywhere close to reality. What this means is that, after the system is delivered into production, the test team should still monitor the reliability of the system and use that data to determine the correctness of the reliability model chosen.

In the real world, reliability is a fickle thing. For example, you might expect that the longer a system is run without changes, the more reliable it will become. Balancing that is the observation that the longer a system runs, the more people might try to use it differently; this may result in using some capability that had never been tried before and, incidentally, causing the system to fail.

Metrics that have been used for measuring reliability are defect density (how many defects per thousand lines of source code [KLOC] or per function point). Cyclomatic complexity, essential complexity, number of modules, and counting certain constructs (such as GOTOs) have all been used to try to determine correlation of complexity, size, and programming techniques to the number and distribution of failures.

Object-oriented development has its own ways of measuring complexity, including number of classes, average number of methods per class, cohesion in classes and coupling between classes, depth of inheritance used, and many more.

We tend to try to develop reliability measures during test so that we can predict how the system will work in the real world. But balancing that is the fact that the real world tends to be much more complex than our testing environment. As you might guess, that means there are going to be a whole host of issues that are then going to affect reliability in production. In test, we need to allow for the artificiality of our testing when trying to predict the future. The more production-like our testing environment, the more likely our reliability growth model will be meaningful.

So what does it all mean? Exact real numbers for reliability might not be as meaningful as trends. Are we trending down in failures (meaning reliability is trending up)?

4.3.1 Maturity

One term that is often used when discussing reliability is *maturity*. This is one of three subcharacteristics that are defined by ISO 9126. *Maturity* is defined as the capability of the system to avoid failure as a result of faults in the software. We often use this term in relation to the software development life cycle; the more mature the system, the closer it is to being ready to move on to the next development phase.

In maturity testing, we monitor software maturity and compare it to desired, statistically valid goals. The goal can be the mean time between failures (MTBF), the mean time to repair (MTTR), or any other metric that counts the number of failures in terms of some interval or intensity. Of course, there are failures and then there are failures. Maturity testing requires that all parties involved come to agreement as to what failures count toward these metrics. During maturity testing, as we find bugs that are repaired, we'd expect the reliability to improve. As discussed earlier, many organizations use reliability growth models to monitor this growth.

4.3.1.1 Internal Maturity Metrics

Fault detection: A measure of how many defects¹⁷ were detected in the reviewed software product, compared to expectations. This metric is collected by counting the number of detected bugs found in review and comparing that number to the amount that were estimated to be found in this phase of static testing. The metric is calculated by the formula

$$X = A / B$$

where A is the actual number of bugs detected (from the review report) and B is the estimated number expected.¹⁸ If X varies significantly above or below 1.0, then something unexpected has happened that requires further investigation.

Fault removal: A measure of how many defects that were found during review are removed (corrected) during design and implementation phases. The metric can be calculated by the formula

$$X = A / B$$

where A is the number of bugs fixed prior to dynamic testing (either during requirements, design, or coding) and B is the number of bugs found during review. The result will be $0 \leq X \leq 1$. The closer the value is to one, the better. A fault removal value of exactly one would mean that every detected defect had been removed.

Test adequacy: A measure of how many of the required test cases are covered by the test plan. To calculate this metric, count the number of test cases planned (in the test plan) and compare that value to the number of test cases required to “obtain adequate test coverage” using the formula

$$X = A / B$$

where A is the number of test cases designed in the test plan and confirmed in review and B is the number of test cases required. Our research into exactly what determines the number of test cases required or exactly how you can determine that value came up with no specific model in the ISO 9126 standard. Without a specifying model for the number of tests required to achieve the needed level of coverage, we do not find this metric useful. After all, tests can come from the

17. Remember that ISTQB treats the terms *bug*, *defect*, and *fault* as synonyms. ISO 9126 appears to use the term *fault* to mean bug, defect, and sometimes failure. Wherever possible, we have tried to reword the standard to meet ISTQB definitions.

18. ISO 9126 uses the term *estimated faults* a number of times. The standard notes that these will tend to come from either past history of the system or a reference model.

requirements, risk analysis, use cases, tester experience, and literally dozens of different places. How do you quantify that? Some of Rex's clients are able to use substantial, statistically valid, parametric models to predict B, in which case this metric is useful.

4.3.1.2 External Maturity Metrics

Estimated latent fault density: A measure of how many defects remain in the system that may emerge as future failures. This metric depends on using a reliability growth estimation model as we discussed earlier in the chapter. The formula for this metric is as follows:

$$X = \{\text{abs}(A_1 - A_2)\} / B$$

A₁ is the total number of predicted latent defects in the system, A₂ is the total number of actually occurring failures, and B is the product size. To get the predicted number of latent defects, you count the number of defects that are detected through testing during a specified trial period and then predict the number of future failures using the reliability growth model. The actual count of failures found comes from incident reports. Interestingly enough, ISO 9126-2 does not define how size (B) is measured; it should not matter as long as the same measurement is used in the reliability growth model. Some organizations prefer KLOC (thousands of lines of source code), while others prefer to measure function points.

There is the very real possibility that the number of actual failures may exceed the number estimated. ISO 9126-2 recommends reestimating, possibly using a different reliability growth model if this occurs. The standard also makes the point that estimating a larger number of latent defects should not be done simply to make the system look better when fewer failures are actually found. We're sure that could never happen!

A conceptual issue with this metric is that it appears to assume that there is a one-to-one relationship between defects and failures. As Rex points out, one latent defect may cause 0 to N failures, invalidating that relationship and possibly making this metric moot. However, the reliability growth model may have this concept built in, allowing for such an eventuality. The standard suggests trying several reliability growth models and finding the one that is most suitable. In some ways, we feel that is like shopping for a lawyer based on them saying what you want to hear...

Failure density against test cases: A measure of how many failures were detected during a defined trial period. In order to calculate this metric, use the formula

$$X = A_1 / A_2$$

where A_1 is the number of detected failures during the period and A_2 is the number of executed test cases. A desirable value for this metric will depend on the stage of testing. The larger the number in earlier stages (unit, component, and integration testing), the better. The smaller the number in later stages of testing and in operation, the better. This is a metric that is really only meaningful when it is monitored throughout the life cycle and viewed as a trend rather than a snapshot in time. Although ISO 9126-2 does not mention it, the granularity of test cases might skew this metric if there are large differences between test cases at different test levels. A unit test case will typically test a single operation with an object or function, while a system test case can easily last for a couple of hours and cause thousands of such operations to occur within the code. If an organization intends to use this metric across multiple test levels, they must derive a way of normalizing the size of A_2 . It is likely most useful when only used in each test level singularly, without comparing across test levels. The standard does note that testing should include appropriate test cases: normal, exceptional, and abnormal tests.

Failure resolution: A measure of how many failure conditions are resolved without reoccurrence. This metric can be calculated by the formula

$$X = A_1 / A_2$$

where A_1 is the total number of failures that are resolved and never reoccur during the trial period and A_2 is the total number of failures that were detected. Clearly, every organization would prefer that this value be equal to one. In real life, however, some failures are not resolved correctly the first time; the more often failures reoccur, the closer to zero this metric will get. The standard recommends monitoring the trend for this metric rather than using it as a snapshot in time.

Fault density: A measure of how many failures were found during the defined trial period in comparison to the size of the system. The formula is as follows:

$$X = A / B$$

A is the number of detected failures and B is the system size (again, ISO 9126-2 does not define how size is measured). This is a metric where the trend is most

important. The later the stage of testing, the lower we would like this metric to be. Two caveats must be made when discussing fault density. Duplicate reports on the same defect will skew the results, as will erroneous reports (where there was not really a defect, but instead the failure was caused by the test environment, bad test case, or other external problem). This metric can be a good measure of the effectiveness of the test cases. A less charitable view might be that it is a measure of how bad the code was when first released into test.

Fault removal: A measurement of how many defects have been corrected. There are two components to this metric: one covering the actually found defects and one covering the estimated number of latent defects. The formulae are

$$X = A_1 / A_2$$

$$Y = A_1 / A_3$$

where A_1 is the number of corrected defects, A_2 is the total number of actually detected defects, and A_3 is the total number of estimated latent defects in the system. In reality, the first formula is measuring how many found defects are not being removed. If the value of X is one, every defect found was removed; the smaller the number, the more defects are being left in the system. As with some of the other metrics, this measurement is more meaningful when viewed as a trend rather than isolated in time.

If the organization does not estimate the number of latent defects (A_3), this metric clearly cannot be used because A_3 would equal zero, causing bad stuff to happen when the calculation is made.

The value of Y will be $0 \leq 1$. If the estimated value of Y is greater than one, the organization may want to investigate if the software is particularly buggy or if the estimate based on the reliability growth model was faulty. If Y is appreciably less than one, the organization may want to investigate if the testing was not adequate to detect all of the defects. Remember that duplicate incident reports will skew this metric. The closer Y is to 1.0, the fewer defects in the system should be remaining (assuming that the reliability model is suitable).

Mean time between failures (MTBF): A measure of how frequently the software fails in operation. To calculate this metric, count the number of failures that occurred during a defined period of usage and compute the average interval between the failures. Not all failures are created the same. For example, minor failures such as misspellings or minor rendering failures are not included in this metric. Only those failures that interrupt availability of the system are counted. This metric can be calculated two ways:

$$X = T1 / A$$

$$Y = T2 / A$$

T1 is the total operation time, and T2 is the sum of all the time intervals where the system was running. The second formula can be used when there is appreciable time spent during the interval when the system was not running. In either case, A is the total number of failures that were observed during the time the system was operating.

Clearly, the greater the value of X or Y, the better. This metric may require more research as to why the failures occurred. For example, there may be a specific function that is failing while other functionality may be working fine. Determination of the distribution of the types of failures may be valuable, especially if there are different ways to use the system.

Test coverage: A measure of how many test cases have been executed during testing. This metric requires us to estimate how much testing it would require to obtain adequate coverage based on...something. As we noted earlier, the standard is hazy on exactly how many tests it takes to get “adequate coverage” and exactly where those tests come from. The formula to calculate this metric is as follows:

$$X = A / B$$

A is the number of test cases actually executed and B is the estimated number of test cases, from the requirements, that are needed for adequate coverage. The closer this value comes to one, the better.

Test maturity: A measure of how well the system has been tested. This is used, according to ISO 9126-2, to predict the success rate the system will achieve in future testing. As before, the formula consists of

$$X = A / B$$

where A is the number of test cases passed and B is the estimated number of test cases needed for adequate coverage based on the requirements specification documentation. There is that term *adequate coverage* again. If your organization can come up with a reasonable value for this, some of these metrics would be very useful. In this particular case, the standard does make some recommendation as to the type of testing to be used in this metric. The standard recommends stress testing using live historical data, especially from peak periods. It further recommends using user operation scenarios, peak stress testing, and overloaded data input. The closer this metric is to one, that is, the more test cases that pass in comparison to all that should be run, the better.

4.3.2 Fault Tolerance

The second subcharacteristic of reliability is *fault tolerance*, defined as the capability of a system to maintain a specified level of performance in case of software faults. When fault tolerance is built into the software, it often consists of extra code to avoid and/or survive and handle exceptional conditions. The more critical it is for the system to exhibit fault tolerance, the more code and complexity are likely to be added.

Other terms that may be used when discussing fault tolerance are *error tolerance* and *robustness*. In the real world, it is not acceptable for a single, isolated failure to bring an entire system down. Undoubtedly, certain failures may degrade the performance of the system, but the ability to keep delivering services—even in degraded terms—is a required feature that should be tested.

Negative testing is often used to test fault tolerance. We partially or fully degrade the system operation via testing while measuring specific performance metrics. Fault tolerance tends to be tested at each phase of testing.

During unit test we should test error and exception handling with interface values that are erroneous, including out of range, poorly formed, or semantically incorrect. During integration test we should test incorrect inputs from user interface, files, or devices. During system test we might test incorrect inputs from OS, interoperating systems, devices, and/or user inputs. Valuable testing techniques include invalid boundary testing, exploratory testing, state transition testing (especially looking for invalid transitions), and attacks. In Chapter 6, we discuss fault injection tools, which will be useful for this type of testing.

In several different metrics, ISO 9126 uses the term *Fault Pattern*. This is defined as a “generalized description of an identifiable family of computations”¹⁹ that cause harm. These have formally defined characteristics and are fully definable in the code. The theory is that if we can define the common characteristics of computations in a certain area that cause harm, we can use those as a vocabulary that will help us find instances of the fault pattern in an automated way so they can be removed. A website we introduced earlier when discussing security, CWE, was created to categorize many of the diverse fault patterns that have been discovered. CWE stands for Common Weakness Enumeration and has the following charter:

19. *DoD Software Fault Patterns*, Dr. Nikolai Mansourov, <https://buildsecurityin.us-cert.gov/sites/default/files/Mansourov-SoftwareFaultPatterns.pdf>

International in scope and free for public use, CWE provides a unified, measurable set of software weaknesses that is enabling more effective discussion, description, selection, and use of software security tools and services that can find these weaknesses in source code and operational systems as well as better understanding and management of software weaknesses related to architecture and design.

Each fault pattern is fully described and defined with the following information, allowing developers to identify and remove them when finding them in their code.

- Full description
- When introduced
- Applicable platforms
- Common consequences
- Detection methods
- Examples in code
- Potential mitigations
- Relationships to other patterns

4.3.2.1 Internal Fault Tolerance Metrics

Failure avoidance: A measure of how many fault patterns were brought under control to avoid critical and serious failures. To calculate this metric, count the number of avoided fault patterns and compare it to the number of fault patterns to be considered, using the formula

$$X = A / B$$

where A is the number of fault patterns that were explicitly avoided in the design and code and B is the number to be considered. The standard does not define where the number of fault patterns to be considered is to come from. We believe their assumption is that each organization will identify the fault patterns that they are concerned about and have that list available to the developers and testers, perhaps using a site like CWE.

Incorrect operation avoidance: A measure of how many functions are implemented with specific designs or code added to prevent the system from incorrect operation. As with failure avoidance, we count the number of functions that have been implemented to avoid or prevent critical and serious failures from occurring and compare them to the number of incorrect operation patterns that have been defined by the organization. While the term *operation patterns* is not

defined formally, examples of incorrect operation patterns to be avoided include accepting incorrect data types as parameters, incorrect sequences of data input, and incorrect sequences of operations. Based on the examples given, operation patterns are very similar to fault patterns, defining specific ways that the system may be used incorrectly. Once again, calculation of the metric is done by using the formula

$$X = A / B$$

where A is the number of incorrect operations that are explicitly designed to be prevented and B is the number to be considered as listed by the organization. ISO 9126-3 does not define exactly what it considers critical or serious failures; our assumption is that each organization will need to compile their own list, perhaps by mining the defect database.

An example of avoidance might be having precondition checks before processing an API call where each argument is checked to make sure it contains the correct data type and permissible values. Clearly, the developers of such a system must make trade-offs between safety and speed of execution.

4.3.2.2 External Fault Tolerance Metrics

Breakdown avoidance: A measure of how often the system causes the breakdown of the total production environment. This is calculated using the formula

$$X = 1 - A / B$$

where A is the number of total breakdowns and B is the number of failures. The problem that an organization might have calculating this metric is defining exactly what a breakdown is. The term *breakdown* in ISO 9126 is defined as follows: “the execution of user tasks is suspended until either the system is restarted or that control is lost until the system is forced to be shut down.” If minor failures are included in the count, this metric may appear to be closer to one than is meaningful.

The closer this value is to one (i.e., the number of breakdowns is closer to 0), the better. For example, if there were 100 total failures (90 minor and 10 major) and one breakdown, then including minor failures would yield the following measurement:

$$1 - 1/100 \rightarrow .99$$

Not including them would yield a wholly different number:

$$1 - 1/10 \rightarrow .90$$

Ideally, a system will be engineered to be able to handle internal failures without causing the total breakdown of the system; this of course usually comes about by adding extra fault sensing and repair code to the system. ISO 9126-2 notes that when few failures do occur, it might make more sense to measure the time between them—MTBF—instead of this metric.

Failure avoidance: A measure of how many fault patterns were brought under control to avoid critical and serious failures. Two examples are given: out of range data and deadlocks. Remember that these metrics are for dynamic testing. In order to capture this particular metric, we would perform negative tests and then calculate how often the system would be able to survive the forced fault without having it tumble into a *critical* or *serious* failure. ISO 9126-2 gives us examples of impact of faults as follows:

- Critical: The entire system stops or serious database destruction occurs.
- Serious: Important functionality becomes inoperable with no possible workarounds.
- Average: Most functions are still available, but limited performance occurs with workarounds.
- Small: A few functions experience limited performance with limited operation.
- None: Impact does not reach end user.

The metric is calculated using the familiar formula

$$X = A / B$$

where A is the number of avoided critical and serious failure occurrences against test cases for a given fault pattern and B is the number of executed test cases for the fault pattern. A value closer to one, signifying that the user will suffer fewer critical and serious failures, is better.

Incorrect operation avoidance: A measure of how many system functions are implemented with the ability to avoid incorrect operations or damage to data. Incorrect operations that may occur are defined in this case to include the following:

- Incorrect data types as parameters
- Incorrect sequence of data input
- Incorrect sequence of operations

To calculate this measurement, we count the number of negative test cases that fail to cause critical or serious failures (i.e., the negative test cases pass)

compared to the number of test cases we executed for that purpose. We use the formula

$$X = A / B$$

where A is the number of test cases that pass (i.e., no critical or serious failures occur) and B is the total number run. The closer to one, the better this metric shows incorrect operation avoidance. Of course, this metric should be matched with a measure of coverage to be meaningful. For example, if we run 10 tests and all pass, our measurement would be 1. If we run 1,000 tests and 999 of them pass, the measurement would be 0.999. We would argue that the latter value would be much more meaningful.

4.3.3 Recoverability

The next subcharacteristic for reliability is called *recoverability*, defined as the capability to reestablish a specified level of performance and recover the data directly affected in case of a failure.

Clearly, recoverability must be built into the system before we can test it. Assuming that the system has such capabilities, typical testing would include running negative tests to cause a failure and then measure the time that the system takes to recover and the level of recovery that occurs. According to ISO 9126, the only meaningful measurement is the recovery that the system is capable of doing automatically. Manual intervention does not count.

Failover functionality often is built into a system when the consequences of a software failure are so unthinkable that they must never be allowed to occur. *High availability* is a term that is often used in these scenarios. There are three principles of high availability engineering that should be tested:

- No single point of failure (add redundancy)
- Reliable failover when a failure does occur
- Immediate detection of failures when they occur (required to trigger failover)

If you consider this list, high availability really requires all three subcharacteristics of reliability (ignoring compliance for the moment) to be present (and hence tested). Jamie spent some time consulting at an organization that handled ATM and credit card transactions. That organization was a perfect example of one that required high availability.

When someone inserts their card in an ATM, or swipes it at a gasoline station or grocery store, they are not willing to wait for hours for the charge to be

accepted. Most people are barely willing to wait seconds. When a fault occurs and the processing of the card is interrupted, it is not permissible for the transaction to get lost, forgotten, or duplicated.

To be fair, not all of the magic for recoverable systems comes from the software. While working for the organization just noted, Jamie learned that a great deal of the high availability came from special hardware (Tandem computers) and a special operating system. However, all associated software had to be built with the utmost reliability and tested thoroughly to ensure that every single transaction was tracked and fulfilled, completed and logged.

Having said that, much recoverability does come completely—or mostly—from software systems.

Redundant systems may be built to protect groups of processors, discs, or entire systems. When one item fails, the recoverability includes the automatic failover to still-working items and the automatic logging of the failure so that the system operator will know that the system is working in a degraded mode. Testing this kind of system includes triggering various failures to ascertain exactly how much damage the system can take and still remain in service.

In certain environments, it is not sufficient to have arrays of similar systems to ensure recoverability. If all of the items of an array are identical, a failure mode that takes out one portion of the array may also affect the other portions. Therefore, some high availability systems are designed with multiple dissimilar systems. Each system performs exactly the same task, but does it in a different way. A failure mode affecting one of the systems would be unlikely to affect the others. Testing such a system would still require failover testing to ensure that the entire system remains in service after the fault.

Backup and restore testing also fits the definition of recoverability testing. This would include testing the physical backup and restoration tasks and the processes to actually perform them as well as testing the data afterward to ensure that the recovered data was actually complete and correct.

Many organizations have found that they do, indeed, have backups of all their data. Unfortunately, no one ever tested that the data was valid. After a catastrophe, they found that resources they saved by not fully testing their data recovery paled in respect to the losses incurred when that data restore did not work correctly. Oops!

4.3.3.1 Internal Recoverability Metrics

Restorability: A measure of how capable the system is to restore itself to full operation after an abnormal event or request. This metric is calculated by counting the number of restoration requirements that are implemented by the design and/or code and comparing them to the number in the specification documents. For example, if the system has the ability to redo or undo an action, it would be counted as a restoration requirement. Also included are database and transaction checkpoints that allow rollback operations. The formula for the measurement again consists of a ratio

$$X = A / B$$

where A is the number of restoration requirements that are confirmed to be implemented via reviews and B is the number called for in the requirements or design documents. This value will evaluate to $0 \leq X \leq 1$. The closer the value is to one, the better.

Restoration effectiveness: A measure of how effective the restoration techniques will be. Remember that all of these are internal metrics that are expected to come from static testing. According to ISO 9126-3, we get this measurement by calculation and/or simulation. The intent is to find the number of restoration requirements (defined earlier) that are expected to meet their time target *when those requirements specify a specific time target*. Of course, if the requirements do not have a time target, this metric is moot. The measurement has the now-familiar formula of

$$X = A / B$$

where A is the number of implemented restoration requirements meeting the target restore time and B is the total number of requirements that have a specified target time. For example, suppose that the requirements specification not only requires the ability to roll back a transaction, but it also defines that it must do so within N milliseconds once it is triggered. If this capability is actually implemented, and we calculate/simulate that it would actually work correctly, the metric would equal 1/1. If not implemented, the metric would be 0/1.

4.3.3.2 External Recoverability Metrics

Availability: A measure of how available the system is for use during a specified period of time. This is calculated by testing the system in an environment as much like production as possible, performing all tests against all system functionality and uses the formulae

$$X = \{ T_o / (T_o + T_r) \}$$

$$Y = A1 / A2$$

where T_o is the total operation time and T_r is the amount of time the system takes to repair itself (such that the system is not available for use). We are only interested in time that the system was able to automatically repair itself; in other words, we don't measure the time when the system is unavailable while manual maintenance takes place. $A1$ is the total number of times that a user was able to use the software successfully when they tried to and $A2$ is the number of total number of times the user tried to use the software during the observation period.

In the above formulae, X is the total time available (the closer to one, the more available the system was) while Y is a measure of the amount of time a user was able to successfully use the system. The closer Y approaches 1, the better the availability.

Mean down time: A measure of the average amount of time the system remains unavailable when a failure occurs before the system eventually starts itself back up. As we said before, ISO 9126-2 is only interested in measuring this value when the restoration of functionality is automatic; this metric is not used when a human has to intervene with the system to restart it. To calculate this measurement, measure the time the system is unavailable after a failure during the specified testing period using the formula

$$X = T / N$$

where T is the total amount of time the system is not available and N is the number of observed times the system goes down. The closer X is to zero, the better this metric is (i.e., the shorter the unavailable time).

Mean recovery time: A measure of the average time it takes for the system to automatically recover after a failure occurs. This metric is measured during a specified test period and is calculated by the formula

$$X = \text{Sum}(T) / N$$

where T is the total amount of time to recover from all failures and N is the number of times that the system had to enter into recovery mode. Manual maintenance time should not be included. Smaller is clearly better for this measure. ISO 9126-2 notes that this metric may need to be refined to distinguish different types of recovery. For example, the recovery of a destroyed database is liable to take much longer than the recovery of a single transaction. Putting all different kinds of failures into the same measurement may cause it to be misleading.

Restartability: A measure of how often a system can recover and restart providing service to users within a target time period. As before, this metric is only concerned with automatic recovery (where no manual intervention occurs). The target time period likely comes from the requirements (e.g., a specified service-level agreement is in place). To calculate this metric, we count the number of times that the system must restart due to failure during the testing period and how many of those restarts occurred within the target time. We use the formula

$$X = A / B$$

where A is the number of restarts that were performed within the target time and B is the total number of restarts that occurred. The closer this measurement is to one the better. The standard points out that we may want to refine this metric because different types of failures are liable to have radically different lengths of recovery time.

Restorability: A measure of how capable the system is in restoring itself to full operation after an abnormal event or request. In this case, we are only interested in the restorations that are defined by the specifications. For example, the specifications may define that the system contain the ability to restore certain operations, including database checkpoints, transaction checkpoints, redo functions, undo functions, and so on. This metric is not concerned with other restorations that are not defined in the specifications. Calculation of this metric is done using the formula

$$X = A / B$$

where A is the number of restorations successfully made and B is the number of restoration cases tested based on the specifications. As before, values closer to one, showing better restorability, are preferable.

Restore effectiveness: A measure of how effective the restoration capability of the system is. This metric is calculated using the now familiar formula:

$$X = A / B$$

A is the number of times restoration was successfully completed within the target time and B is the total number of restoration cases performed when there is a required restoration time to meet. Values closer to one show better restoration effectiveness. Once again, target time is likely defined in the requirements or service-level agreement.

4.3.4 Compliance

4.3.4.1 Internal Compliance Metrics

Reliability compliance: A measure of the capability of the system to comply with such items as internal standards, conventions, or regulations of the user organization in relation to reliability. This metric is calculated by using the formula:

$$X = A / B$$

A is the number of regulations we are in compliance with as confirmed by our static testing and B is the number of total regulations, standards, and conventions that apply to the system.

4.3.4.2 External Compliance Metrics

Reliability compliance: A metric that measures how acquiescent the system is to applicable regulations, standards, and conventions. This is measured by using the formula

$$X = 1 - A / B$$

where A is the number of reliability compliance items that were specified but were not implemented during the testing and B is the total number of reliability compliance items specified. ISO 9126-2 lists the following places where compliance specifications may exist:

- Product description
- User manual
- Specification of compliance
- Related standards, conventions, and regulations

The closer to one, the better; 1 would represent total compliance with all items.

4.3.5 An Example of Good Reliability Testing

The United States National Aeronautics and Space Administration (NASA) is responsible for trying to keep mankind in space. When its software fails, it might mean the end of a multibillion-dollar mission (not to mention many lives).

The NASA Software Assurance Standard, NASA-STD-8739.8, defines software reliability as a discipline of software assurance that does the following:

1. Defines the requirements for software controlled system fault/failure detection, isolation, and recovery
2. Reviews the software development processes and products for software error prevention and/or reduced functionality states
3. Defines the process for measuring and analyzing defects and defines/derives the reliability and maintainability factors

NASA uses both trending and predictive techniques when looking at reliability.

Trending techniques track the metrics of failures and defects found over time. The intent is to develop a reliability operational profile of a given system over a specified time period. NASA uses four separate techniques for trending:

1. Error seeding:²⁰ Estimates the number of errors in a program by using multistage sampling. Defects are introduced to the system intentionally. The number of unknown errors is estimated from the ratio of induced errors to noninduced errors from debugging data.
2. The failure rate: Study the failure rate per fault at the failure intervals. The theory goes that as the remaining number of faults change, the failure rate of the program changes accordingly.
3. Curve fitting: NASA uses statistical regression analysis to study the relationship between software complexity and the number of faults in a program as well as the number of changes and the failure rate.
4. Reliability growth: Measures and predicts the improvement of reliability programs throughout the testing process. Reliability growth also represents the failure rate of the system as a function of time and the number of test cases run.

NASA also uses predictive reliability techniques: These assign probabilities to the operational profile of a software system. For example, the system has a 5 percent chance of failure over the next 60 operational hours. This clearly involves a

20. This is also known as fault seeding, which we cover in Chapter 6.

capability for statistical analysis that is far beyond the capabilities of most organizations (due to lack of resources and skills).

Metrics that NASA collects and evaluates can be split into two categories: static and dynamic.

The following measures are static measures:

1. Line count, including lines of code (LOC) and source lines of code (SLOC)
2. Complexity and structure, including cyclomatic complexity, number of modules and number of GOTO statements
3. Object-oriented metrics, including number of classes, weighted methods per class, coupling between objects, response for a class, number of child classes, and depth of inheritance tree

Dynamic measures include failure rate data and the number of problem reports.

The question that must be asked when discussing reliability testing is, With which faults are we going to be concerned? A complex system can fail at hundreds or thousands of places. There is a great deal of cost involved in reliability testing. As the reliability needs of the organization go up, the costs escalate even faster. Therefore, an organization must plan the testing carefully. While NASA can afford to pull out all stops when it comes to reliability testing, most other organizations are not so lucky.

The following events may be of concern:

- An external event that should occur does not, a device that should be on line is not (or has degraded performance), or an interface or process that the system needs is not available
- The network is slow or not available, or it suddenly crashes
- Any of the operating system capabilities that the system relies on are not available or are degraded
- Inappropriate, unexpected, or incorrect user input occurs

A test team must determine which of these (or other possibilities) are important for the system's mission. The team would create [usually negative] tests to degrade or remove those capabilities and then measure the response of the system. Measurements from these tests are then used to determine if the system reliability was acceptable.

4.3.6 Exercise: Reliability Testing

Using the HELLOCARMS System Requirements document, analyze the risks and create an informal test design for reliability.

4.3.7 Exercise: Reliability Testing Debrief

We selected two related requirements, 020-010-020 and 020-010-030. The first, set in release two, requires that fewer than five (5) failures in production occur per month. The second requires that the number of failures in production be less than one (1) per month by release four. In essence, we are going to test mean time between failures (MTBF). Frankly, we might challenge this kind of a firm requirement in review because it sets a (seemingly) arbitrary value that may be impossible to meet within project constraints.

However, since the requirement is firm, it strikes us as an opportunity to create a long-running automated test that could be run over long periods (overnight, weekends, or perhaps on a dedicated workstation running for weeks).

This would depend on having automated tests available that exercise the GUI screens of the Telephone Banker. In order to be useful, the tests would need to be data-driven or keyword-driven tests, tests that can be run randomly with a very large data set.

Based on the workflow of the Telephone Banker, we would create a variety of scenarios, using random customer data. These scenarios would be as follows:

- Accepted and rejected loans of all sorts and amounts
- Accepted but declined-by-customer loans
- Hang-ups and disconnects
- Insurance accepts and declines

The defect theory that we would be testing is that the system may have reliability issues, especially when unusual scenarios are run in random order. Each test would clearly need to check for expected versus actual results. We would be looking for the number of failures that occur within the testing time period so we can get a read on the overall reliability of the system over time.

Each test build's metrics would be compared to the previous builds' to determine if the maturity of the system is growing (fewer failures per time period would indicate growing maturity).

Fault tolerance metrics would be extrapolated by determining how often the entire system fails when a single transaction fails as compared to being able to continue running further transactions despite the failure.

In those cases where the entire system does fail, recoverability would be measured by the amount of time it takes to get the entire HELLOCARMS system up and running again.

4.4 Efficiency Testing

Learning objectives

TTA-4.5.1 (K3) Define the approach and design high-level operational profiles for performance testing.

ISO 9126 defines efficiency as the capability of the software product to provide appropriate performance relative to the amount of resources used, under stated conditions. When speaking of resources, we could mean anything on the system, software, hardware, or any other abstract entities. For example, network bandwidth would be included in this definition.

Efficiency of a distributed system is almost always important. When might it not be important? Several years ago, Jamie was teaching a class in Juneau, Alaska. After class, he struck up a conversation—in a bar—with a tester who said he worked at the Alaska Department of Transportation. Discussion turned to a new distributed system that was going to be going live, allowing people from all over the state to renew their driver's licenses online. When Jamie asked about performance testing of the new system, the tester laughed. He claimed that a good day might have 10 to 12 users on the site. While the tester was likely exaggerating, the point we should draw from it is that efficiency testing, like all other testing, must be based on risk. Not every type of testing must always be done to every software system.

Efficiency failures can include slow response times, inadequate throughput, reliability failures under conditions of load, and excessive resource requirements. Efficiency defects are often design flaws at their core, which make them very hard to fix during late-stage testing. So, efficiency testing can and should be done at every test level, particularly during design and coding (via reviews and static analysis).

There are a lot of myths surrounding performance testing. Let's discuss a few.

Some testers think that the way to performance test is to throw hundreds (thousands?) of virtual users against the system and keep ramping them up until the system finally breaks down. The truth is that most performance testing is done while measuring the working system without causing it to fail. There is a

ISTQB Glossary

efficiency: (1) The capability of the software product to provide appropriate performance, relative to the amount of resources used under stated conditions. [ISO 9126]
(2) The capability of a process to produce the intended outcome, relative to the amount of resources used.

performance testing: The process of testing to determine the performance of a software product.

resource utilization testing: The process of testing to determine the resource utilization of a software product.

kind of performance testing that does try to find the breaking point of the system, but it is a small part of the entire range of ways we test.

A second myth states that we can only do performance testing at the end of system test. This is dangerously wrong and we will address it extensively. As noted, performance testing, like all other testing, should be pervasive throughout the life cycle.

Last, we often hear the myth that a good performance tester only needs to know about a performance tool. Learn the tool and you can walk into any organization and start making big money running tests tomorrow. Turns out this is also dangerously false. We will discuss all of the tasks that must be done for good performance testing before we ever turn on a tool.

4.4.1 Multiple Flavors of Efficiency Testing

There is an urban myth that the native Inuit peoples have more than 50 different names for snow, based on nuances in snow that they can see. While researching this story, we found there is wide dispute as to whether this is myth or provable fact.²¹ If factual, the theory is that they have so many names because to the Inuit, who live in the snow through much of the year, the fine distinctions are important while to others the differences are negligible. It depends on your viewpoint. Consider that in America we have machines that have very little difference between them; these go by the names Chevy, Buick, Cadillac, Ford, and so on. Show them to an Inuit and they might fail to see any big distinction between them.

21. If you are interested in more information on this topic, we suggest, http://www.washingtonpost.com/national/health-science/there-really-are-50-eskimo-words-for-snow/2013/01/14/e0e3f4e0-59a0-11e2-beee-6e38f5215402_story.html.

One of the most talented performance testers that Jamie ever met once showed Jamie a paper he was writing that enumerated some 40 different flavors of performance testing. Frankly, as Jamie read it, he did not understand many of the subtle differentiations the author was making. However, listening to others review the paper was an education in itself as they discussed subtle differences that Jamie had never considered.

The one thing we know for sure is that efficiency testing covers a lot of different test types. Shortly we will provide a sampling of the kinds of testing that might be performed. We have used definitions from the ISTQB glossary and ISTQB Advanced syllabus when available. Other definitions come from a performance testing class that was written by Rex. A few of the definitions come from a book by Graham Bath and Judy McKay.²² In each case, we tried to pick definitions that a wide array of testers have agreed on.

Most of these disparate test types go by the generic name *performance testing*. From the ISTQB glossary comes the following definition for performance testing itself: “the process of testing to determine the performance of a software product.”

A better definition can be constructed by blending the glossary definition of performance with that of performance testing as follows: Testing to evaluate the degree to which a system or component accomplishes its designated functions, within given constraints, regarding processing time and throughput rate.

A classic performance or response-time test looks at the ability of a component or system to respond to user or system inputs within a specified period of time, under various legal conditions. It can also look at the problem slightly differently, by counting the number of functions, records, or transactions completed in a given period; this is often called throughput. The metrics vary according to the objectives of the test.

So, with that in mind, here are some specific types of efficiency testing:

- Load testing: A type of performance testing conducted to evaluate the behavior of a component or system with increasing load (e.g., numbers of parallel users and/or numbers of transactions) to determine what load can be handled by the component or system. Typically, load testing involves various mixes and levels of load, usually focused on anticipated and realistic loads. The loads often are designed to look like the transaction requests generated by certain numbers of parallel users. We can then measure response time or

22. *The Software Test Engineer's Handbook*

throughput. Some people distinguish between multiuser load testing (with realistic numbers of users) and volume load testing (with large numbers of users), but we've not encountered that too often.

- Stress testing: A type of performance testing conducted to evaluate a system or component at or beyond the limits of its anticipated or specified workloads or with reduced availability of resources such as access to memory or servers. Stress testing takes load testing to the extreme and beyond by reaching and then exceeding maximum capacity and volume. The goal here is to ensure that response times, reliability, and functionality degrade slowly and predictably, culminating in some sort of "go away I'm busy" message rather than an application or OS crash, lockup, data corruption, or other antisocial failure mode.
- Scalability testing: Takes stress testing even further by finding the bottlenecks and then testing the ability of the system to be enhanced to resolve the problem. In other words, if the plan for handling growth in terms of customers is to add more CPUs to servers, then a scalability test verifies that this will suffice. Having identified the bottlenecks, scalability testing can also help establish load monitoring thresholds for production.
- Resource utilization testing: Evaluates the usage of various resources (CPU, memory, disk, etc.) while the system is running at a given load.
- Endurance or soak testing: Running a system at high levels of load for prolonged periods of time. A soak test would normally execute several times more transactions in an entire day (or night) than would be expected in a busy day to identify any performance problems that appear after a large number of transactions have been executed. It is possible that a system may stop working after a certain number of transactions have been processed due to memory leaks or other defects. Soak tests provide an opportunity to identify such defects, whereas load tests and stress tests may not find such problems due to their relatively short duration.
- Spike testing: The object of a spike test is to verify system stability during a burst of concurrent user and/or system activity to varying degrees of load over varying time periods. This type of test might look to verify a system against the following business situations:
 - A fire alarm goes off in a major business center and all employees evacuate. The fire alarm drill completes and all employees return to work and log into an IT system within a 20-minute period.
 - A new system is released into production and multiple users access the system within a very small time period.

- A system or service outage causes all users to lose access to a system. After the outage has been rectified, all users then log back onto the system at the same time.
 - Spike testing should also verify that an application recovers between periods of spike activity.
- Reliability testing: Testing the ability of the system to perform required functions under stated conditions for a specified period of time or number of operations.
 - Background testing: Executing tests with active background load, often to test functionality or usability under realistic conditions.
 - Tip-over testing: Designed to find the point where total saturation or failure occurs. The resource that was exhausted at that point is the weak link. Design changes (ideally) or more hardware (if necessary) can often improve handling and sometimes response time in these extreme conditions.

There are lots more, but our brains hurt. Unless we have a specific test in mind, we are just going to call all efficiency type testing by the umbrella name of performance testing for this chapter.

Not all of the tests just listed are completely disjoint; we could actually run some of them concurrently by changing the way we ramp up the load and which metrics we monitor.

To be meaningful, no matter which of these we run, there is much more to creating a performance test than buying a really expensive tool with 1,000 virtual user licenses and start cranking up the volume. We will discuss how to model a performance test correctly in the next section.

We have been in a number of organizations that seemed to believe that performance testing could not even be started until late in system testing. The theory goes that performance testing has to wait until the system is pretty well complete, with all the functionality in and mostly working.

Of course, if you wait until then to do the testing and you find a whole bundle of bugs when you do test (usually the case), then your organization will have the choice of two really bad options: delay the delivery of the system into production while fixes are made (that could happen, but don't hold your breath), or go ahead and deliver a crippled system while desperately scrambling to fix the worst of the defects. The latter is what we have mostly seen occur. Consider the melodrama of the 2013 rollout of healthcare.gov in the United States, perhaps the poster child for why not to skip the performance testing.

Good performance testing, like most good testing, should be distributed throughout all of the phases of the SDLC:

- During the development phases: From requirements through implementation, static testing should be done to ensure meaningful requirements and designs from an efficiency viewpoint.
- During unit testing: Performance testing of individual units (e.g., functions or classes) should be done. All message and exception handling should be scrutinized; each message type could be a bottleneck. Any synchronization code, use of locks, semaphores, and threading must be tested thoroughly, both statically and dynamically.
- During integration testing: Performance testing of collections of units (builds, backbones, and/or subsystems) should be performed. Any code that transfers data between modules should be tested. All interfaces should be scrutinized for deadlock problems.
- During system testing: Performance testing of the whole system should be done as early as possible. The delivery of functionality into test should be mapped so that those pieces that are delivered can be scheduled for the performance testing that can be done.
- During acceptance testing: Demonstration of performance of the whole system in production should be performed.

Realism of the test environment generally increases with each level, until system test, which should (ideally) test in a replica of the production or customer environment.

4.4.2 Modeling the System

In the early days of performance testing, many an organization would buy or lease a tool, pick a single process, record a transaction, and immediately start testing. They would create multiple virtual users using the same profile, using the same data. To call such testing meaningless is to tread too lightly.

If a performance test is to be meaningful, there are a lot of questions that must be answered that are more important than, How many users can we get on the system at one time? Come to think of it, that question (by itself) is about as meaningful as the old days when the raging question was how many teens can you get in a phone booth?

Here then are some important questions that should be asked before we get into the physical performance testing process.

What is the proposed scope of the effort; exactly which subsystems are we going to be testing? Which interfaces are important? Which components are we going to be testing? Are we doing the full end-to-end customer experience or is there a particular target we are after? Which configurations will be tested? These are just some of the questions we need to think about.

How realistic is the test going to be? If production has several hundred massive servers and we are going to be testing against a pair of small, slow, ancient servers, any extrapolation we would attempt to do would still come up with a meaningless answer since we have no way of knowing the network bandwidth, load balancers, bottlenecks, and issues in the real production architecture.

How many concurrent users do we expect? Average? Peak? Spike? What tasks are they going to be doing? Odds are really good that all of the users will not be touching the same record with the same user ID.

What is the application workload mix that we want to simulate? In other words, how many different types of users will we be simulating on the system and what percentages do they make up (e.g., 20 percent login, 40 percent search, 15 percent checkout, etc.)?

And while we are at it, how many different application workload mixes do we want on the system while we are testing? Many systems support several different concurrent applications running on the same servers. Testing only one may not be meaningful.

In that same vein, is virtualization going to be used? Will we be sharing a server with other virtualized processes? Will our processes be spread over multiple servers? Our research and discussions with a number of performance testers shows that there are a lot of different opinions as to how virtualization can affect performance testing.

Which back-end processes are going to be running during the testing? Any batch processes? Any dating processes? Month end processing? Those processes are going to happen in real life; do we need to model them for this test?

Be of good cheer—there are dozens more questions, but performance testing is possible to do successfully.

To give an example of a coherent methodology that an organization might use for doing performance testing of a web application, we have pulled one from the Microsoft Developer Network.²³ This methodology consists of seven steps as detailed in the following sections.

23. *Performance Testing Guidance for Web Applications*, <http://msdn.microsoft.com/en-us/library/bb924375.aspx>

4.4.2.1 Identify the Test Environment

Identify the test environment—and the production environment, including hardware, software, and network configurations.

Assess the expected test environment and evaluate how it compares to the expected production environment. Clearly, the closer our test system is to the expected production system, the more meaningful our test results can be. Remember, the production environment will undoubtedly already have a certain amount of load on it.

Balanced against that is the cost. Replicating the environment exactly as it is found in production is usually not going to happen. Somewhere we need to strike a balance.

Understand the tools and resources available to the test team. Having the latest and greatest of every tool along with an unlimited budget for virtual users would be a dream. If you are working in the kind of organizations that we have, dreaming of that is as close as you will get.

Identify challenges that must be met during the process. Realistically, consider what is likely to happen. Many software people are unquenchable optimists; we just know that everything is going to go right *this time*. While we can hope, we need to plan as realists—or, as the Foundation syllabus says, be professional pessimists.

This first step is likely to be one that is revisited throughout the process as compromises are made and changes are made. Like risk analysis, which we discussed in Chapter 1, we need to always be reevaluating the future based on what we discover during the process.

4.4.2.2 Identify the Performance Acceptance Criteria

Identify the goals and constraints for the system. Remember that many in the project may not have thought these issues through. Testers can help focus the project on what we really can achieve. There are three main ways of looking at what we are interested in:

- Response time: user's main concern
- Throughput: often the business' concern
- Resource utilization: system concerns

Identify system configurations that might result in the most desirable combinations of the items in the preceding list. This might take some doing since people in the project may not have considered these issues yet.

Identify project success criteria—how will you know when you are done? While it is tempting for testers to want to determine what success looks like, it is up to the project as is, to make that determination. Our job is to capture information that allows the other project members to make informed decisions as to pass/fail. So which metrics are we going to collect? Don't try to capture every different metric that is possible. Settle on a given set of measurements that are meaningful to proving success—or disproving it.

4.4.2.3 Plan and Design Tests

Model the system as mentioned earlier to identify key scenarios and likely usage.

Determine how to simulate the inevitable variability of scenarios—what do different users do and how do they do it? What is the business context in which the system is used? Focus on groups of users; look for common ways they interact with the system.

Define test data—and enough of it! Remember that different user groups will likely have distinctive differences in the data they use. Log files from production can be very helpful in gathering data information. Don't forget to review the data you will be using with the actual users themselves when possible; they can help you find what you might have overlooked.

Make sure you consider timing as part of the data collection. Different groups will work at different rates. Not accounting for actual work patterns will very likely skew results. Don't forget user abandonment; not every task is completed by all users. Consolidate all of the above suggestions into different models of system usage to be tested

4.4.2.4 Configure the Test Environment

Prepare the test environment, tools, and resources needed to execute the models designed in the preceding step (plan and design tests). Don't forget to generate background load as closely as possible to production. Validate that your environment matches production to the extent that it can and document where it doesn't. Differences between test and production environments must be taken into account or the test results will not model reality.

Determine the schedule for when the necessary features will become available (i.e., match up with the SDLC). Not all functionality will likely be available on day one of testing.

And, finally, instrument the test environment to enable collection of the desired metrics.

4.4.2.5 Implement the Test Design

Implement the test design. Create the performance testing scripts using the tools available.

Ensure that the data parameterization is as needed. This is a good place to double-check that you have sufficient data to run your tests. If you are performing soak testing, you will need a lot of data.

Smoke test the design and modify scripts as needed. One phrase Jamie remembers vividly from his five years of Latin language classes: *Quis custodiet ipsos custodes?*²⁴ Who will guard the guardians themselves? Nonvalidated tests could easily be giving us bogus information. Always ensure—*before beginning the actual testing*—that the test scripts are meaningful and performing the actual tasks you are expecting them to. Make sure to ask yourself, “Do these results make sense?” Do not report the results of the smoke test as part of the official test results.

4.4.2.6 Execute the Test

Run the tests, monitoring the results. Work with database, network, and system personnel to facilitate the testing (first runs often show serious issues that must be addressed). Ideally, any issues would have been addressed during the validation of the scripts, but expecting the unexpected is pretty much par for the course in performance as with all testing.

Validate the tests as being able to run successfully, end to end. Run the testing in one- to two-day batches to constantly ask the reasonableness question: Are the results we are getting sensible? Beware of a common mistake made among scientists, however. When the results are not what were expected, sometimes scientists believe that the tests are invalid rather than there might be something wrong with their hypotheses. It might just be that your expectations were wrong.²⁵

Execute the validated tests for the specified time and under the defined conditions to collect the metrics. It often makes sense to repeat tests to ensure that the results are similar. If they are not similar, why not? Often there are hidden

24. *Satires of Juvenal*. Probably not talking about software, but still worth considering...

25. Arno A. Penzias and Robert W. Wilson, working for Bell Labs in the early '60s, accidentally discovered the first observational proof of the “Big Bang” when nothing they could do would eliminate the static they kept picking up with their microwave receiver. However, they spent months not believing what their tests were telling them. See http://www.amnh.org/education/resources/rfl/web/essaybooks/cosmic/cs_radiation.html.

factors that might be missed if tests are only run once. When you stop getting valuable information, you have run the tests enough.

4.4.2.7 Analyze the Results, Tune and Retest

Analyze the completed metrics. Do they prove what you wanted to prove? If they do not match expected results, why not?

Consolidate and share results data with stakeholders. As with all other testing, remember that you must report the results to stakeholders in a meaningful way. The fact that you had 1,534 users active at the same time is really cool; however, the stakeholders are more interested in whether the system will support their needed business goals.

Tune the system. Can we make changes to the system to positively affect the performance of it? This becomes more important the closer to production your test environment is. Small changes can often create huge differences in the performance of the system. Our experience is that those changes are often negative. This task will, of course, depend on time and resources.

How do you know when you are done with efficiency testing? According to the MS test guide:

When all of the metric values are within accepted limits, none of the set thresholds have been violated, and all of the desired information has been collected, you have finished testing that particular scenario on that particular configuration.

You may or may not have the time and resources to completely finish performance testing your system to the extent mentioned here. If not, don't let it stop you from performing those parts you can afford. Don't let the perfect be the enemy of the good!

4.4.3 Time Behavior

ISO 9126 identifies three subcharacteristics for efficiency: *performance (time behavior), resource utilization, and compliance*.

The good thing for technical test analysts is that many of our measurements for efficiency testing are completely quantifiable. We can measure them directly. Well, kind of. The truth is, every time we make a measurement, we can get an exact value. It took 3 milliseconds for this thing to occur. We had 100 virtual users running concurrently, doing this, this, and this.

In performance testing, we often have to run the same test over and over again so that we can average out the results. Because the system is extremely

complex, and other things may be going on, and timing of all the things going on may not be completely deterministic, and CPU loading may be affected by internal processes and a hundred other things...well, you get the picture. So, we run the tests multiple times, measure the things we want to an exact amount, and average them over the multiple runs.

When Jamie was in the military, they used to joke about how suppliers met government specifications. Measure something with a micrometer and then cut it with a chainsaw. Sometimes that is how we feel about the metrics we get from performance testing.

Time-critical systems, which include most safety-critical, real-time, and mission-critical systems, must be able to provide their functions in a given amount of time. Even less critical systems like e-commerce and point-of-sale systems should have good time response to keep users happy.

Time behavior uses measurements that look at the amount of time it takes to do something. The following measurements are possible:

- Number of context switches per second
- Network packet average round-trip time
- Client data presentation time
- Amount of time a transaction takes between any pair of components

Let's discuss four separate scenarios where we may discover time behavior anomalies when performance testing:

- Slow response under all load levels
- Slow response under moderate loading of the system where the amount of loading is expected and allowed
- Response that degrades over time
- Inadequate error handling when loaded

First, let's discuss the underlying graph that we will use to illustrate these bugs. In Figure 4–6, the vertical scale represents the amount of time transactions are taking to process on average. In general, the less time a transaction takes to execute, the happier the user will be. The horizontal scale shows the arrival time rate; in other words, how many transactions the system is trying to execute over a specified time.

Normally, as more and more transactions arrive, we would expect that the system may get a little slower (shown by the line getting a little higher on the graph the farther to the right it travels). Ideally, the line will stay in the gray, lower area, which is labeled the acceptable performance area. As long as the line

stays in the gray, we are within the expected performance range and we would expect our users to be satisfied with the service they are getting.

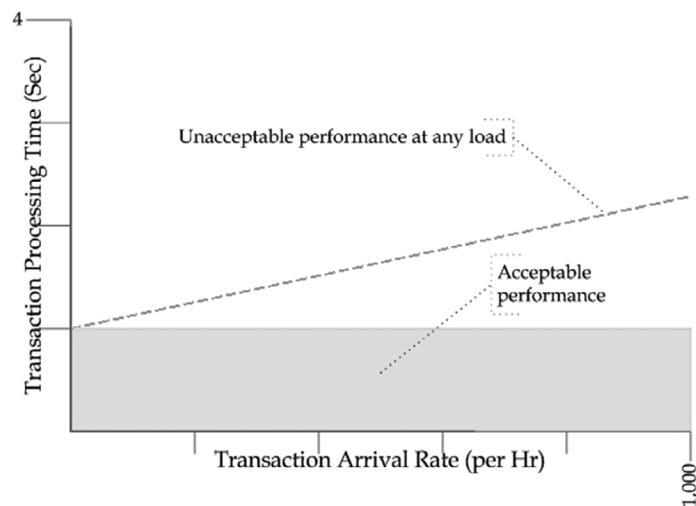


Figure 4-6 Unacceptable performance at any load example

In Figure 4–6, you can see that we definitely have a problem. Even with no loading at all, the performance is just barely acceptable; as load just begins to ramp up, we move immediately out of the acceptable range. This is something we would expect [hope] to find during functional testing, long before we start performance testing. However, we often miss it because functional testing tends to test the system with a single user.

Some of the issues that might be causing this are a bad database design or implementation where trying to access data just takes too long. Network latency may be problematic, or the server might be too loaded with other processes. This is a case where monitoring a variety of different metrics should quickly point out the problem.

In Figure 4–7, we start out well within the acceptable range. However, there is a definite knee before we get to 400 transactions per hour. Where the response had degraded slightly in a linear fashion, all of a sudden the degradation got much faster, rapidly moving out of the acceptable range at about 500 transactions per hour.

This is representative of a resource reaching its capacity limit and saturating. Looking at the key performance indicator metrics at this point will generally show this; we may have high CPU utilization, insufficient memory, or some

other similar problem. Again, the problem could also be that there are background processes that are chewing up the resources.

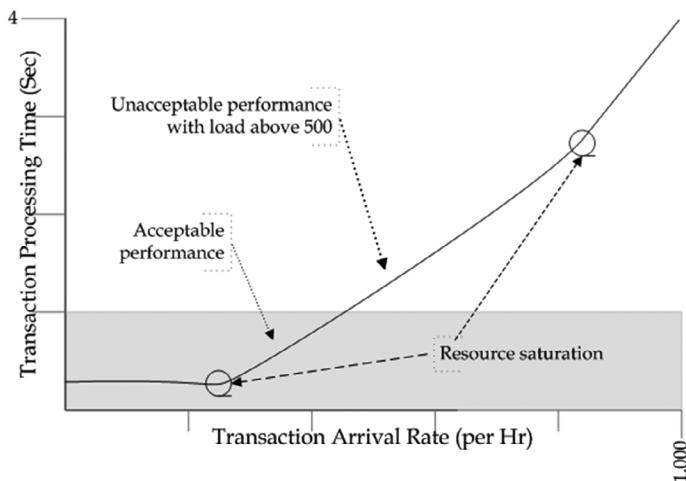


Figure 4–7 Slow response under moderate loading example

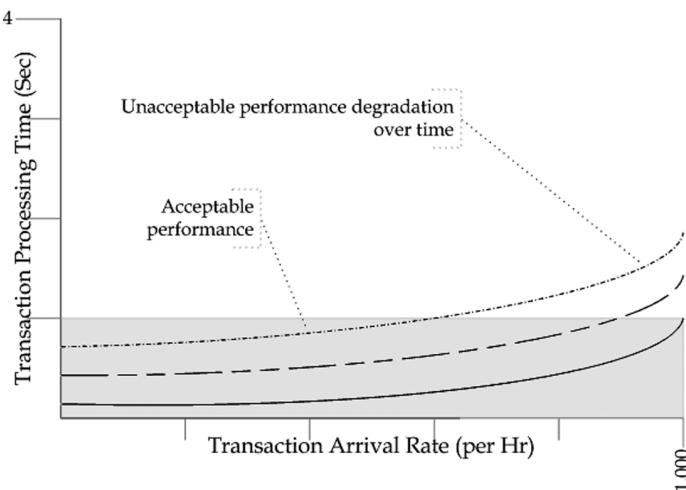


Figure 4–8 Response that degrades over time example

In Figure 4–8, we show several curves. The first, solid, line shows a sample run early in the test. The next, dashed, line shows a run that was made somewhat later, and the dotted line shows a run made even later into the test.

What we are seeing here is a system that is degrading with time. The exact same load run later in the test was markedly slower than the previous run, and the third run was worse yet.

This looks like a classic case of memory leaking, or disk access slowing down due to fragmentation. Notice that there is no knee in this graph; no sudden dislocation of resources. It is just a balloon losing air; eventually, if the system kept running, we would expect that performance would eventually reach unacceptable levels even at low loading.

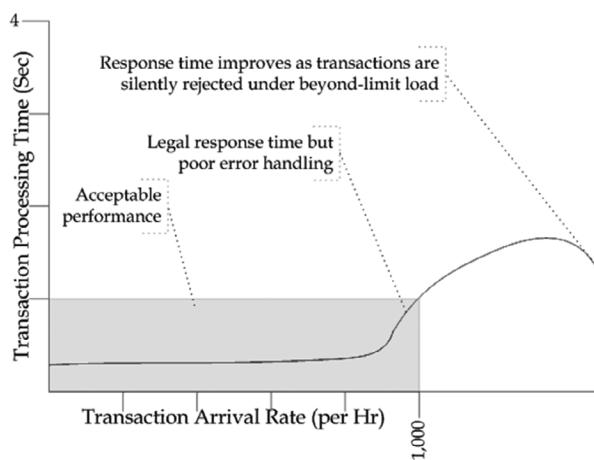


Figure 4–9 Inadequate error handling example

Finally, in Figure 4–9, we see a system that does not look too bad right up until it is fairly heavily loaded. At about 900 transactions per hour, we see a knee where response starts rapidly rolling off. Is this good or bad? Anytime you see a graph or are presented with metrics, remember that everything is relative. It might be really good if the server was rated at 500 transactions per hour, but in this case we want to achieve 1,000 transactions per hour.

Suppose it were rated at 900 transactions per hour, and it is [barely] in tolerance; what else does the graph show? Looking at the legend on the graph, the assumption is that error handling is problematic. The real concern should be seen as what is happening at the very tail end of the curve. The only way the curve can go down after being in the unacceptable range is if it is sloughing off requested transactions. In other words, more transactions are being requested, but the system is denying them. These transactions may be explicitly denied (not good but understandable to the user) or simply lost (which would be totally unacceptable).

Possible causes of the symptoms might be insufficient resources, queues and stacks that are too small, or time-out settings that are too short.

Ideally, performance testing will be run with experts standing by to investigate anomalies. Unlike other testing, where we might write an incident report to be read sometime later by the developer, the symptoms of performance test failures are often investigated right away, while the test continues. In this case, the server, network, and database experts are liable to be standing by to troubleshoot the problems right away.

We will discuss the tools that they are likely to be using in Chapter 6.

4.4.3.1 Internal Time Behavior Metrics

Response time: A measure of the estimated time to perform a given task. This measurement estimates the time it will take for a specific action to complete based on the efficiency of the operating system and the application of the system calls. Any of the following might be estimated:

- All (or parts) of the design specifications
- Complete transaction path
- Complete modules or parts of the software product
- Complete system during test phase

Clearly, a shorter time is better. The inputs to this measurement are the known characteristics of the operating system added to the estimated time in system calls. Both developers and analysts would be targets for this measurement.

Throughput time: A measure of the estimated number of tasks that can be performed over a unit of time. This is measured by evaluating the efficiency of the resources of the system that will be handling the calls as well as the known characteristics of the operating system. The greater this number, the better. Both developers and analysts would be targets for this measurement.

Turnaround time: A measure of the estimated time to complete a group of related tasks performing a specific job. As in the other time-related metrics, we need to estimate the operating system calls that will be made and the application system calls involved. The shorter the time, the better. The same entities listed for response time can all be estimated for this metric. Both developers and analysts would be targets for this measurement.

4.4.3.2 External Time Behavior Metrics

ISO 9126-2 describes several external metrics for efficiency. Remember that these are to be measured during actual dynamic testing or operations. The standard emphasizes that these should be measured over many test cases or intervals and averaged since the measurements fluctuate depending on conditions of use, processing load, frequency of use, number of concurrent users, and so on.

Response time: A measure of the time it takes to complete a specified task. Alternately, how long does it take before the system responds to a specified operation? To measure this, record the time (T1) a task is requested. Record the time that the task is complete (T2). Subtract T1 from T2 to get the measurement. Sooner is better.

Mean time to response: A measure of the average response time for a task to complete. Note that this metric is meaningful only when it is measured while the task is performed within a specified system load in terms of concurrent tasks and system utilization. To calculate this value, execute a number of scenarios consisting of multiple concurrent tasks to load the system to a specified value. Measure the time it takes to complete the specified tasks. Then calculate the metric using the formula

$$X = T_{\text{mean}} / T_{X\text{mean}}$$

where T_{mean} is the average time to complete the task (for N runs) and $T_{X\text{mean}}$ is the required mean time to response. The required mean time to response can be derived from the system specification, from user expectation of business needs, or through usability testing to observe the reaction of users. This value will evaluate as $0 \leq X \leq 1.0$. We would want this to measure less than but close to 1.0.

Worst case response time: A measure of the absolute limit on the time required to complete a task. This metric is subjective, in that it asks if the user will always get a reply from the system in a time short enough to be tolerable for that user. To perform this measurement, emulate a condition where the system reaches maximum load. Run the application and trigger the task a number of times, measuring the response each time. Calculate the metric using the formula

$$X = T_{\text{max}} / R_{\text{max}}$$

where T_{max} is the maximum time any one iteration of the task took and R_{max} is the maximum required response time. This value will evaluate as $0 \leq X \leq 1.0$. We would want this to measure less than but close to 1.0.

Throughput: A measure of how many tasks can be successfully performed over a given period of time. Notice that these are likely to be different tasks, all being executed concurrently at a given load. To calculate this metric, use the formula

$$X = A / T$$

where A is the number of completed tasks and T is the observational time period. The larger the value, the better. As before, this measurement is most interesting when comparing the mean and worst case throughput values as we did with response time.

Mean amount of throughput: The average number of concurrent tasks the system can handle over a set unit of time, calculated using the same formula ($X = X_{\text{mean}} / R_{\text{mean}}$), where X_{mean} is the average throughput and R_{mean} is the required mean throughput.

Worst case throughput ratio: The absolute limit on the system in terms of the number of concurrent tasks it must perform. To calculate it, use the same formula we saw earlier ($X = T_{\text{max}} / R_{\text{max}}$), where T_{max} is the worst case time of a single task and R_{max} is the required throughput.

Turnaround time: A measure of the wait time the user experiences after issuing a request to start a group of related tasks until their completion. As we might expect, this is most meaningful when we look at the mean and worst case turnaround times as follows.

Mean time for turnaround: The average wait time the user experiences compared to the required turnaround time as calculated by ($X = T_{\text{mean}} / R_{\text{mean}}$). This should be calculated at a variety of load levels.

Worst case turnaround time ratio: The absolute acceptable limit on the turnaround time, calculated in the same way we saw before ($X = T_{\text{max}} / R_{\text{max}}$).

Waiting time: A measure of the proportion of time users spend waiting for the system to respond. We execute a number of different tasks at different load levels and measure the time it takes to complete the tasks. Then, calculate the metric using the formula

$$X = T_a / T_b$$

where T_a is the total time the user spent waiting and T_b is the actual task time when the system was busy. The measurement will evaluate as $0 \leq X$. An efficient system, able to multitask efficiently, will have a waiting time of close to zero.

4.4.4 Resource Utilization

For many systems, including real-time, consumer-electronics, and embedded systems, resource usage is important. You can't always just add a disk or add memory when resources get tight, as the NASA team managing one of the Mars missions found out when storage space ran out.²⁶

Resource utilization uses measurements of actual or projected resource usage that it takes to perform a task.

Inside those categories, there are dozens of different metrics that can be captured. Here we have listed some of the most important metrics that an organization might want to collect.²⁷

- Processor utilization percentage at key points of the test.
- Available memory, both RAM and virtual, at different points through the test. That includes memory page usage.
- Top n processes active—remembering that some of them may not be part of the test but may be internal or external processes running concurrently.
- Length on queues (processor, disk, etc.) at any given time.
- Disk saturation and usage
- Network errors—both in- and outbound

Remember, too many measurements waste time and resources, too few and you don't learn what you need to know. Metrics are a science, an art, and perhaps the most frustrating thing we deal with when testing. Following are recommended ISO 9126 measurements in which an organization might be interested.

4.4.4.1 Internal Resource Utilization Metrics

All of these metrics are targeted at developers in reviews.

I/O utilization: A measure of the estimated I/O utilization to complete a specified task. The value of this metric is the number of buffers that are expected to be required (calculated or simulated). The smaller this value is, the better. Note that each task will have its own value for this metric.

I/O utilization message density: A measure of the number of error messages relating to I/O utilization in the lines of code responsible for making system calls. To calculate this value, count the number of error messages pertaining to

26. A memory shortage caused the *Spirit* Mars rover to become unresponsive on January 2, 2004. A brief summary can be found at: http://www.computerworld.com/s/article/89829/Out_of_memory_problem_caused_Mars_rover_s_glitch.

27. These come from a book by Ian Molyneaux, *The Art of Application Performance Testing*.

I/O failure and warnings and compare that to the estimated number of lines of code involved in the system calls using the formula

$$X = A / B$$

where A is the number of I/O related error messages and B is the number of lines of code directly related to system calls. The greater this size, the better.

Memory utilization: A measure of the estimated memory size that the software system will occupy to complete a specified task. This is a straightforward estimation of the number of bytes. Each different task should be estimated. As expected, the smaller this estimated footprint, the better.

Memory utilization message density: A measure of the number of error messages relating to memory usage in the lines of code responsible for making the system calls that are to be used. To calculate this metric, count the number of error messages pertaining to memory failure and warnings and compare that to the number of lines of code responsible for the system calls, using the formula

$$X = A / B$$

where A is the number of memory-related error and warning messages and B is the number of lines of code directly related to the system calls. The greater this ratio, the better.

Transmission utilization: A measure of the amount of transmission resources that will likely be needed based on an estimate of the transmission volume for performing tasks. This metric is calculated by estimating the number of bits that will be transmitted by system calls and dividing that by the time needed to perform those calls. As always, because this is an internal value, these numbers must all be either calculated or simulated.

4.4.4.2 External Resource Utilization Metrics

External resource utilization metrics allow us to measure the resources that the system consumes during testing or operation. These metrics are usually measured against the required or expected values.

I/O devices utilization: A measure of how much the system uses I/O devices compared to how much it was designed to use. This is calculated using the formula

$$X = A / B$$

where A is the amount of time the devices are occupied and B is the specified time the system was expected to use them. Less than and nearer to 1.0 is better.

I/O related errors: A measure of how often the user encounters I/O-type problems. This is measured at the maximum rated load using the formula

$$X = A / T$$

where A is the number of warning messages or errors encountered and T is the user operating time. The smaller this measure, the better.

Mean I/O fulfillment ratio: A measure of the number of I/O-related error messages and/or failures over a specified length of time at the maximum load. This is compared to the required mean using the formula

$$X = A_{\text{mean}} / R_{\text{mean}}$$

where A_{mean} is the average number of I/O error messages and failures over a number of runs and R_{mean} is the required²⁸ average number of I/O-related error messages. This value will evaluate as $0 \leq X \leq 1$. Lower is better. For example, assume that we allow 10 error messages over a certain period, and we actually average 2. That would give us an X value of 0.2. On the other hand, if we average 9 error messages over that period, we would get an X of 0.9.

User waiting time of I/O devices utilization: A measure of the impact of I/O utilization on the waiting time for users. This is a simple measurement of the waiting times required while I/O devices operate. As you might expect, the shorter this waiting time, the better. This should be measured at the rated load.

Maximum memory utilization: A measure of the absolute limit on memory required to fulfill a specific function. Despite its name, this measurement actually looks at error messages rather than the number of bytes needed in memory. This is measured at maximum expected load using the formula

$$X = A_{\text{max}} / R_{\text{max}}$$

where A_{max} is the maximum number of memory-related error messages (taken from one run of many) and R_{max} is the maximum (allowed) number of memory-related error messages. The smaller this value, the better.

Mean occurrence of memory error: A measure of the average number of memory related error messages and failures over a specified length of time and specified load on the system. We calculate this metric using the same formula as before:

$$X = A_{\text{mean}} / R_{\text{mean}}$$

28. This appears to be an awkward translation. It would not make much sense to require a system to average N number of error messages over a certain period. Perhaps a better term would be *maximum allowed* number of error messages. We will use that phrase in future metrics.

where Amean is the average number of memory error messages over a number of runs and Rmean is the maximum allowed mean number of memory-related error messages. The measure will evaluate as $0 \leq X$; the lower, the better.

Ratio of memory error/time: A measure of how many memory errors occurred over a given period of time and specified resource utilization. This metric is calculated by running the system at the maximum rated load for a specified amount of time and using the formula

$$X = A / T$$

where A is the number of memory-related warning messages and system errors that occurred and T is the amount of time. The smaller this value, the better.

Maximum transmission utilization: A measure of the actual number of transmission-related error messages to the allowed (required) number while running at maximum load. This metric is calculated using the formula

$$X = A_{\max} / R_{\max}$$

where A_{max} is the maximum number of transmission-related error messages (taken from one run of many) and R_{max} is the maximum (allowed) number of transmission-related error messages. The smaller this value, the better.

Mean occurrence of transmission error: A measure of the average number of transmission-related error messages and failures over a specified length of time and utilization. This is measured while the system is at maximum transmission load. Run the application under test and record the number of errors due to transmission failure and warnings. The calculation uses the formula

$$X = A_{\text{mean}} / R_{\text{mean}}$$

where A_{mean} is the average number of transmission-related error messages and failures over multiple runs and R_{mean} is the maximum allowed number as defined earlier. The smaller the number, the better.

Mean of transmission error per time: A measure of how many transmission-related error messages were experienced over a set period of time and specified resource utilization. This value is measured while the system is running at maximum transmission loading. The application being tested is run and the number of errors due to transmission failures and warnings are recorded. The metric is calculated using the formula

$$X = A / T$$

where A is the number of warning messages or system failures and T is the operating time being measured. Smaller is better for this metric.

Transmission capacity utilization: A measure of how well the system is capable of performing tasks within the expected transmission capacity. This is measured while executing concurrent tasks with multiple users, observing the transmission capacity, and comparing it to specified values using the formula

$$X = A / B$$

where A is the measured transmission capacity and B is the specified transmission capacity designed for the software to use. Less than and nearer to one is better.

4.4.5 Compliance

4.4.5.1 Internal Compliance Metric

ISO 9126-3 defines efficiency compliance as a measure of how compliant the system is estimated to be against applicable regulations, standards, and conventions. To calculate this, we use the formula

$$X = A / B$$

where A is the number of items related to efficiency compliance that are judged—in reviews—as being correctly implemented and B is the total number of compliance items. The closer this value is to one, the more compliant it is.

4.4.5.2 External Compliance Metric

Finally, ISO 9126-2 defines an external efficiency compliance metric. This is a measure of how compliant the efficiency of the product is with respect to applicable regulations, standards, and conventions. Calculation of this metric is done using the formula

$$X = 1 - A / B$$

where A is the number of efficiency compliance items that have not been implemented during testing and B is the total number of efficiency compliance items that have been specified for the product. The closer to one, the better.

4.4.6 Exercise: Efficiency Testing

Using the HELLOCARMS system requirements document, analyze the risks and create an informal test design for efficiency testing.

4.4.7 Exercise: Efficiency Testing Debrief

We selected requirement 040-010-080. This requirement states that “once a Senior Banker has made a determination, the information shall be transmitted to the Telephone Banker within two (2) seconds.”

Once again, we would use automation to test this requirement. This one interests us because of the issue of measuring time on two different workstations. If their real-time clocks were set to appreciably different times, then any measurements that we could make would be suspect.

Jamie actually had a similar problem a few years ago that he had to solve; we would use the same solution here. The solution consists of writing a simple listener application on a separate workstation. When the Telephone Banker’s workstation is triggered to escalate a loan to the Senior Banker, a message is sent to the listener, which logs it in a text file using a time stamp from its own real-time clock. Automation on the Senior Banker workstation will handle the request. When it finishes, it sends a message to the same listener, which logs it in the same file, again with a time stamp. When the Telephone Banker automation, which has been in a waiting state for the return, gets the notification, it sends another message to the listener.

Note that this same test would also satisfy the conditions necessary to test requirement 040-010-070, which requires no more than a one-second delay for the escalation to occur.

We are disregarding the transport time from both automated workstations. This may be problematic, but we are going to assume (with later testing to confirm) that three local test workstations in the same lab running on the same network will incur pretty much the same transport time, canceling them out. Even so, because the times we are testing are relatively large (one second and two seconds), we believe the testing would likely be valid.

4.5 Maintainability Testing

Learning objectives

Only common learning objectives.

Maintainability refers to the ability to update, modify, reuse and test the system. This is important for most systems because most will be updated, modified, and tested many times during their life. Often pieces of systems and even whole systems are used in new and different situations.

Why all of the changes to a system? Remember that when we discussed reliability, we said that software does not wear out, but it does become obsolete. We will want new and extended functionality. There will also be patches and updates to make the system run better. New environments will be released that we must adapt to, and interoperating systems will be updated, usually requiring updates on our system.

Jamie remembers one of his first software experiences; they had worked for over six months putting together and delivering the new system. Jamie was so glad to see it go that he spoke out loud, “Hope I never see that software again!” Everyone laughed at him, not believing that he did not know how often that boomerang was going to come back at them.

What Jamie did not know at the time was that only a small fraction of the overall cost of a system was spent in the original creation and rollout. On the day you ship that first release, you can be pretty sure that 80 percent or more of the eventual cost has not yet been incurred. Or, as the Terminator said, “I’ll be back.”

In the ISTQB Foundation level syllabus, it was discussed that we could not do maintenance testing on a system that was not already in production. After we ship the first time, we start what some call the SMLC: software maintenance life cycle.

Okay, quick! Just off the top of your head, come up with a dynamic maintainability test for HELLOCARMS. We’ll wait. Hmmmmmmmm.

Tough to do, isn’t it?

The simple fact is that much of maintainability testing is not going to be done by scripting test cases and then running them when the code gets delivered. Many, if not most, maintainability defects are invisible to dynamic testing.

Maintainability defects include hard-to-understand code, environment dependencies, hidden information and states, and excessive complexity. They

ISTQB Glossary

analyzability: The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.

changeability: The capability of the software product to enable specified modifications to be implemented.

maintainability testing: The process of testing to determine the maintainability of a software product.

stability: The capability of the software product to avoid unexpected effects from modifications in the software.

testability: The capability of the software product to enable modified software to be tested.

can also include “painted yourself into a corner” problems when software is released without any practical mechanism for updating it in the field. For example, think of all the problems Microsoft had stabilizing their security-patch process in the mid-2000s.

Design problems. Conceptual problems. Standards and guidelines (or lack of same) compliance. We have a good way of finding these kinds of issues. It’s called static testing. From the first requirements to the latest patch, there is likely no better way to ensure that the system is maintainable. This is one case where a new tool, a couple of scripted tests, or reasonably attentive testers are not going to magically transform the pig’s ear into a silk purse.

Management must be made to understand the investment that good maintainability is going to entail. This must be seen as a long-term investment because much of the reward is going to come down the road. And, it is the worst kind of investment to try to sell—one that is mostly invisible. If we do a good job building a maintainable system, how do we show management the rewards in a physical, tangible way?

Well, we won’t have as many patches, but we can’t prove without a doubt it was due to the investment. Our maintenance programmers will make fewer mistakes, leading to fewer regression bugs, but we can’t necessarily point to the mistakes we did not make.

This is not just theoretical. Jamie was a test lead in a small start-up organization and he kept on arguing to put some standards and guidelines around the code. To spend some of the design time thinking about making sure that the application was going to be changeable. To spend some extra time documenting

the assumptions we were making. To write self-documenting code using naming conventions. For each argument he made, he was challenged to prove the payback in empirical terms. He pretty much failed and the system that was built turned out to be a nightmare.

To say that we need to test the above-mentioned issues is not to say you shouldn't test updates, patches, upgrades, and migration. You definitely should. You should not only test the software pieces, but also the procedures and infrastructure involved. For example, verifying that you can patch a system with a 100 MB patch file is all well and good until you find you have forgotten that real users will have to download this patch through a slow and balky Internet connection and will need a PhD in computer science to hand-install half of the files!

Here are a few of the project issues that exacerbate maintainability problems.

Schedules: Get the system out the door. Push it, prod it, nudge it, just get it out the door! Is it maintainable? Come on, we don't have time for joking. Often, the general consensus of the team seems to be that we can always fix it when we have time. Let us ask the question that needs to come after that statement. Do we ever have time? We get this thing out the door, the next project is right there, filling our inbox. In our entire careers, we have never enjoyed that downtime that we were led to expect when we could catch up on the things we shelved.

Frankly, this may not be the fault of the team, entirely. The human brain seems to have hard wiring in it for this short-term gratification versus long-term benefit calculation.

If I eat this cookie now, I intellectually know that I will have to work out—sometime later—much harder than I like to. But what the heck—the cookie looks so good right now.²⁹

Testers must learn to push the idea of short-term pain, long-term value when it comes to maintainability.

Optimism: If we can get it to work now, it likely will always work. Why invest a lot of effort into improving the maintainability since we probably won't have any problems with it.

We wish we had a nickel for every time we heard a development manager (or project manager) say that they have hired the very best people available so

29. An amusing take on this, the “marshmallow test,” can be found at <http://www.newyorker.com/magazine/2009/05/18/dont-2>.

of course it will work. Robert Heinlein once wrote about the optimism of a religious man sitting in a poker game with four aces in the hole...

Of course, when the project doesn't work out successfully, it must have been the failure of the testers. And the regression bugs were simply one-time things. And on and on. While each incident is unique, the pattern of failures isn't.

Contracts are often the problem. The contract might specify minimum functionality that has to be delivered. We don't have time to build a better system—it is not what they asked for. We low-balled the estimate to get the job so we can't afford to make it good too.

Initiation: One issue that we have seen repeatedly is the idea of *initiation into the club*. Many developers start their career doing maintenance programming on lousy systems. They have "paid their dues"! One might think that this would teach the necessity of building a maintainable system—and sometimes it does. But often we run into the mind-set of "we had to do it; you should have to do it."

Jamie's wife is a nurse, so he has had the chance to socialize with a number of doctors at holiday parties, picnics, and such. You might be surprised how little sympathy there is for interns and residents who often have to work 36- to 48-hour shifts. The prevailing opinion of many doctors is that "we had to do it and we survived—they should do it also." It goes with the territory! Frankly, we hate that phrase.

Lack of ownership is clearly a problem. Maintainability, as well as quality, should be owned by the team as a whole, but rarely is.

Short timer syndrome: And finally, this one is probably not as big a problem as many think. We have occasionally heard that, "I won't be here anymore when the bill comes due." We call that short-timer thinking and used to see it in the United States military during the sixties and seventies when the military draft was the norm rather than voluntary service. "I'm outa here in 17 days. I'm so short I can't see over a nickel if I was standing on a dime!"

There are probably a dozen more issues that we have not thought of. The fact is that education is the solution to many of the reasons given for ignoring maintainability. But, we have to make sure that the reasons we give to insist on maintainable development are colored green. It is about dollars or euros or pounds, or whatever term you think in. Money is the reason we should care about maintainability. Time and resources are important, but the tie-in to cost must be made for management to care.

Because maintainability is such a broad category, perhaps the best way to discuss it is to break it up into its subcategories as defined by the ISO 9126 standard. These include analyzability, changeability, stability, testability, and compliance.

4.5.1 Analyzability

The definition of *analyzability*, as given in ISO 9126, is the capability of the software product to be diagnosed for deficiencies or causes of failures or for the parts to be modified to be identified. In other words, how much effort will it take to diagnose defects in the system or to identify where changes can be made when needed?

Here are four common causes of poor analyzability in no particular order.

In the old days, we called it spaghetti code. Huge modules tied together with GOTO or jump constructs. No one we know still writes code like that, but some techniques are still being used that are not much better.

One of the basic tenets of Agile programming is a tactic called refactoring. Part of refactoring is that, when you do something more than once, you rewrite the code to create a callable function and then call it in each place it is needed. This is a great idea that every programmer should follow. Instead, what many programmers do is copy and paste code they want to reuse. Each module then begins to be a junior version of spaghetti code. Code should be modular. Modules should be relatively small, unless speed is of paramount issue. Each module should be understandable. Thomas McCabe understood this when he came up with cyclomatic complexity.

Back in the 1990s, while working at a large, multinational company, Jamie worked with a group that completely rewrote the operating system for a very popular mini-computer (going from PL/MP to C++). One main intention was to create a library of C++ classes that were reusable throughout the operating system. Management found that the library was not really being used extensively, so they investigated. It turned out, the group was told, that when a programmer needed a particular class, they were likely to search through the library for less than 10 minutes before giving up and writing their own class. What made this confusing was that writing their own class might take several days, and at that point, they would have a class where the code was not yet debugged. Had they searched a little longer, they likely would have found a completely debugged module that supplied the functionality they needed. Management termed this behavior the “not created here” problem. Had the company spent money on a librarian/archivist, they likely would have had no issues.

The second reason for poor analyzability is lack of good documentation. Many organizations try to save time by limiting the documentation that is created. Or, sometimes when documentation is required, it is just done poorly. Or, after changes are made, the documentation is not updated. Or, documentation is not under version control so there are a dozen different versions of a docu-

ment floating around. Whatever the reason, good documentation helps us understand and analyze a system more easily.

The third reason for poor analyzability is poor—or nonexistent—standards and guidelines. Each programmer is liable to program in their own style—unless they are told to follow some kind of standards. These standards might include the following items:

- Indentation, white space, and other structure guidelines
- Naming conventions
- Modular guidelines (At the company mentioned earlier, our rule of thumb was that no module should be longer than one printed page.)
- Meaningful error messages and standard exception handling
- Meaningful comments

Clearly, following some standards would help us analyze the system more easily.

The fourth reason involves code abstraction. Code abstraction, theoretically, is a good thing; like all good things, however, you can get too much of it. Object-oriented code is supposed to have a level of abstraction that allows the developers to build good, inheritable classes. By hiding the gory details in superclasses, a developer can ensure that other developers who inherit from those classes don't depend on the details in their implementation. That way, if the implementation details have to change, it should not cause failures in the derived classes.

For example, suppose we supply a calculation for the sine of an angle. How that calculation works should not matter to any consumer of the calculation—as long as the calculation is correct. If we decide in a later release to change the way we make the calculation, it should not break any existing code that uses the value calculated. However, it is conceivable that a clever developer may decide to utilize some side effect of our original method of calculation because they understood how we originally made it. Now, changing our code is likely to break their code, and worse, we would not be aware of it.

The more abstraction there is in a module, however, the harder it is to understand exactly what is being done. Each organization must decide how much to abstract and how much to clarify.

The fix for most of these problems is the same. Good, solid standards and guidelines. Enforcement via static testing at all times—especially when time is pinched. No excuses. Organizations that make it clear that poor analyzability is an important class of defects on its own that will not be tolerated often do not suffer from problems with this quality subcharacteristic.

4.5.1.1 Internal Analyzability Metrics

ISO 9126-3 defines a number of internal maintainability metrics that an organization may wish to track. These should help predict the level of effort required when modifying the system.

Activity recording: A measure of how thoroughly the system status is recorded. This is calculated by counting the number of items that are found to be written to the activity log as specified compared to how many items are supposed to be written based on the requirements. The calculation of the metric is made using the formula

$$X = A / B$$

where A is the number of items implemented that actually write to the activity log as specified (as confirmed in review) and B is the number of items that should be logged as defined in the specifications. The closer this value is to one, the more complete the logging is expected to be.

Readiness of diagnostic function: A measure of how thorough the provision for diagnostic functions is. A diagnostic function analyzes a failure and provides an output to the user or log with an explanation of the failure. This metric is a ratio of the implemented diagnostic functions compared to the number of required diagnostic functions in the specifications and uses the same formula to calculate it:

$$X = A / B$$

A is the number of diagnostic functions that have been implemented (as found in review) and B is the required number from the specifications. This metric can also be used to measure failure analysis capability in the system and ability to perform causal analysis.

4.5.1.2 External Analyzability Metrics

ISO 9126-2 defines external maintainability metrics that an organization may wish to track. These help measure such attributes as the behavior of the maintainer, the user, or the system when the software is maintained or modified during testing or maintenance.

Audit trail capability: A measure of how easy it is for a user (or maintainer) to identify the specific operation that caused a failure. ISO 9126-2 is somewhat abstract in its explanation of how to record this metric: it says to observe the user

or maintainer who is trying to resolve failures. This would appear to be at odds with the way of calculating the metric, which is to use the formula

$$X = A / B$$

where A is the number of data items that are actually logged during the operation and B is the number of data items that should be recorded to sufficiently monitor status of the software during operation. As in many of the other metrics in this standard, an organization must define for itself exactly what the value for B should be. How many of the different error conditions that might occur should actually be logged? A value for X closer to one means that more of the required data is being recorded. An organization must see this as a tradeoff; the more we want to log, the more code is required to be written to monitor the execution, evaluate what happened, and record it.

Diagnostic function support: A measure of how capable the diagnostic functions are in supporting causal analysis. In other words, can a user or maintainer identify the specific function that caused a failure? Like the previous metric, the method of application merely says to observe the behavior of the user or maintainer who is trying to resolve failures using diagnostic functions. To calculate, use the formula

$$X = A / B$$

where A is the number of failures that can be successfully analyzed using the diagnostic function and B is the total number of registered failures. Closer to one is better.

Failure analysis capability: A measure of the ability of users or maintainers to identify the specific operation that caused a failure. To calculate this metric, use the formula

$$X = 1 - A / B$$

where A is the number of failures for which the causes are still not found and B is the total number of registered failures. Note that this is really the flip side of the previous metric diagnostic function support. This metric is a measure of how many failures we could not diagnose, where the previous metric is a measure of how many we did. Closer to 1.0 is better.

Failure analysis efficiency: A measure of how efficiently a user or maintainer can analyze the cause of failure. This is essentially a measure of the average amount of time used to resolve system failures and is calculated by the formula

$$X = \text{Sum}(T) / N$$

where T is the amount of time for each failure resolution and N is the number of problems resolved. Two interesting notes are included in the standard for this metric. ISO 9126-2 says that only the failures that are successfully resolved should be included in this measurement; however, it goes on to say that failures not resolved should also be measured and presented together. It also points out that person-hours rather than simply hours might be used for calculating this metric so that effort may be measured rather than simply elapsed time.

Status monitoring capability: A measure of how easy it is to get monitored data for operations that cause failures during the actual operation of the system. This metric is generated by the formula

$$X = 1 - A / B$$

where A is the number of cases where the user or maintainer tried but failed to get monitor data and B is the number of cases where they attempted to get monitored data during operation. The closer to one the better, meaning they were more successful in getting monitor data when they tried.

4.5.2 Changeability

The second subcharacteristic for maintainability is *changeability*. The definition in ISO 9126 is the capability of the software product to enable a specified modification to be implemented. Once again, no dynamic test cases come to mind to ensure changeability. There are a number of metrics that ISO 9126 defines, but they are all retrospective, essentially asking, “Was the software changeable?” after the fact.

Virtually all of the factors that influence changeability are design and implementation practices.

Problems based on design include coupling and cohesion. *Coupling* and *cohesion* are terms that reference how a system is split into modules. Larry Constantine is credited with pointing out that high coupling and low cohesion are detriments to good software design.

Coupling refers to the degree that modules rely on the internal operation of each other during execution. High coupling could mean that there are a lot of dependencies and shared code between modules. Another possibility is that one module depends on the *way* the other module does something. If that dependency is changed, it breaks the first module. When high coupling is allowed, it becomes very difficult to make changes to a single module; changes *here* generally mean that there are more changes *there* and *there* and *there*. Low coupling

is desirable; each module has a task to do and is essentially self-contained in doing it.

There are a number of different types of coupling:

- Content coupling: when one module accesses local data in another.
- Common coupling: when two or more modules share global data.
- External coupling: when two or more modules share an external interface or device.
- Control coupling: when one module tells another module what to do.
- Data-structure coupling: when multiple modules share a data structure, each using only part of it.
- Data coupling: when data is passed from one module to another—often via parameters.
- Message coupling: when modules are not dependent on each other; they pass messages without data back and forth.
- No coupling: when modules do not communicate at all.

Some coupling is usually required; modules usually have to be able to communicate. Generally, message coupling or data coupling would be the best options here.

Cohesion, on the other hand, describes how focused the responsibilities of the module are. In general, a module should do a single thing and do it well. Indicators that cohesion is low include when there are many functions or methods in the module that do different things that are unrelated or when they work on completely different sets of data.

In general, low coupling and high cohesion go together and are a sign that changeability is going to be good. High coupling and low cohesion are generally symptoms of a poor design process and an indicator of poor changeability.

Changeability problems caused by improper implementation practices run the gamut of many of the things we were told not to do in programming classes.

Using global variables is one of the biggest problems; it causes a high degree of coupling in that a change to the variable in one module may have any number of side effects in other modules.

Hard-coding values into a module should be considered a serious potential bug. Named constants are a much better way for developers to write code; when they decide to change the value, all instances are changed at once. When hard-coded values (which some call magic numbers) are used, invariably some get changed and some don't.

Hard coding design to the hardware is also a problem. In the interest of speed, some developers like to program right down to the metal of the platform they are writing for. That might mean using implementation details of the operating system, hardware platform, or device that is being used. Of course, when time passes and hardware, operating systems and/or devices change, the software is now in trouble.

The more complex the system, the worse changeability is affected. As always, there is a trade-off in that sometimes, we need the complexity.

The project Jamie was on that we mentioned earlier, rewriting a mid-range computer operating system, was an example of software engineering done right! Every developer and most testers were given several weeks of full-time object-oriented development and design training. They were moving 25 million lines of PL/MP (an elegant procedural language) code to C++, not as a patch but a complete rewrite. They spent a lot of time coming up with standards and guidelines that every developer had to follow. They made a huge investment in static testing with training for everyone.

Low coupling and *high cohesion* were the buzzwords du jour. Perhaps *buzzword* is incorrect; they truly believed in what they were doing. Maintainability was the rule.

Their rule of thumb was to limit any method, function, or piece of code to what would fit on one page printed out. Maybe 20 to 25 lines of code. They used good object-oriented rules with classes, inheritance, and data hiding. They used messaging between modules to avoid any global variables. They had people writing libraries of classes and testing the heck out of them so they could really get reuse.

So you might assume that everything went fabulously. Well, the final result was outstanding, but there were more than a few stumbles along the way. They had one particular capability that the operating system had to deliver; their estimate was that using the new processor, they had to complete the task within 7,000 CPU cycles. After the rewrite, they found that it took over 99,000 CPU cycles to perform the action. They were off by a huge amount. And, because this action might occur thousands of times a minute, their design needed to be completely rethought. The fact is low coupling, high cohesion, inheritance, and data hiding have their own costs.

Every design decision has trade-offs. For many systems, good design and techniques are worth every penny we spend on them. But when speed of execution is paramount, very often those same techniques do not work well. In order to make this system work, they had to throw out all of the rules. For this mod-

ule, they went back to huge functions with straight procedural code, global variables, low cohesion, and high coupling. And, they made it fast enough. However, as you might expect, it was very trouble prone; it took quite a while to stabilize it.

Poor documentation will adversely affect changeability. Where a maintenance programmer must guess at how to make changes, to infer as to what the original programmer was thinking, changeability is going to be greatly impacted. Documentation includes internal (comments in the code, self-documenting code through good naming styles, etc.) and external documentation, including high- and low-level design documents.

4.5.2.1 Internal Changeability Metrics

Internal changeability metrics help in predicting the maintainer's or user's effort when they're trying to implement a specified modification to the system.

Change recordability: A measure of how completely changes to specifications and program modules are documented. ISO 9126-3 defines these as change comments in the code or other documentation. This metric is calculated with the formula

$$X = A / B$$

where A is the number of changes that have comments (as confirmed in reviews) and B is the total number of changes made. The value should be $0 \leq X \leq 1$, where the closer to one, the better. A value near 0 indicates poor change control.

4.5.2.2 External Changeability Metrics

External changeability metrics help measure the effort needed when you're trying to implement changes to the system.

Change cycle efficiency: A measure of how likely a user's problem can be solved within an acceptable time period. This is measured by monitoring the interaction between the user and the maintainer and recording the time between the initial user's request and the resolution of their problem. The metric is calculated by using the formula

$$Tav = \text{Sum}(Tu) / N$$

where Tav is the average amount of time, Tu is the elapsed time for the user between sending the problem report and receiving a revised version, and N is the number of revised versions sent. The shorter Tav is, the better; however, large

numbers of revisions would likely be counterproductive to the organization, so a balance should be struck.

Change implementation elapse time: A measure of how easily a maintainer can change the software to resolve a failure. This is calculated by using the formula

$$Tav = \frac{\text{Sum}(Tm)}{N}$$

where Tav is the average time, Tm is the elapsed time between when the failure is detected and the time that the failure cause is found, and N is the number of registered and removed failures. As before, there are two notes. Failures not yet found should be excluded, and effort (in person-hours) may be used instead of elapsed time. Shorter is better.

Modification complexity: This is also a measure of how easily a maintainer can change the software to solve a problem. This calculation is made using the formula

$$T = \frac{\text{Sum}(A / B)}{N}$$

where T is the average time to fix a failure, A is work time spent to change a specific failure, B is the size of the change, and N is the number of changes. The size of the change may be the number of code lines changed, the number of changed requirements, the number of changed pages of documentation, and so on. The shorter this time the better. Clearly, an organization should be concerned if the number of changes are excessive.

Software change control capability: A measure of how easily a user can identify a revised version of the software. This is also listed as how easily a maintainer can change the system to solve a problem. It is measured using the formula

$$X = A / B$$

where A is the number of items actually written to the change log and B is the number of change log items planned such that we can trace the software changes. Closer to one is better, although if there are few changes made, the value will tend toward zero.

4.5.3 Stability

The third subcharacteristic of maintainability is *stability*, defined as the ability of the system to avoid unexpected effects from modifications of the software. After we make a change to the system, how many defects are going to be generated simply from the change?

This subcharacteristic is essentially the side effect of all of the issues we dealt with in changeability. The lower the cohesion, the higher the coupling, and the worse the programming styles and documentation, the lower the stability of the system.

In addition, we need to consider the quality of the requirements. If the requirements are well delineated, well understood, and competently managed, then the system will tend to be more stable. If they are constantly changing and poorly documented and understood, then not so much.

System timing matters to stability. In real-time systems or when timing is critical, change will tend to throw timing chains off, causing failures in other places.

4.5.3.1 Internal Stability Metrics

Stability metrics help us predict how stable the system will be after modification.

Change impact: A measure of the frequency of adverse reactions after modification of the system. This metric is calculated by comparing the number of adverse impacts that occur after the system is modified to the actual number of modifications made, using the formula:

$$X = 1 - A / B$$

where A is the number of adverse impacts and B is the number of modifications made. The closer to one, the better. Note that, since there could conceivably be multiple adverse conditions coming from a sloppy change-management style, this metric could actually be negative. For example, suppose one change was made, but three adverse reactions were noted. The calculation, using this formula would be as follows:

$$X = 1 - 3/1 ==> -2.$$

Modification impact localization: A measure of how large the impact of a modification is on the system. This value is calculated by counting the number of affected variables from the modification and comparing it to the total number of variables in the product using the formula

$$X = A / B$$

where A is the number of affected variables as confirmed in review and B is the total number of variables. The definition of an affected variable is any variable in a line of code or computer instruction that was changed. The closer this value is to zero, the less the impact of modification is likely to be.

4.5.3.2 External Stability Metrics

Change success ratio: A measure of how well the user can operate the software system after maintenance without further failures. There are two formulae that can be used to measure this metric:

$$X = Na / Ta$$

$$Y = \{(Na / Ta) / (Nb / Tb)\}$$

Na is the number of cases where the user encounters failures after the software is changed, Nb is the number of times the user encounters failures before the software is changed, Ta is the operation time (a specified observation time) after the software is changed, and Tb is the time (a specified observation time) before the software is changed. Smaller and closer to 0 is better. Essentially, X and Y represent the frequency of encountering failures after the software is changed. The specified observation time is used to try to normalize the metric so we can compare release to release metrics better. Also, ISO 9126-2 suggests that the organization may want to differentiate between failures that come after repair to the same module/function and failures that occur in other modules/functions.

Modification impact localization: This is also a measure of how well the user can operate the system without further failures after maintenance. The calculation uses the formula

$$X = A / N$$

where A is the number of failures emerging after modification of the system (during a specified period) and N is the number of resolved failures. The standard suggests that the “chaining” of the failures be tracked; that is, the organization should differentiate between a failure that is attributed to the change for a previous failure and failures that do not appear to be related to the change. Smaller and closer to zero is better.

4.5.4 Testability

Testability is defined as the capability of a software product to be validated after change occurs. This certainly should be a concern to all technical test analysts.

A number of issues can challenge the testability of a system.

One of our all-time favorites is documentation. When documentation is poor or nonexistent, testers have a very hard time trying to figure out what to test. When a requirement or functional specification clearly states that “the system works this way!” then we can test to validate that it does. When we have no

requirements, no previous system, no oracle as to what to expect, testing becomes a crap shoot. Is it working right? Shrug! Who knows?

Related to documentation is our old standby, lack of comments in the code and poor naming conventions, which make it harder to understand exactly what the code is supposed to do.

Implementing independent test teams can lead to unintentional (or even sometimes intentional) breakdowns in communications. Good communication between the test and development teams is important when dealing with testability.

Certain programming styles make the code harder to test. For example, object orientation was designed with data-hiding as one of its main objectives. Of course, data-hiding can also make it really difficult to figure out whether a test passed or not. And multiple levels of inheritance make it even harder; you might not know exactly where something happened, which class (object) actually was responsible for the action that was to be taken.

Lack of instrumentation in the code causes testability issues. Many systems are built with the ability to diagnose themselves; extra code is written to make sure that tasks are completed correctly and to log issues that occur. Unfortunately, this instrumentation is often seen as fluff rather than being required.

And as a final point, data issues can cause testability issues on their own. This is a case where better security and good encryption may make the system less testable. If you cannot find, measure, or understand the data, the system is harder to test. Like so much in software, intelligent trade-offs must be made.

4.5.4.1 Internal Testability Metrics

Internal testability metrics measure the expected effort required to test the modified system.

Completeness of built-in test function: A measure of how complete any built-in test capability is. To calculate this, count the number of implemented built-in test capabilities and compare it to how many the specifications call for. The formula to use is

$$X = A / B$$

where A is the number of built-in test functions implemented (as confirmed in review) and B is the number required in the specifications. The closer to one, the more complete.

Autonomy of testability: A measure of how independently the software system can be tested. This metric is calculated by counting the number of dependencies on other systems that have been simulated with stubs or drivers and comparing it to the total number of dependencies on other systems. As in many other metrics, the formula

$$X = A / B$$

is used, where A is the number of dependencies that have been simulated using stubs or drivers and B is the total number of dependencies on other systems. The closer to one the better. A value of one means that all other dependent systems can be simulated so the software can (essentially) be tested by itself.

Test progress observability: A measure of how completely the built-in test results can be displayed during testing. This can be calculated using the formula

$$X = A / B$$

where A is the number of implemented checkpoints as confirmed in review and B is the number required in the specifications. The closer to one, the better.

4.5.4.2 External Testability Metrics

Testability metrics measure the effort required to test a modified system.

Availability of built-in test function: A measure of how easily a user or maintainer can perform operational testing on a system without additional test facility preparation. This metric is calculated by the formula

$$X = A / B$$

where A is the number of cases in which the maintainer can use built-in test functionality and B is the number of test opportunities. The closer to one, the better.

Re-test efficiency: A measure of how easily a user or maintainer can perform operational testing and determine whether the software is ready for release or not. This metric is an average calculated by the formula

$$X = \text{Sum}(T) / N$$

where T is the time spent to make sure the system is ready for release after a failure is resolved and N is the number of resolved failures. Essentially, this is the average retesting time after failure resolution. Note that nonresolved failures are excluded from this measurement. Smaller is better.

Test restartability: A measure of how easily a user or maintainer can perform operational testing with checkpoints after maintenance. This is calculated by the formula

$$X = A / B$$

where A is the number of test cases in which the maintainer can pause and restart the executing test case at a desired point and B is the number of cases where executing test cases are paused. The closer to one, the better.

4.5.5 Compliance

4.5.5.1 Internal Compliance Metric

Maintainability compliance is a measure of how compliant the system is estimated to be with regard to applicable regulations, standards, and conventions. The ratio of compliance items implemented (as based on reviews) to those requiring compliance in the specifications is calculated using the formula

$$X = A / B$$

where A is the correctly implemented items related to maintainability compliance and B is the required number. The closer to one, the more compliant the system is.

4.5.5.2 External Compliance Metric

Maintainability compliance metrics measure how close the system adheres to various standards, conventions, and regulations. Compliance is measured using the formula

$$X = 1 - A / B$$

where A is the number of maintainability compliance items that were not implemented during testing and B is the total number of maintainability compliance items defined. Closer to one is better.

4.5.6 Exercise: Maintainability Testing

Using the HELLOCARMS system requirements document, analyze the risks and create an informal test design for maintainability, using one requirement as the basis.

4.5.7 Exercise: Maintainability Testing Debrief

Maintainability is an interesting quality characteristic for testers to deal with. Most maintainability issues are not amenable to our normal concept of a dynamic test, with input data, expected output data, and so on. Certainly some maintainability testing is done that way, when dealing with patches, updates, and so forth.

For this exercise, we are going to select requirement 050-010-010: Standards and guidelines will be developed and used for all code and other generated materials used in this project to enhance maintainability.

Our first effort, therefore, done as early as possible, would be to review the programming standards and guidelines with the rest of the test team and the development group—assuming, of course, that we have standards and guidelines. If there were none defined, we would try to get a cross-functional team busy defining them.

The majority of our effort would be during static testing. Starting (specifically for this requirement) at the low-level design phase, we would want to attend reviews, walk-throughs, and inspections. We would use available checklists, including Marick's, Laszlo's, and our own internal checklists (discussed in Chapter 5) based on defects found previously.

Throughout each review, we would be asking the same questions: Are we actually adhering to the standards and guidelines we have? Are we building a system that we will be able to troubleshoot when failures occur? Are we building a system with low coupling and high cohesion? Is it modular? How much effort will it take to test?

Since these standards and guidelines are not optional, we would work with the developers to make sure they understood them, and then we would start processing exceptions to them through the defect tracking database as with any other issues.

Beyond the standards and guidelines, there would still be some dynamic testing of changes made to the system, specifically for regression after patches and other modifications. We would want to mine the defect tracking database and the support database to try to learn where regression bugs have occurred. New testing would be predicated on those findings, especially if we found hot spots where failures occurred with regularity.

Many of our metrics would have to come from analyzing other metrics. How hard was it to figure out what was wrong? (Analyzability) When changes are needed, how much effort and time does it take to make them? (Change-

ability) How many regression bugs are found (in test and in the field) after changes are made? (Stability) And, how much effort has it taken for testers to be able to test the system? (Testability)

4.6 Portability Testing

Learning objectives

Only common learning objectives.

Portability refers to the ability of the application to install to, use in, and perhaps move to various environments. Of course, the first two are important for all systems. In the case of PC software, given the rapid pace of changes in operating systems, cohabiting and interoperating applications, hardware, bandwidth availability, and the like, being able to move and adapt to new environments is critical too.

Back when the computer field was just starting out, there was very little idea of portability. A computer program started out as a set of patch cords connecting up logic gates made out of vacuum tubes. Later on, assembly language evolved to facilitate easier programming. But still no portability—the assembler was based on the specific CPU and architecture that the computer used. The push to engineer higher-level languages was driven by the need for programs to be portable between systems and processors.

A number of classes of defects can cause portability problems, but certainly environment dependencies, resource hogging, and nonstandard operating system interactions are high on the list. For example, changing shared Registry keys during installation or removing shared files during uninstallation are classic portability problems on the Windows platform.

Fortunately, portability defects are amenable to straightforward test design techniques like pairwise testing, classification trees,³⁰ equivalence partitioning, decision tables, and state-based testing. Portability issues often require a large number of configurations for testing.

Some software is not designed to be portable, nor should it be. If an organization designs an embedded system that runs in real time, we would expect that portability is the least of its worries. Indeed, in a review, we would question any compromises that were made to try to make the system portable if the organi-

30. Pairwise testing and classification trees are discussed in *Advanced Software Testing, Vol. 1*.

ISTQB Glossary

adaptability: The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered.

portability testing: The process of testing to determine the portability of a software product.

coexistence: The capability of the software product to coexist with other independent software in a common environment sharing common resources.

installability: The capability of the software product to be installed in a specified environment.

replaceability: The capability of the software product to be used in place of another specified software product for the same purpose in the same environment.

zation had the possibility of marginalizing the operation of the system. However, there may come a day when the system must be moved to a different chip, a different system. At that point, it might be good if the system had some portability features built into it.

More than the other quality characteristics we have discussed in this chapter, portability requires compromises. A technical test analyst should understand the need for compromise but still make sure the system, as designed and delivered into test, is still suitable for the tasks it will be called to do.

The best way to discuss portability is to look at each of its subcharacteristics. This is a case of the total being a sum of its parts. Very little is published about portability without specifying these subcharacteristics: adaptability, replaceability, installability, coexistence, and compliance.

4.6.1 Adaptability

Adaptability is defined as the capability of the system to be adapted for different specified environments without applying actions other than those provided for that purpose.

In general, the more tightly a system is designed to fit a particular environment, the more suitable it will be for that environment and the less adaptable to other environments. Adaptability, for its own sake, is not all that desirable, frankly. On the other hand, adaptability for solid business or technical reasons

is a very good idea. It is essential to understand the business (or technical) case in determining which trade-offs are advantageous.

When Jamie was a child, his mother read about a mysterious piece of clothing called a “Hawaiian muumuu.” They lived in a small town in the early 1960s; she was excited to be able to order such an exotic item. When she ordered from the catalog, it said, “one size fits all.” Jamie learned from that muumuu that, while one size fits all, it also fits nothing. The thing his mother was sent was enormous—they kidded about using it for a tent.

So what is the point? If we try to write software that will run on every platform everywhere, it likely will not fit any environment well. There are programming languages—such as Java—that are supposed to be able to “write once, run anywhere.” The ultimate portability! However, Java runs everywhere by having its own runtime virtual machine for each different platform. The Java byte code is portable, but only at a huge cost of engineering each virtual machine for each specific platform.

You don’t get something for nothing. Adaptability comes at a price: more design work, more complexity, more code bloat, and those tend to come with more defects. So, when your organization is looking into designing adaptability, make sure you know what the targeted environments are and what the business case is.

When testing adaptability, we must check that an application can function correctly in all intended target environments. Confusingly, this is also commonly referred to as compatibility testing. As you might imagine, when there are lots of options, specifying adaptability or compatibility tests involves pairwise testing, classification trees, and equivalence partitioning.

Since you likely will need to install the application into the environment, adaptability and installation might both be tested at the same time. Functional tests should then be run in those environments. Sometimes, a small sample of functions is sufficient to reveal any problems. More likely, many tests will be needed to get a reasonable picture. Unfortunately, many times a small amount of testing is all that organizations can afford to invest. Given the potentially enormous size of this task, our adaptability testing is often insufficient. As always in testing, the decision of how much to test, how deeply to dig in, will depend on risk and available resources.

There might also be procedural elements of adaptability that need testing. Perhaps data migration is required to move from one environment to another. In that case, we might have to test those procedures as well as the adaptability of the software.

4.6.1.1 Internal Adaptability Metrics

Adaptability metrics help predict the impact on the effort to adapt the system to a different environment.

Adaptability of data structures: A measure of how adaptable the product is to data structure changes. This metric is calculated using the formula

$$X = A / B$$

where A is the number of data structures that are correctly operable after adaptation, as confirmed in review, and B is the total number of data structures needing adaptation. The closer to one, the better.

Hardware environmental adaptability: A measure of how adaptable the software is to the hardware-related environmental change. This metric is specifically concerned with hardware devices and network facilities. The formula

$$X = A / B$$

is used, where A is the number of implemented functions that are capable of achieving required results in specified multiple hardware environments as required, confirmed in review, and B is the total number of functions that are required to have hardware adaption capability. The closer to one, the better.

Organizational environment adaptability: A measure of how adaptable the software is to organizational infrastructure change. This metric is calculated by the formula

$$X = A / B$$

where A is the number of implemented functions that are capable of achieving required results in multiple specified organizational and business environments, as confirmed in review, and B is the total number of functions requiring such adaptability. The closer to one, the better.

System software environmental adaptability: A measure of how adaptable the software product is to environmental changes related to system software. The standard specifically lists adaptability to operating system, network software, and co-operated application software as being measured. This metric uses the formula

$$X = A / B$$

where A is the number of implemented functions that are capable of achieving required results in specified multiple system software environments, as con-

firmed in review, and B is the total number of functions requiring such capability. Closer to one, is better.

Porting user friendliness: A measure of how effortless the porting operations on the project are estimated to be. This metric uses the same formula:

$$X = A / B$$

A is the number of functions being ported that are judged to be easy, as based in review, and B is the total number of functions that are required to be easy to adapt.

4.6.1.2 External Adaptability Metrics

Adaptability metrics measure the behavior of the system or user who is trying to adapt the software to different environments.

Adaptability of data structures: A measure of how easily a user or maintainer can adapt software to data in a new environment. This metric is calculated using the formula

$$X = A / B$$

where A is the number of data items that are not usable in the new environment because of adaptation limitations, and B is the number of data items that were expected to be operable in the new environment. The larger the number (i.e., the closer to one) the better. The data items here include such entities as data files, data tuples,³¹ data structures, databases, and so on. When calculating this metric, the same type of data should be used for both A and B.

Hardware environmental adaptability: A measure of how easily the user or maintainer can adapt the software to the environment. This metric is calculated using the formula

$$X = 1 - A / B$$

where A is the number of tasks that were not completed or did not work to adequate levels during operational testing with the new environment hardware and B is the total number of functions that were tested. The larger (closer to one) the better. ISO 9126-2 specifies this metric is to be used in reference to adapt-

31. The best definition we can find for this term is a multidimensional data set. ISO 9126-2 does not give a formal definition.

ability to hardware devices and network facilities. That separates it from the next metric.

Organizational environment adaptability: A measure of how easily the user or maintainer can adapt software to the environment, specifically the adaptability to the infrastructure of the organization. This metric is calculated much like the previous one using the formula

$$X = 1 - A / B$$

where A is the number of tasks that could not be completed or did not meet adequate levels during operational testing in the user's business environment and B is the total number of functions that were tested. This particular metric is concerned with the environment of the business operation of the user's organization. This separates it from the next similar measure. Larger is better.

System software environmental adaptability: This is also a measure of how easily the user or maintainer can adapt software to the environment, specifically adaptability to the operating system, network software, and co-operated application software. This metric is also calculated using the same formula:

$$X = 1 - A / B$$

A is the number of tasks that were not completed or did not work to adequate levels during operational testing with operating system software or concurrently running application software, and B is the total number of functions that were tested. Again, larger is better.

Porting user friendliness: This final adaptability metric also is a measure of how easily a user can adapt software to the environment. In this case, it is calculated by adding up all of the time that is spent by the user to complete adaptation of the software to the user's environment when the user attempts to install or change the setup.

4.6.2 Replaceability

Replaceability testing is the capability for the system/component to be used in place of another specified software product for the same purpose in the same environment. We are concerned with checking into whether software components within a system can be exchanged for others.

The Microsoft style of system architecture has been a primary driver of the concept of software components, although Microsoft did not invent the idea.

Remote procedure calls (RPCs) have been around a long time, allowing some of a system's processing to be done on an external CPU rather than having all processing performed on one. In Windows, the basic design was for much of the application functionality to be placed outside the EXE file and into replaceable components called dynamic link libraries (DLLs). Early Windows functionality was mainly stored in three large DLLs. For testers, the idea of split functionality has created a number of problems; any tester who has sat for hours trying to emerge from DLL hell where incompatible versions of the same file cause cryptic failures can testify to that.

However, over the years, things have gotten better. From the Component Object Model (COM) to the Distributed Component Object Model (DCOM), all the way to service-oriented architecture (SOA), the idea of having tasks removed from the central executable has become more and more popular. Few organizations would now consider building a single, monolithic executable file containing all functionality. Many complex systems now consist of commercial off-the-shelf (COTS) components wrapped together with some connecting code. HELLOCARMS is a perfect example of that.

The design of the Microsoft Office suite is a pretty good example of replaceable/reusable components—even if it often does not seem that way. Much of the Office functionality is stored in COM objects; these may be updated individually without replacing the entire EXE. This architecture allows Office components to share, upgrade, and extend functionality on the fly. It also facilitates the use of macros and automation of tasks between the applications.

Many applications now come with the ability to use different major database management system (DBMS) packages. Moving forward, many in the industry expect this trend to only accelerate.

Testers must consider this whole range of replaceable components when they consider how they are going to test. The best way to consider distributed component architecture, from RPCs to COTS packages, is to think of loosely coupled functionality where good interface design is paramount. Essentially, we need to consider the interface to understand what to test. Much of this testing, therefore, is integration-type testing.

We should start with static testing of the interface. How will we call distributed functionality, how will the modules communicate? In integration test, we want to test all of the different components that we expect may be used. In system test, we certainly should consider the different configurations that we expect to see in production.

Low coupling is the key to successful replaceability. When designing a system, if the intent of the design is to allow multiple components to be used, then coupling too tightly to any one interface will cause irreplaceability. At this point the system is dependent on those external modules—that are likely not controlled by our organization.

This is an issue that must be considered by management when moving along a path of component-based systems. When everything was in one executable, we could responsibly test all of that functionality. With the growth of decentralization through replaceability of components, the question of who is responsible for testing what becomes paramount. That is a discussion we leave to the Advanced Test Management book, *Advanced Software Testing, Vol. 2*.

4.6.2.1 Internal Replaceability Metrics

Replaceability metrics help predict the impact the software may have on the effort of a user who is trying to use the software in place of other specified software in a specific environment and context of use.

Continued use of data: A measure of the amount of original data that is expected to remain unchanged after replacement of the software. This is calculated using the formula

$$X = A / B$$

where A is the number of data items that are expected to be usable after replacement, as confirmed in review, and B is the number of old data items that are required to be usable after replacement. The closer to one, the better.

Function inclusiveness: A measure of the number of functions expected to remain unchanged after replacement. This measurement is calculated using the formula

$$X = A / B$$

where A is the number of functions in the new software that produce similar results as the same functions in the old software (as confirmed in review) and B is the number of old functions. The closer to one the better.

4.6.2.2 External Replaceability Metrics

Replaceability metrics measure the behavior of the system or user who is trying to use the software in place of other specified software.

Continued use of data: A measure of how easily a user can continue to use the same data after replacing the software. Essentially, this metric measures the success of the software migration. The formula

$$X = A / B$$

is used, where A is the number of data items that are able to be used continually after software replacement and B is the number of data items that were expected to be used continuously. The closer to one, the better. This metric can be used both for a new version of the software and for a completely new software package.

Function inclusiveness: A measure of how easily the user can continue to use similar functions after replacing a software system. This metric is calculated the same way, using the formula

$$X = A / B$$

where A is the number of functions that produce similar results in the new software where changes have not been required and B is the number of similar functions provided in the new software as compared to the old. The closer this value is to one, the better.

User support functional consistency: A measure of how consistent the new components are to the existing user interface. This is measured by using the formula

$$X = 1 - A / B$$

where A is the number of functions found by the user to be unacceptably inconsistent to that user's expectation and B is the number of new functions. Larger is better, meaning that few new functions are seen as inconsistent.

4.6.3 Installability

Installability is the capability of a system to be installed into a specific environment. Testers have to consider uninstallability at the same time.

There is good news and bad news about installability testing. Conceptually it is straightforward. That is the good news. We must install the software, using its standard installation, update, and patch facilities, onto its target environment or environments. How hard can that be? Well, that is the bad news. There are an almost infinite number of possible issues that may arise during that testing.

Here are just some of the risks that must be considered:

- We install a system and the success of the install is dependent on all of the other software that the new system depends on working correctly. Are all co-installed systems working correctly? Are they all the right versions? Does the install procedure even check?
- We find that the typical people involved in doing the installation can't figure out how to do it properly, so they are confused, frustrated, and making lots of mistakes (resulting in systems left in an undefined, crashed, or even completely corrupted state). This type of problem should be revealed during a usability test of the installation documentation. You are testing the usability of the install, right?
- We can't install the software according to the instructions in an installation or user's manual or via an installation wizard. This sounds straightforward, but notice that it requires testing in enough different environments that you can have confidence that it will work in most if not all final environments as well as looking at various installation options like partial, typical, or full installation. Remember—the install is a software system unto itself and must be tested.
- We observe failures during installation (e.g., failure to load particular DLLs) that are not cleaned up correctly, so they leave the system in a corrupted state. It's the variations in possibilities that make this a challenge.
- We find that we can't partially install, can't abort the install, or can't uninstall.
- We find that the installation process or wizard will not shield us from—or perhaps won't even detect—invalid hardware, software, operating systems, or configurations. This is likely to be connected to failures during or after the installation that leave the system in an undefined, crashed, or even completely corrupted state.
- We find that trying to uninstall and clean up the machine destroys the system software load.
- We find that the installation takes an unbearable amount of time to complete, or perhaps never completes.
- We can't downgrade or uninstall after a successful or unsuccessful installation.
- We find that some of the error messages are neither meaningful nor informational.
- Upon un-installation, too few—or too many—modules are removed

By the way, for each of the types of risks we just mentioned, we have to consider not only installation problems, but also similar problems with updates and patches.

Not only do these tests involve monitoring the install, update, or patch process, but they also require some amount of functionality testing afterward to detect any problems that might have been silently introduced. Because, at the end of the day, the most important question to ask is, When we are all done installing, will the system work correctly?

And, just because it was not already interesting enough, we have to think about security. During the install, we need to have a high level of access to be able to perform all of the tasks. Are we opening up a security hole for someone to jump into?

How do we know that the install worked? Does all the functionality work? Do all the interoperate systems work correctly?

At the beginning of discussing installability, we said it was a good news, bad news scenario, the good news being that the install was conceptually straightforward. We lied. It's not. The best way we know to deal with install testing is to make sure that it is treated as a completely different component to test. Some organizations have a separate install test team; that actually makes a lot of sense to us.

And one final note. As an automator, Jamie once thought it would be great to take all of the stuff we just talked about and automate the entire process—test it all by pushing a button. Unfortunately, we've never personally seen that done.

At a recent conference, Jamie talked to an automator who claimed that her group had successfully automated all of their installation tasks. However, she was woefully short on details of how, so we don't know what to believe about her story.

With all of the problems possible in trying to test install and uninstall, with all of the different ways it can fail, it takes a human brain to deal with it all. Until our tools and methodologies get a whole lot better, we think we will continue to be doing most of this testing manually.

During Jamie's first opportunity at being lead tester on a project, he decided to facilitate better communication between the test team and the support team by setting up a brown-bag lunch with both teams. They were testing a very complex system that included an AS/400 host module, a custom ODBC driver, and a full Windows application. We were responsible for testing everything that we sold.

During the lunch, Jamie asked the support team to list the top 10 customer complaints. It turned out that 7 of the top 10 complaints were install related. Oops! We weren't even testing the install because Jamie figured it was not a big deal. He had come from an organization where they tested an operating system—there the install was tested by another group in another state.

Very often, install complaints rank very high in all problems reported to support.

4.6.3.1 Internal Installability Metrics

Installability metrics help predict the impact on a user trying to install the software into a specified environment.

Ease of set-up retry: A measure of how easy it is expected to repeat the setup operation. This is calculated using the formula

$$X = A / B$$

where A is the number of implemented retry operations for set-up, confirmed in review, and B is the total number of set-up operations required. The closer to one, the better.

Installation effort: A measure of the level of effort that will be required for installation of the system. This metric is calculated using the formula

$$X = A / B$$

where A is the number of automated installation steps, as confirmed in review, and B is the total number of installation steps required. Closer to one (i.e., fully automated) is better.

Installation flexibility: A measure of how customizable the installation capability is estimated to be. This is calculated using the formula

$$X = A / B$$

where A is the number of implemented customizable installation operations, as confirmed in review, and B is the total number required. The closer to one, the more flexible the installation.

4.6.3.2 External Installability Metrics

Installability metrics measure the impact on the user who is trying to install the software into a specified environment.

Ease of installation: A measure of how easily a user can install software to the operational environment. This is calculated by the formula

$$X = A / B$$

where A is the number of cases in which a user succeeded in changing the install operation for their own convenience and B is the total number of cases in which a user tried to change the install procedure. The closer to one, the better.

Ease of set-up retry: A measure of how easily a user can retry the set-up installation of the software. The standard does not address exactly why the retry might be needed, just that the retry is attempted. The metric is calculated using the formula

$$X = 1 - A / B$$

where A is the number of cases where the user fails in retrying the set-up and B is the total number of times it's attempted. The closer to one, the better.

4.6.4 Coexistence

The fourth subcharacteristic we need to discuss for the portability characteristic is called *coexistence* testing—also called sociability or compatibility testing. This is defined as the capability to coexist with other independent software in a common environment sharing common resources. With this type of testing, we check that one or more systems that work in the same environment do so without conflict. Notice that this is not the same as interoperability testing because the systems might not be directly interacting. Earlier, we referred to these as “cohabiting” systems, though that phrase is a bit misleading since human cohabitation usually involves a fair amount of direct interaction.

It's easy to forget coexistence testing and test applications by themselves. This problem is often found in groups that are organized into silos and where application development takes place separately in different groups. Once everything is installed into the data center, though, you are then doing de facto compatibility testing in production, which is not really a good idea. There are times when we might need to share testing with other project teams to try to avoid coexistence problems.

With coexistence testing, we are looking for problems like the following:

- Applications have an adverse impact on each other's functionality when loaded on the same environment, either directly (by crashing each other) or indirectly (by consuming all the resources). Resource contention is a common point of failure.
- Applications work fine at first but then are damaged by patches and upgrades to other applications because of undefined dependencies.
- DLL hell. Shared resources are not compatible and the last one installed will work, breaking the others.

Assume that we just installed this system. How do we know what other applications are on that system, much less which ones are going to fail to play nice? This is yet another install issue that must be considered. In systems where there is no shared functionality (i.e., one without DLLs), this is less important.

One solution that is becoming more common is the concept of virtual machines. We can control everything in the virtual machine, so we can avoid direct resource contention between processes.

4.6.4.1 Internal Coexistence Metrics

Coexistence metrics help predict the impact the software may have on other software products sharing the same operational hardware resources.

Available coexistence: A measure of how flexible the system is expected to be in sharing its environment with other products without adverse impact on them. The formula

$$X = A / B$$

is used where A is the number of entities with which the product is expected to coexist and B is the total number of entities in the production environment that require such coexistence. Closer to one is better.

Replaceability metrics help predict the impact the software may have on the effort of a user who is trying to use the software in place of other specified software in a specific environment and context of use.

Continued use of data: A measure of the amount of original data that is expected to remain unchanged after replacement of the software. This is calculated using the formula

$$X = A / B$$

where A is the number of data items that are expected to be usable after replacement as confirmed in review and B is the number of old data items that are required to be usable after replacement. The closer to one, the better.

Function inclusiveness: A measure of the number of functions expected to remain unchanged after replacement. This measurement is calculated using the formula

$$X = A / B$$

where A is the number of functions in the new software that produce similar results as the same functions in the old software (as confirmed in review) and B is the number of old functions. The closer to one, the better.

4.6.4.2 External Coexistence Metrics

Coexistence metrics measure the behavior of the system or user who is trying to use the software with other independent software in a common environment sharing common resources.

Available coexistence: A measure of how often a user encounters constraints or unexpected failures when operating concurrently with other software. This is calculated using the formula

$$X = A / T$$

where A is the number of constraints or failures that occur when operating concurrently with other software and T is the time duration of operation. The closer to zero, the better.

4.6.5 Compliance

4.6.5.1 Internal Compliance Metrics

Portability compliance metrics help assess the capability of the software to comply with standards, conventions, and regulations that may apply. It is measured using the formula

$$X = A / B$$

where A is the number of correctly implemented items related to portability compliance as confirmed in review and B is the total number of compliance items.

4.6.5.2 External Compliance Metrics

Portability compliance metrics measure the number of functions that fail to comply with required conventions, standards, or regulations. This metric uses the formula

$$X = 1 - A / B$$

where A is the number of portability compliance items that have not been implemented and B is the total number of portability compliance items that are specified. The closer to one, the better. ISO 9126-2 notes that this is a metric that works best when seen as a trend, with increasing compliance coming as the system becomes more mature.

4.6.6 Exercise: Portability Testing

Using the HELLOCARMS system requirements document, analyze the risks and create an informal test design for portability using one requirement.

4.6.7 Exercise: Portability Testing Debrief

Portability testing consists of adaptability, installability, coexistence, replaceability, and compliance subattributes. Because HELLOCARMS is surfaced on browsers, we find the compelling subattribute to be adaptability. Therefore, we have selected requirement 060-010-030 for discussion. It reads: “HELLOCARMS shall be configured to work with all popular browsers that represent five percent (5%) or more of the currently deployed browsers in any countries where Globobank does business.”

Our first effort would be to try to get a small change to this requirement during the requirements review period. The way it is written, it appears that, by

release three, we need to be concerned about all versions of browsers rather than just the latest two versions as expressed in requirements 060-010-010 and 060-010-020. We hope this is an oversight and will move forward in our design assuming that we only need the latest two versions.

This particular requirement is not enforced until release three. However, we would start informally testing it with the first release. This is because we would not want the developers to have to remove technologies they used after the first two releases simply because they are not compatible with a seldom-used browser that still meets the five percent threshold. We would make sure that we stressed this upcoming requirement at low-level design and code review meetings.

We would have to survey what browsers are available. This entails discovering what countries Globobank is active in and performing Web research on them. We would hope to get our marketing group interested in helping out to prevent spending too much time on the research ourselves.

We would create a matrix of all the possible browsers that meet the criteria, including the current version and one previous version for each. We would also build into that matrix popular operating systems and connection speeds (dial-up and two speeds of wideband).

This matrix is likely to be fairly large. We do not cover pairwise techniques in this book.³² However, if we did not know how to deal with this powerful configuration testing technique, we would enlist a test analyst to help us out. We would spread out our various planned tests over the matrix to get acceptable coverage, focusing most tests on those browsers/operating systems/speeds that represented most of our prospective users.

After release, we would make sure to monitor reported production failures through support, ensuring that we were tracking environment-related failures. We would use that information to tweak our testing as we move into the maintenance cycle.

32. Interested readers can find this technique in *Advanced Software Testing, Volume 1* or at the website, pairwise.org.

4.7 General Planning Issues

Learning objectives

Only common learning objectives.

Let's close this chapter with some general planning issues related to nonfunctional testing. It's often the case that nonfunctional testing is overlooked or underestimated in test plans and project plans. As we've seen throughout this chapter, though, nonfunctional facets of system behavior can be very important, even critical to the quality of the system. Therefore, such omissions can create serious risks for the product, and thus the project. As was mentioned earlier, the failure to address nonfunctional risks—specifically those related to performance—led to a disastrous project release in the case of the United States health-care.gov website.

While not wanting to comment on the specifics of that example, one common reason for such omissions is that many test managers lack adequate technical knowledge to understand the risks. As technical test analysts, we can contribute to the quality risk analysis and test planning processes to help ensure that such risks are not overlooked or minimized. However, technical test analysts do not always have the management perspective needed to see these gaps in advance. This section is designed to help you anticipate the gaps and help the test manager address them.

The first significant gap that can exist is a lack of understanding of stakeholder needs in terms of nonfunctional requirements. Business stakeholders especially are often highly attuned to functional issues and can explain what accurate, suitable behavior looks like; the systems with which the system under test must interoperate; and the regulations with which the system must comply. However, when asked about other behaviors, they'll say, "Yes, of course the system must be secure, easy to use, reliable, quick, useful in any supported configuration, and ready for updates with new features and bug fixes at a moment's notice." Those are hardly useful requirements for security, usability, reliability, efficiency, portability, and maintainability. The opacity of stakeholder requirements is yet another hurdle to the test manager's ability to plan for nonfunctional testing.

This is where you, the intrepid technical test analyst, come in. You should—especially after the material discussed in this chapter—be ready to have a more detailed discussion with stakeholders about what exactly they want in terms of

nonfunctional behavior. Of course, you need to make sure that these discussions include all relevant stakeholders. These stakeholders include business stakeholder such as customers and users. They also include operations and maintenance staff. They include developers, architects, designers, and database administrators. As discussed in Chapter 1, including the broadest cross-section of stakeholders gives the most complete view of the issues. This applies not only for risks but also for planning considerations and requirements.

In some cases, the response to your queries about nonfunctional requirements, especially performance, reliability, security, and usability, can be, “It should work at least as well as the existing system,” or, “It should be as good as our competitors’ systems.” While this can be frustrating at first, notice that such a response actually provides you with a real, live, usable test oracle—which is what you were looking for anyway. If a stakeholder has given such a response, don’t be frustrated by it—thank them for giving you a realistic, existing system to measure your systems against.

Another significant challenge to nonfunctional testing is that tools are often required to carry out such tests. Reliability and performance tests are usually impossible to perform without tools. Some security testing can be done manually, but most hackers use tools as well as their twisted wiles, so security testers must use these tools as well. Open-source and other free tools are available, but many of our clients opt to use commercial tools for reasons discussed in Chapter 6. So, the time and money required to acquire, implement, and deploy the tools should be included in planning; even if you do use tools that are open-source or free, the cost of deploying the tools can be significant.

In addition, these tools are not easy to use. If you have no experience with a particular tool—or, worse yet, a whole type of tool—simply trying to use them yourself, no matter how technical you are, might result in false positives, false negatives, and a lot of unproductive thrashing around. So, the plan, as well as the budget and estimate, must include the costs of training people, learning how the tool works in your environment, and possibly even hiring consultants to help you use the tool, at least initially.

In some cases, a tool might be needed, but no commercial, open-source, or free tool exists. In this case, you might need to build your own tool. The implications of this approach are discussed to some extent in Chapter 6, and more extensively in the ISTQB Advanced Test Manager syllabus.³³ Tool development is typically a significant development effort, though in some cases open-source components can be used to build the tool at a very reasonable cost. Any tool you build—just like any software your organization builds—should be assumed

buggy, and so testing is required. The tool is an important asset, and so documentation is necessary. In order to use the tool in the future, maintenance will be needed. If the tool is used for safety-critical applications such as medical devices or avionics, certification of the tool according to regulatory guidelines is also required. All of these activities take time and cost money and thus must be planned.

And that's not all. In many cases, in order to yield meaningful results, non-functional tests must be run in production-like environments. For example, performance and reliability tests are very sensitive to resource constraints, so running them in scaled-down environments can be misleading. Rex has seen security tests in production environments yield very different results than those same tests run in testing environments, just because the production environments were configured differently.

Replicating a production environment can be very easy and cheap, very hard and expensive, or somewhere in between. At the easy end, thanks to virtualization and cloud computing, whistling up an exact replica of an environment that already exists in a cloud computing setting can be quite simple, though the cost might still be high if the capacity of the environment is large. If you have a large, complex, unique, and expensive production environment—for example, a supercomputer or mainframe networked to an unusual configuration of servers and clients—then the cost, difficulty, and time lag associated with replicating that environment could be prohibitive in all but the most high-risk situations.

When environments cannot be replicated, most of Rex's clients tend to resort to one of two options. The first, and probably most popular, is to test in a scaled-down environment. This can work for security testing, if great care is taken to preserve exactly—and we mean *exactly*—the configuration of the production environment, if not the scale of its resources. However, for performance and reliability testing, in the absence of exceptionally accurate models of how changes in resource capacity will affect behavior, it's highly likely that such tests, while revealing some important defects, will also miss other important defects.

Another option is to test in the production environment. Here there can be no question of realism in the test environment. However, there is a signifi-

33. A discussion of the management implications of tools as described in the ISTQB Advanced Test Manager syllabus can be found in Rex's book *Advanced Software Testing: Volume 2, Second Edition*.

cant issue of how you insulate the users from the tests. Putting load on the system, as is necessary in performance testing (at least on a transient basis) and reliability testing (for a much more extended period), has the possibility of affecting real users. Security tests could result in the leakage of actual, sensitive customer data, perhaps to the wider world than just the testers. In addition, simply embarrassing implications of testing can occur. Rex is aware of an instance where testing in production resulted in test data being sent to real customers, which might seem trivial except that the test data was a letter whose salutation read “Dear idiot” and the recipients were every customer the company had.

The test plan must carefully consider all these issues and risks. At the least, planning for the environment should be done carefully to avoid any conflict between the functional and nonfunctional tests. Timing will often be an issue, especially if production replica environments or actual production environments are used, but in some cases even when standard test environments can be used.

In addition to test environments, test data is often a challenge for nonfunctional testing. Especially in the case of performance, reliability, and security testing, using realistic data, including configuration data, is critical to obtaining meaningful results and finding important defects. One way to ensure realistic test data is to test using production data. However, production data often contains sensitive information such as personal identifying information, health status, financial information, and the like. In all cases, businesses, governments, nonprofits, and other organizations have an ethical duty to protect the public from harm that arises from the misuse of such data, though recent history shows that organizations are often less than diligent in this regard. When the pangs of conscience are not enough—and lately they don’t seem to be—governments step in, imposing civil and criminal requirements to be good stewards of such data.

So, whether for reasons of good organizational citizenship or due to fear of lawsuits or worse, an increasing number of our clients are adopting data protection policies. These policies restrict who can access production data, which often has the side effect of preventing testers from using the data in its pure form. Fortunately, there are tools and techniques that can be used to anonymize production data as part of the process of transferring the data into the test environment, making it impossible for a malevolent tester (or just a malevolent person with access to the test data) to do harm to individuals or society. Unfortunately, these tools are often expensive. Even when price is not an issue, the

process of anonymizing and transferring often huge volumes of data from the production environment into the test environment can be a small project all by itself. Either way, obtaining nonfunctional test data often requires advanced planning.

In some cases, the issues associated with test data are not the volume or provenance but the nature of the data itself. Testing involves the use of a test oracle, which means a way of determining whether the test has passed or failed. However, application security increasingly involves transmitting data in encrypted forms. Encryption makes it unreadable by those who shouldn't be able to read it, such as, for example, hackers looking to steal credit card information from a customer database. Of course, encryption makes it impossible for us to check that credit card number in the customer database to see whether it has saved properly—at least, encryption that is implemented properly will.

So, some clever solutions must be used to check the data. Partial oracles are one way of doing so. Using service virtualization to replicate a third-party service that has the authorization to receive the unencrypted information is another way of doing so. Asking the developers to install a backdoor that allows you to circumvent the encryption might seem clever at first, but it is in fact very, very stupid in most cases. These backdoors have a way of being discovered, and you don't want to be the person who asked it to be put there when it is.

Speaking of service virtualization, this is just one of the organizational considerations involved in nonfunctional testing. Applications often—in fact, these days, usually—interact with other applications via various services available on the internal company network or the Internet. You'll need to work with the teams that offer these services to identify the interfaces involved, how they work, whether test interfaces are available, and so forth. If you can coordinate a test interface with them, so much the better, provided they agree to support the interface. If you can't get a test interface, or if you simply don't want to rely on such an interface, you might decide to use service virtualization tools to replace the interface.

In some cases, nonfunctional tests must extend end to end, across all the applications, in which case you need people in other groups involved. Or perhaps you don't have all the skills required to run a certain set of tests, such as security tests. If, for whatever reason, the skills, expertise, and involvement of other groups is required to successfully carry out nonfunctional tests, this involvement will need to be planned and coordinated. You'll need to help the test manager identify these groups and the specific involvement necessary.

4.8 Sample Exam Questions

1. You are new to the organization and have been placed in a technical testing role. You are asked to investigate a number of complaints from customers who fear that they may have security issues with the software. Symptoms include virus warnings from security software, missing and corrupted documents, and the system suddenly slowing down. Which two of the following websites might you access to get help finding causes for these symptoms?
 - A. Wikipedia
 - B. CAPEC
 - C. SUMI
 - D. CVE
 - E. WAMMI
2. Nonfunctional testing has never been done at your organization, but your new director of quality has decided that it will be done in the future. And, she wants metrics to show that the system is getting better. One metric you are calculating is based on a period of testing that occurred last week. You are measuring the time that the system was actually working correctly compared to the time that it was automatically repairing itself after a failure. Which of the following metrics are you actually measuring?
 - A. MTBF
 - B. Mean down time
 - C. Mean recovery time
 - D. Availability

3. You find out that marketing has put a new claim into the literature for the system, saying that the software will work on Windows 95 through Win 7. Which of the following nonfunctional attributes would you most likely be interested in testing and collecting metrics for?
 - A. Adaptability
 - B. Portability
 - C. Coexistence
 - D. Stability
4. You are doing performance testing for the system your company sells. You have been running the system for over a week straight, pumping huge volumes of data through it. What kind of testing are you most likely performing?
 - A. Stress testing
 - B. Soak testing
 - C. Resource utilization testing
 - D. Spike testing
5. Rather than developing all of your own software from the ground up, your management team has decided to use available COTS packages in addition to new code for an upcoming project. You have been given the task of testing the entire system with a view to making sure your organization retains its independence from the COTS suppliers. Which of the following non-functional attributes would you most likely investigate?
 - A. Replaceability
 - B. Portability compliance
 - C. Coexistence
 - D. Adaptability

5 Reviews

Anything becomes interesting if you look at it long enough.

—Gustave Flaubert

The fifth chapter of the Advanced syllabus is concerned with reviews. As you will recall from 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, design specification, database schema, or code. The primary goal is typically to find defects in that work product before it serves as a basis for further project activity, though other goals can also apply.

In the Advanced Technical Test Analyst syllabus, we focus on reviews of technical work products. Chapter 5 of the Advanced Technical Test Analyst syllabus has two sections.

1. Introduction
2. Using Checklists in Reviews

Let's look at each section.

5.1 Introduction

Learning objectives

TTA 5.1.1 (K2) Explain why review preparation is important for the technical test analyst.

Again, think back to the beginning of Chapter 2. We introduced a taxonomy for tests, shown in Figure 2–1. You should remember, from the Foundation syllabus, the distinction between static and dynamic tests. Static tests are those tests that

do not involve execution of the test object. Dynamic tests do involve execution of the test object. In Chapters 2 and 4, we talked about test techniques and quality characteristics, mostly from the point of view of dynamic testing.

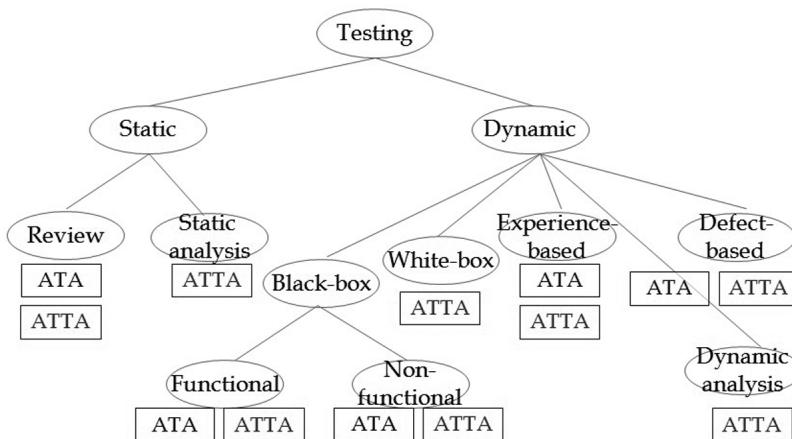


Figure 5–1 Advanced syllabus testing techniques

In this chapter, we cover static testing, focusing on one branch of the static test tree, that of reviews. (We covered static analysis earlier, in Chapter 3.) The material in this chapter builds on what was covered on reviews in the Foundation syllabus.

To have success with reviews, an organization must invest in and ensure good planning, preparation, participation, and follow-up. It's 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 (referred to as professional pessimism in the Foundation syllabus). That outlook makes them useful in a review, though they have to remain aware of the need to contribute in a positive way.

This is true not only of testers but of everyone involved. Being a group activity, all review participants must commit to well-conducted reviews. One negative, confrontational, or belligerent participant can damage the entire process profoundly.

One important part of ensuring well-conducted reviews, with proper attitudes and behaviors, is ensuring that participants have been trained. Common courtesy alone, while helpful, is not enough; in fact, excessive courtesy can stymie reviews. Each reviewer must know the right ways to discover *and* commu-

nicate defects, questions, and comments. People are not born knowing how to participate effectively in reviews. How to use a checklist of the kinds of defects to look for, and in which documents, for example, is not obvious (nor are the checklists themselves obvious), but people can learn to do so.¹ How to participate effectively in brainstorming solutions in a technical review is also not obvious and also something people can learn. How to rephrase a potentially harsh or confrontational—but accurate—observation about a defect into a constructive, collaborative—and equally accurate—observation is something people can learn to do. Reviews, like any other form of testing, are not like breathing: You weren't born knowing how to review and test software, and for some people it comes a lot less naturally than for others.

As a technical test analyst, in addition to professional pessimism, you should bring your technical expertise to bear on the question of, What is likely to happen in operational or production environments if this design, this architecture, or this code is put into use? That's a different question than one of elegance or even correctness, because it focuses on suitability of the behavior. Sometimes, you might find that checklists for work products on design, architecture, and code don't always consider this perspective, so be ready to help define, apply, and maintain checklists in order to reflect such a perspective. (We'll discuss checklists further in the next section.) In addition, when reviews reveal defects, you should help participants understand why these defects may have a higher impact on users, customers, and business stakeholders than might be immediately obvious. To some extent, technical test analysts can bridge the gap between technology and the problem that technology solves during a review (and during dynamic testing).

Another important question for the technical test analyst is, Is there enough information in this work product to allow me to design, implement, and execute a test and, of course, to evaluate whether that test has passed or failed? Testability is a critical concern, but one that only testers tend to have at the top of their priority list. A testable requirement is one where you know what preconditions, steps, inputs, and data are necessary to evaluate the requirement, and the effort associated with doing so is not excessive.² A testable requirement is also one

1. Examples of such checklists, and ideas on how to use them effectively, can be found in Karl Wiegers's books *Peer Reviews in Software: A Practical Guide* and *Software Requirements, Third Edition*.

2. One reviewer, Bernard Homés, said that he did not consider excessive effort a true testability defect. Whether excessive test effort is a testability defect or just a concern, it should be noted in the review.

where you could recognize the expected results and post-conditions associated with the test. Any requirement that doesn't meet these two criteria should be flagged as a potential problem during a review.

While training is important, training participants by itself, however, is not enough. People might know the right thing to do. However, as discussed in Chapter 1, section 5 of the Foundation syllabus, people tend to align their actions and priorities with stated objectives and, especially, financial incentives and disincentives. If, for example, authors are measured and rewarded based on reducing the number of defects found in their work during reviews, static analysis, and dynamic testing, their incentive in a review (and during dynamic testing) will be to deny that any defect exists.

Rex has seen this kind of behavior firsthand, including aggressive and domineering verbalizations and physical postures toward testers. Not only must a rule be in place that reviews are positive experiences for authors—which is essential—there should be a similar rule that no attempts be made to invalidate a reviewer's feedback. However, such rules are for naught if management punishes the discovery of defects in an author's work or rewards the discovery of defects by reviewers.

Woody Allen, the New York film director, is reported to have once said, "Eighty percent of success is showing up." That might be true in the film business, but just showing up would not make someone 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, to us, showing up for a review meeting unprepared qualifies as rude behavior, disrespectful of the time of the others in the room.

What constitutes adequate preparation? Of course, you have to read the work product being reviewed completely and carefully and reflect on your understanding of it. Plan on this process taking longer than reading, say, a blog post or a newspaper article. You'll need to take notes as you go about questions you have, problems you have found, and potential problems you need to check

further prior to the review. You'll need to check internal and external cross-references, because contradictions are frequent.

Even more challenging, you have to ask yourself, "What's not here that should be here?" Omissions can be one of the hardest types of problems to find. By cross-referencing design elements, architecture, and code against higher-level specifications, you can find some of these problems. However, what does the user or customer want or need? Is it here? Will it work the way the user or customer expects?

Rex has seen reviews performed with inadequate or completely absent preparation time. People show up, listen politely, and make perfunctory comments. They point out minor spelling, grammar, and style problems and then agree to the result. That's not a review; that's an event that builds false confidence while making everyone happy about the harmonious process. The downstream results—finding many bugs that could have been detected and removed in a review—put the lie to this kind of success.

Why don't people prepare adequately? In part, the answer is that they do not know how to prepare. Training can address this issue. However, a major problem is the lack of adequate time. Generally, someone should have a chance to spend at least one hour (to get into the flow) and at most two hours (to avoid attention span limits) in review preparation. This is a significant block of time. If someone is attending two or three reviews in a week, the preparation time according to this rule, plus the review time (which is similarly set at 1 to 2 hours), they could spend 12 hours (close to a third of the week) to carry out the reviews properly. A cursory job will take a lot less time, and perhaps people will feel good about it, but the value will not be obtained.

Because reviews are so effective when done properly, organizations should review every important document. That includes test documents. Test plans, test cases, quality risk analyses, bug reports, test status reports, you name it. Our 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 non-trivial 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.

In an informal review, there are no defined rules, no defined roles, and 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 defects, while very formal reviews like inspections can find up to 85 percent of defects.³ 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.

Like any test activity, reviews can have various objectives. One common objective is finding defects. Others, typical of most 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 team and for management. Unique to reviews is the addition of another common objective, that of ensuring uniform understanding of the document—and its implications for the project—and building consensus around the statements in the document. Reviews of test plans, test processes, and tests themselves can also reveal gaps in coverage.

Reviews usually precede dynamic tests. Reviews should complement dynamic tests. Because the cost of a defect increases the longer that defect remains in the system, reviews should happen as soon as possible. However, because not all defects are easy to find in reviews, dynamic tests should still occur.

5.2 Using Checklists in Reviews

Learning objectives

TTA 5.2.1 (K4) Analyze an architectural design and identify problems according to a checklist provided in the syllabus

TTA 5.2.2 (K4) Analyze a section of code or pseudo-code and identify problems according to a checklist provided in the syllabus

We mentioned the role of technical test analysts in defining, applying, and maintaining checklists in the preceding section. Why are checklists important? For one thing, checklists serve to ensure that the same level and type of scrutiny is

3. See Capers Jones's book *Software Assessments, Benchmarks, and Best Practices*.

ISTQB Glossary

anti-pattern: Repeated action, process, structure, or reusable solution that initially appears to be beneficial and is commonly used but is ineffective and/or counterproductive in practice.

brought to each author's work. There can be a tendency of review participants to defer to a senior employee, and thus that person's work, when in fact everyone is fallible and we all make mistakes. Conversely, a less-senior or more-insecure employee might feel threatened by the review. Regardless of the individual author and their skills, there is nothing personal about locating potential problems and improvements for their work from a checklist. The checklists serve as a valuable leveling and depersonalizing tool, both factually and psychologically.

More important, though, checklists serve as a repository of best practices—and worst practices—that can help the participants of a review remember important points during the review. The checklist frees the participants from worrying whether they forgot some critical issue or mistake that was made in the past. Instead, the checklist gives general patterns and anti-patterns to the participants, allowing them to ask instead, "How could this particular item on the checklist be an important consideration for this work product that we're reviewing?"

This idea of including not only best practices but also worst practices is important.⁴ Let's illustrate with an example. In Chapter 4, we introduced the *Common Vulnerabilities and Exposures* site from MITRE, the Federally funded non-profit, reproduced again as Figure 5–2 below. If you are involved in reviewing code where security is important—and security almost always is important now—the items here can be incorporated into your code checklist and/or your

4. The syllabus uses the terms *pattern* and *anti-pattern*, but the general concept is the same as *best practice* and *worst practice*, respectively. In an influential book, *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma coined the term *design patterns*, referring to general, widely used solutions that can be applied to common problems and that have proven their worth in solving these problems. An anti-pattern refers to the opposite situation, where the solution has indeed been widely used but has in fact proven to be dangerous or just suboptimal compared to some other solution. Those contrasts correspond to the usage of the common terms *best practice* and *worst practice*; however, *best practice* and *worst practice* have the advantage of being widely used outside of software engineering and thus are more easily understood by business stakeholders as well as technical stakeholders.

static code analysis tools to ensure that dangerous coding mistakes are not happening.

Rank	Score	ID	Name
[1]	346	CWE-79	Failure to Preserve Web Page Structure ('Cross-site Scripting')
[2]	330	CWE-89	Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection')
[3]	273	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	261	CWE-352	Cross-Site Request Forgery (CSRF)
[5]	219	CWE-285	Improper Access Control (Authorization)
[6]	202	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[7]	197	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[8]	194	CWE-434	Unrestricted Upload of File with Dangerous Type
[9]	188	CWE-78	Improper Sanitization of Special Elements used in an OS Command ('OS Command Injection')
[10]	188	CWE-311	Missing Encryption of Sensitive Data
[11]	176	CWE-798	Use of Hard-coded Credentials
[12]	158	CWE-805	Buffer Access with Incorrect Length Value
[13]	157	CWE-98	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP File Inclusion')
[14]	156	CWE-129	Improper Validation of Array Index
[15]	155	CWE-754	Improper Check for Unusual or Exceptional Conditions
[16]	154	CWE-209	Information Exposure Through an Error Message
[17]	154	CWE-190	Integer Overflow or Wraparound
[18]	153	CWE-131	Incorrect Calculation of Buffer Size
[19]	147	CWE-306	Missing Authentication for Critical Function
[20]	146	CWE-494	Download of Code Without Integrity Check
[21]	145	CWE-732	Incorrect Permission Assignment for Critical Resource
[22]	145	CWE-770	Allocation of Resources Without Limits or Throttling
[23]	142	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[24]	141	CWE-322	Use of a Broken or Risky Cryptographic Algorithm
[25]	138	CWE-362	Race Condition

Figure 5–2 List of top 25 security vulnerabilities from CVE website

What should go into a checklist? We'll examine some specific items from the syllabus and from industry examples in subsections below, but we can make some general statements first. If there are common templates, style guides, variable naming rules, or word-usage standards in the company for certain types of work products, the checklists should reflect those guidelines. (Of course, the verification of some of these guidelines can be implemented in static analysis tools. In that case, the checklist or the review entry criteria can simply mention that successful completion of the static analysis is a pre-requisite to the review.)

Often it is the case that, for particular organizations or for particular systems, there are specific quality characteristics such as security and performance that are important and should be verified. So, checklists can include specific sections for each such characteristic, along with good and bad things to watch for. If these sections are applicable only to certain projects or products, the checklist should give guidance on when the items in a given section are important to consider.

While there are many useful examples of checklists for various kinds of work products, some of which we'll present below, you should plan to maintain and evolve your organization's checklists over time. When building on known best practices and worst practices in checklists you've adopted, keep in mind the following factors:

- Product: Different products have different needs. As an extreme example of these contrasts, a smartphone application that allows users to play an entertaining and engaging game is a very different proposition than a mainframe application that handles thousands or even millions of mission-critical transactions per week.
- People: Your team will have different strengths and weaknesses, which are often amplified by the nature of the product. If your team tends to make the same types of mistakes over and over, these should be part of your checklist, regardless of whether these mistakes are common in software engineering in general. In addition, the software development life cycle matters. To use an example provided by one reviewer, Bernard Homés, in an Agile model, a review should need to address not only new and changed functions in an iteration but also any existing functions from previous iterations that could be impacted.
- Tools: If you are able to use static analysis tools to detect certain kinds of defects prior to the review, then the checklist can simply note whether the tools have been applied.
- Priorities: What matters most to technical stakeholders, business stakeholders, users, customers, and testers should be considered as the checklist evolves.
- History: You should look for anti-patterns in your defects. These should be part of your checklist. You should also look for successful examples—patterns—in your past releases. What does your organization do well? What does your organization do poorly?
- Quality characteristics: You should look for ways in which your organization has successfully delivered various technical and domain quality characteristics. Can you see common elements in those successes? How about in the failures?
- Testability: You should consider factors that make it easier, harder, or plain impossible to evaluate whether a requirement, user story, use case, design element, architectural attribute, or other attribute is or is not satisfied. While testability as a quality characteristic is not often the highest priority, it should be considered as part of any checklist. If you can't determine whether something works or not, that's going to be a problem during dynamic testing. Not only will there be a potentially large number of tests for a requirement with testability issues, it will also be hard to distinguish false positives (and false negatives) from actual defects.

While checklists are very useful, there are risks associated with them. One risk is whether it will focus attention on verification (are we building the product right?) rather than validation (are we building the right product?). Checklists will tend to bring your thinking into a verification mindset. Remember to think outside of the checklist to look at whether the product will actually solve the end users' problems.

With these overall points in mind, let's look at some specific examples of checklists. These examples should give you an idea of items to put in your own organization's checklists.

5.2.1 Some General Checklist Items for Design and Architecture Reviews

Software architecture and *software design* are two terms that can cover a lot of territory. Let's look at a few definitions of these terms before we start to discuss checklist items.

First, let's take the syllabus definition of *software architecture*, which says it "consists of the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution."⁵ This definition can easily call to mind network diagrams of systems or the applications running on these systems, and how they communicate with each other. To be complete, though, you'd also need to see the principles behind the decisions related to the architecture. The architectural principles would (one would hope) be present in explanatory text surrounding the diagrams.

Next, a more expansive, perhaps more pragmatic, and definitely more assertive view of software architecture comes from the Software Engineering Institute:

The software architecture of a program or computing system is a depiction of the system that aids in the understanding of how the system will behave. Software architecture serves as the blueprint for both the system and the project developing it, defining the work assignments that must be carried out by design and implementation teams. The architecture is the primary carrier of system qualities such as performance, modifiability, and security, none of which can be achieved without a unifying architectural vision.

5. This definition derives from IEEE Standard 1471:2000, which has been superseded by IEEE Standard 42010:2011, and from Bass, et al.'s book *Software Architecture in Practice, 2nd Edition*.

Architecture is an artifact for early analysis to make sure that a design approach will yield an acceptable system. By building effective architecture, you can identify design risks and mitigate them early in the development process.

Notice the assertions of the importance of early architecture and the evaluation of architecture documents.⁶

So, what's the difference between software architecture and software design? Well, Rex has heard these terms used interchangeably by various technical and business participants and stakeholders on projects, and a quick search on the Internet will reveal all sorts of opinions. To the extent that a difference is intended, it seems to be the level of focus. In that case, software architecture can refer to higher levels of component relationships, where the components can be databases, application servers, and other such coarse-grained elements. Software design can refer to components at a lower level, such as individual units or groups of units within a single application.

In the Advanced Technical Test Analyst syllabus, a list of general items to consider for an architecture review checklist is provided. This list is excerpted from an online resource, and in fact contains only a portion of the complete set of architectural attributes that the online resource says should be checked. The complete list includes the following attributes:

- Performance: Elements of the architecture that affect response time and throughput.
- Reliability: Elements of the architecture that ensure that the system operates properly over an extended period of time, even when confronted with invalid inputs, unexpected events, and so forth.
- Availability: Related to reliability, these are elements of the architecture that ensure that the system stays up and running, which can include failover and redundancy.
- Efficiency: Related to performance, reliability, and availability, and sometimes in tension with those priorities, these are elements of the architecture that enable it to use system resources in an optimal fashion.
- Scalability: Related to performance, reliability, and efficiency, the elements of the architecture that make it possible to expand the number of users and to expand the hardware resources available to the system.

6. For the source of this definition, see: <http://www.sei.cmu.edu/architecture/>.

- Security: Elements of the architecture that allow the system to reject unauthorized attempts to access data or functionality and to avoid denial of service.
- Modifiability: Elements of the architecture that make it possible to update a system quickly, easily, and safely.
- Extensibility: Related to modifiability, the elements of the architecture that make it easy to add new features or components or remove no longer needed features or components.
- Reusability: Related to modifiability and extensibility, the elements of the architecture that make the system's elements reusable for another system, which might or might not be an important consideration for any given system.
- Maintainability: Related to modifiability, the elements of the architecture that enable quick, safe, and easy repair of defects discovered in production.
- Portability: Elements of the architecture that help the system run in various environments.
- Functionality: The elements of the architecture that allow the system to solve the business problems it is designed to solve in an accurate and suitable fashion.
- Conceptual Integrity: The overall feel of the architecture, in terms of consistency across all the elements and attributes.
- Interoperability: The elements of the architecture that relate to the system's ability to communicate internally and externally, which often will have intersections with other attributes of the system architecture, such as security and performance.
- Usability: The elements of the architecture that make the system easy to learn, operate, and understand and that make it attractive to the user, which in some cases must be balanced against other architectural priorities, such as security and performance.
- Testability: The elements of the architecture that make the system easier to test at all test levels.
- Ease of administration: The elements of the architecture that make the system easier for operators to manage, including backup and restore, data migration, account management, and so forth.
- Debug-ability and monitoring: The elements of the architecture that make it easier to find defects and to monitor for operational anomalies, during both development and operation.

- Development productivity: the elements of the architecture that make the developers more productive.

Specific questions and considerations are associated with each of these attributes, and the online resource helpfully provides metrics for each.⁷

5.2.2 Deutsch's Design Review Checklist

Let's look at an example of a checklist we can use to review distributed applications. The checklist comes from some work done by L. Peter Deutsch and others at Sun Microsystems in the 1990s. You might remember Sun's early slogan: "The network is the computer." Sun was in the forefront of distributed application design and development. Deutsch and his colleagues recognized that people designing and developing distributed applications kept making the same mistakes over and over again. They often made these mistakes because of eight typical, incorrect assumptions (which he called *fallacies*).

We can create a checklist based on Deutsch's eight distributed application fallacies. The idea of a checklist is to force you to think. When you're performing a review, we want you to think. That makes a checklist a good tool to use. Failing to consider these fallacies when designing a distributed application is guaranteed to cause issues once the application is delivered.

1. The network is reliable. It will never go down. Murphy once said what can go wrong, will go wrong. Heinlein's corollary to that is, "Murphy was an optimist." The network is made up of hardware and software. We already know that software can fail (we are, after all, professional pessimists). Anyone who has ever owned hardware knows that it can fail. Power can get interrupted, cables can get disconnected. People can do stupid things. In the long run, you can be assured that the network will occasionally go down; the question is what will happen to your application when it does.
2. Latency is zero. How long does it take for your data to go from here to there? It seems like it is fast—speed of light, right? If the network is local, it might be close to zero. What happens if it is a wide area network? The user is two continents away. Well, it is still pretty fast. Is it fast enough? What would need to be so fast anyway? In today's world, you might be doing some distributed processing. Running web services from anywhere. Remote proce-

7. For the full list, including the specific elements to look for and metrics to check, see <http://www.codeproject.com/KB/architecture/SWArchitectureReview.aspx>. There is also information on architecture reviews available at <http://portal.acm.org/citation.cfm?id=308798>.

dure calls from anywhere. When they are local, they are almost instantaneous; when they are remote, they are not. How is a delay in getting an answer to a called function going to affect the computation? What happens when that answer is needed in real time? How about if there are multiple processors waiting for the answer? There are suddenly failure possibilities due to timing. Are you holding locks while you wait? Hmm.

3. Bandwidth is infinite. Each year it is getting better—in most countries if not the United States. The United States is actually falling behind, although there are plans to bring us into the 21st century. As we travel around the United States teaching classes, it blows us away how many people are still on dial-up connections. So, when our fancy distributed application is downloading 5 megabytes of company logos to our customer's computer through that 40 Kbps dial-up, they won't mind a bit.
4. The network is secure. If you truly believe that, you might want to go back and review the technical security section in Chapter 4. As Jamie was writing this section, he looked up and saw that his virus checker icon in the toolbar had become disabled. Several times an hour that occurs when it is accessing the update center, so he did not think about it. Not until, that is, he looked at it minutes later and saw that it had not yet come back. Checking the other three computers in his office, Jamie saw that each was showing disabled icons. He immediately killed the network, shut down all four computers, and spent the next three hours going through each one trying to find the issue. When he couldn't find any problems, he shut down for the night. It turned out to be a false alarm, but the fact is that, if you have been in computers for more than a couple of hours, you have heard of hackers, crackers, thrill seekers, and assorted mopes who break stuff just for kicks. The network is never secure.
5. Topology doesn't change. At least, it doesn't until you put the application into production. It doesn't matter if your distributed application is in-house or worldwide; the only constant in networks is change. It may not change today, or tomorrow, but it will change. How long is your application going to be in use? Even the best prognosticators in the business have no idea what technical advances are going to come out next week or next year. The network must be abstracted away without depending on any given resource being constant.
6. There is one administrator. Sorry, there will be a bunch. Even if your application is in-house, people move on. What does it matter? Expertise, training, problem solving: all of these and more are going to be handled by the ad-

ministrator of the system. Tools will be needed to solve issues with deployment. Where will they get the tools, the training, and the expertise? How user friendly will the system actually be? When you start looking at the details of deployment, you will have to look at the lowest common denominator and figure out how to deal with the administrators.

7. Transport cost is zero. Here you need to look at both time and money. To get the data from here to there takes time. This has to do with the idea of latency we discussed earlier. It takes time to go through the stacks of software, firmware, and hardware and then through the copper or fiber and finally through the hardware, firmware, and software at the other end. This takes time and processing cycles and those are not free. In addition, you are putting more load on already overloaded systems; there are times when new software and hardware are going to be needed. The less efficient your system, the more this will be true.
8. And finally, the network is homogenous. Writing a Windows application? What happens if your client uses Linux? Your application works in UNIX, but your client has a Series 5 app server? Proprietary protocols and services will get you every time. Interoperability is going to be essential to any application that is ever going to be moved outside the lab. Did you design for that?

Any review of a network application should include discussion of all these points.

5.2.3 Some General Checklist Items for Code Reviews

Technical test analysts are likely to be invited to code reviews. So, in the syllabus, a list of general items to consider for a code review checklist is also provided.⁸ As with any externally obtained checklist, you should tailor this list to your specific needs, considering the programming language(s) used in your organization, the major risks for your product, and whatever priorities your organization might have. As mentioned previously, the inclusion of best practices and worst practices (or patterns and anti-patterns, if you prefer) will be helpful in detecting problems.

Static code analysis tools can actually address a number of the items mentioned in the syllabus code checklist. It's Rex's opinion that all such items should

8. Well, the list is actually sourced from Karl Wiegers, found at his website http://www.proces-simpact.com/pr_goodies.shtml. So, we suppose that Karl actually provided it. Thanks Karl!

be handled by such a tool and that you remove any item handled by the tool from the checklist. The most tedious discussions Rex has ever heard in code reviews involved stylistic questions such as, “Which line does the closing brace of a code block belong on?” and “Why aren’t you using Hungarian naming convention for your variable names?” Rex would almost rather shove a sharp pencil in his eye than sit through another such pointless discussion. The beauty of making these decisions about style and other such automatable checkpoints once, then enforcing them via a tool, is that you can permanently outlaw such ridiculous arguments from code review meetings.

Let’s look at some of the items that you might include in your code review checklist, starting with the general items from the syllabus.

The checklist starts with the structure of the code. It should be evaluated against any architecture and design specifications to ensure consistency and completeness. Look for any test software or test-enabling software. (Early in his career, when working as a programmer, Rex made the mistake of leaving a bunch of debug print statements in a piece of code that ultimately went to the customer, which proved quite embarrassing.) You should consider whether the use of memory is inefficient, such as allocating a very large multidimensional array that will only be sparsely populated with actual values. Look at any chunks of code to see whether they might correspond to library functions or system calls that could be used. The checklist also mentions coding standards, style, formatting, uncalled functions, unreachable code, repeated code, the use of hard-coded constants, and excessive complexity in this category, but Rex feels that all of these issues are best delegated to a static code analysis tool, as mentioned previously.

Next, the checklist looks at documentation of the code, which refers to commenting on the code. If there are any applicable style guidelines—for example, for class declaration and function headers—then those can often be checked to some extent by static code analysis tools. However, the comments are ultimately there to help people, usually programmers, understand and maintain the code, so the participants of the review should evaluate the comments carefully. The comments must make sense, describe what the code does in terms that make it clear why the code does what it does, and include all information necessary for ongoing maintenance of the code. The comments also should not simply parrot what is in the code. They should talk about why rather than what (which can be determined by reading the code itself).

The next category on the checklist is variables. Ideally, the kind of dataflow analysis discussed in Chapter 3 will be performed prior to the code review, so

that any problems with dataflows are identified and removed first. Static code analysis tools should be able to detect unused variables, type mismatches of various sorts, and improperly named variables. So, checking whether the variable names are meaningful and add to the readability of the code is probably all the review need do, given the use of these other techniques.

It's in computational and programming logic code that reviews really are without parallel, irreplaceable by tools, since these areas are where the programmer tells the computer what they want it to do. They are also the areas where some very serious bugs live. In terms of computations, it's important to always remember that computers operate on values that are, in significant ways, different than the real-world or mathematical contracts they represent. Rounding errors will occur, and will accumulate and often be amplified, if care is not taken. Integers in a computer behave much like integers in the real world and in math, but there are those pesky upper and lower boundaries imposed by storage sizes. Floating-point numbers are even more constrained by the way in which computers represent them, which can lead to problems with equality comparisons. Division by zero is a risk. Overflow, underflow, and loss of information can happen when mathematical operations occur on numbers with very different sizes, such as dividing a very large floating-point number by a floating-point number very close to zero.

In terms of the programming logic, anytime the program makes a decision, we can have problems in the decision itself (as discussed in Chapter 2), or in the way in which the decision is coded, or in the failure to code some important alternative decision. Any branching, looping, or other logic construct must be checked to make sure nothing is missing, everything is correct, and, if any nesting occurs, the nesting is done properly. To aid efficiency, programming should be done to take advantage of short-circuiting and, in the case of if-elseif chains, the order of the checking, so that the cases most likely to occur are at the top of the chain and in the right place in any complex conditions to minimize condition evaluation. Check to make sure nothing is missing in any switch-case construct or if-elseif chain, including the final else or default.

Loops, being places where blocks of code are repeated dozens, maybe hundreds, maybe thousands of times, are very sensitive areas of a program and require the highest degree of scrutiny. Since an infinite loop can result in a hung application, check that each loop has clear conditions for termination and that those conditions will always be met at some point. If pointers, arrays, lists, stacks, or other such data structures will be accessed within the loop, make sure that they are properly initialized before the loop starts. Since each statement

inside the loop consumes CPU cycles, check to see if anything inside the loop can happen outside the loop. Finally, if the loop uses an index or counter variable, such as a for loop in C and C++, that variable should only be used in the body of the loop, not set, and should not be used after the loop is completed.

Finally, there are certain programming practices, referred to as defensive programming, that help programs avoid abnormal termination, even when something goes awry within the program. Anytime a data structure such as an array, list, record, structure, or file is accessed, the program should check that it is not trying to access beyond the end of the list (e.g., the infamous buffer overflow bug). Any variable that will be used for an output, such as a return value from a function, should be initialized upon declaration, even if it seems impossible that the variable won't be set later. When data is used or set, check that the program is acting on the right element; for example, when accessing a two-dimensional array, it's easy to get the indices reversed. If heap memory is used in a language such as C or C++ where the programmer must manage the heap, check to make sure all heap variables are eventually released. (Of course, dynamic analysis tools are also useful in checking for such problems, and static code analysis tools will also watch for potential memory leaks.)

Finally, a lot can go wrong when interacting with the great wide wild world. Data coming into the program from an outside source, such as the user interface, an API, or a file produced by another program, should be validated, including making sure nothing is missing and nothing that shouldn't be there is present. The program shouldn't assume that external devices will respond or that files exist; time-outs, error traps, and checking files for existence before using them is smart. And, when the program is exiting—under whatever conditions, normal or abnormal—it should return the files, devices, and any other resources used to the correct state.

5.2.4 Marick's Code Review Checklist

So, let's go through Brian Marick's code review checklist, which he calls a "question catalog." This catalog has several categories of questions that developers should ask themselves when going through their code. These questions are useful for many procedural and object-oriented programming languages, though in some cases certain questions might not apply.⁹

9. This is drawn from Brian Marick's *The Craft of Software Testing*.

For variable declarations, Marick says we should ask the following questions:

- Are the literal values correct? How do we know?
- Has every single variable been set to a known value before first use? When the code changes, it is easy to miss changing these.
- Have we picked the right data type for the need? Can the value ever go negative?

For each data item and for operations on data items, Marick says we should ask the following questions:

- Are all strings NULL terminated? If I have shortened or lengthened a string, or processed it in any way, did the final byte get changed?
- Did we check every assignment to a buffer for length?
- When using bitfields, are our manipulations (shifts, rotates, etc.) going to be portable to other architectures and endian schemes?
- Does every sizeof() function call actually go to the object we meant it to?

For every allocation, deallocation, and reallocation of memory, Marick says we should ask the following questions:

- Is the amount of memory sufficient to the purpose without being wasteful?
- How will the memory be initialized?
- Are all fields being initialized correctly if it is a complex data structure?
- Is the memory freed correctly after use?
- Do we ever have side effects from static storage in functions or methods?
- After reallocating memory, do we still have any pointers to the old memory location?
- Is there any chance that the memory might be freed multiple times?
- After deallocation, are there still pointers to the memory?
- Are we mistakenly freeing data we don't mean to?
- Is it possible that the pointer we are using to free the memory is already NULL?

For all operations on files, Marick says we should ask the following questions:

- Do we have a way of ensuring that each temp file we create is unique?
- Is it possible to reuse a file pointer while it is pointing to an open file?
- Do we recover each file handle when we are done with it?
- Do we close each file explicitly when we are done with it?

For every computation, Marick says we should ask the following questions:

- Are parentheses correct? Do they mean what we want them to mean?
- When using synchronization, are we updating variables in the critical sections together?
- Do we allow division by zero to occur?
- Are floating-point numbers compared for exact equality?

For every operation that involves a pointer, Marick says we should ask the following questions:

- Is there any place in the code where we might try to dereference a NULL pointer?
- When dealing with objects, do we want to copy pointers (shallow copy) or content (deep copy?)

For all assignments, Marick says we should ask the following question:

- Are we assigning dissimilar data types where we can lose precision?

For every function call, Marick says we should ask the following questions:

- Was the correct function with the correct arguments called?
- Are the preconditions of the function actually met?

Finally, Marick provides a couple of miscellaneous questions:

- Have we removed all of the debug code and bogus error messages?
- Does the program have a specific return value when exiting?

As you can see, this is a very detailed code review checklist. However, customization based on your own experience, and your organization's needs, is encouraged.

5.2.5 The Open Laszlo Code Review Checklist

In the previous section, we discussed Marick's questions that should be asked about the code itself. In this section, we will discuss questions that should be asked about the changes to the system. These are essentially meta-questions about the changes that occurred during maintenance. These come from the OpenLaszlo website.¹⁰

10. The complete list can be found at http://wiki.openlaszlo.org/Code_Review_Checklist.

For all changes in code, here are the main questions we should ask:

- Do we understand all of the code changes that were made and the reasons for them?
- Are there test cases for all changes?
- Have they been run?
- Were the changes formally documented as per our guidelines?
- Were any unauthorized changes slipped in?

In terms of coding standards, here are some additional questions to ask (assuming you are not enforcing coding standards via static analysis):

- Do all of the code changes meet our standards and guidelines? If not, why not?
- Are all data values to be passed parameterized correctly?

In terms of design changes, here are the questions to ask:

- Do you understand the design changes and the reasons they were made?
- Does the actual implementation match the designs?

Here are the maintainability questions to ask:

- Are there enough comments? Are they correct and sufficient?
- Are all variables documented with enough information to understand why they were chosen?

Finally, here are the documentation questions included in the Open Laszlo checklist:

- Are all command-line arguments documented?
- Are all environmental variables needed to run the system defined and documented?
- Has all user-facing functionality been documented in the user manual and help file?
- Does the implementation match the documentation?

We would also add one additional question that should be checked by the testers for every release: Do the examples in the documentation actually work? We have both been burned too often by inconsistencies—in some cases quite serious—between examples and the way the system actually works.

5.3 Deutsch Checklist Review Exercise

As you can see in the diagram at the beginning of the HELLOCARMS system requirements document, the HELLOCARMS system is distributed. In fact, it's highly distributed, as multiple network links must work for the application to function.

In this exercise, you apply Deutsch's distributed application design review checklist to the HELLOCARMS system requirements document.

This exercise consists of three parts:

1. Prepare: Based on Deutsch's checklist, review the HELLOCARMS system requirements document, identifying potential design issues.
2. Hold a review meeting: Assuming you are working through this class with others, work in a small group to perform a walk-through, creating a single list of problems.
3. Discuss: After the walk-through, discuss your findings with other groups and the instructor.

The solution to the first part is shown in the next section.

5.4 Deutsch Checklist Review Exercise Debrief

Senior RBCS Associate Jose Mata reviewed an early version of the HELLOCARMS system requirements document (which did not include the nonfunctional sections). Jamie Mitchell reviewed the latest version of the document. Both used Wiegers' checklist to guide their reviews.

- The network is reliable. It will not go down, or will do so only very infrequently.

Jose: HELLOCARMS design in Figure 1 does not include any redundancy as in failover servers, backup switches, or alternate networking paths. Contingencies for how the system will not lose information when communication is lost are not defined.

Jamie: This version of the requirements still does not address system-wide reliability. Both fault tolerance (020-020-010) and recoverability (020-030-010) are seen only from the Telephone Banker's point of view.

- Latency is zero. Information arrives at the receiver at the exact instant it left the sender.

Jose: Efficiency was not discussed in the draft of the requirements that was reviewed.

Jamie: Time behavior is fairly well defined for the Telephone Banker front-end side (040-010-010, -060, -070 and -080). Some system-wide latency has been addressed (040-010-040 and -050). Other systems are not so defined.

- Bandwidth is infinite. You can send as much information as you want across the network.

Jose: Efficiency was not discussed in the draft of the requirements that was reviewed.

Jamie: While the amount of information to be transmitted has not been defined (nor should it be in the requirements document), resource utilization has been addressed for the database server (040-020-010), the web server (040-020-020), and the app server (040-020-030). While some definitions are not complete, we would expect to clarify them based on the static review.

- The network is secure. No one can hack in, disrupt data flows, steal data, and so on.

Jose: Section 010-040-010 doesn't include enough detail to lend confidence.

Jamie: In this latest version, the security section has not changed. Overall security has not been well defined or (seemingly) considered.

- Topology doesn't change. Every computer, once on the network, stays on the network.

Jose: In the earlier version of the requirements document, fault tolerance and recoverability, in section 020, were TBD. The part regarding application is covered in section 010-010-060.

Jamie: Fault tolerance and recoverability are defined only at the Telephone Banker front-end level. Reliability for the rest of the system is currently underdefined.

- There is one administrator. All changes made to the network will be made by this one person. Problems can be escalated to this one person. This person is infallible and doesn't make mistakes.

Jose: Specific types of users, and their permissions, are not defined.

Jamie: Some attempts at defining usability (learnability: 030-020-010 and -020) have been made, but only at the Telephone Banker front-end level. Actual administration of the system has yet to be addressed.

- Transport cost is zero. So, you can send as much information as you want and no one is paying for it.

Jose: There are no size limitations in section 010-010-160, “Support brokers and other business partners by providing limited partner-specific screens, logos, interfaces, and branding.” Some graphics can be large, if left undefined.

Jamie: Throughput values have been defined (040-010-110 to 040-010-160). The actual issue of cost of that throughput has not been defined.

- The network is homogeneous. It’s all the same hardware. It’s all the same operating system. It’s all the same security software. The configurations of the network infrastructure are all the same.

Jose: Supported computer systems, operating systems, browsers, protocols, etc., are not defined. Versions should be specified, though perhaps this detail should be in a design document rather than a requirements specification.

Jamie: The new version of the requirements document does not change this.

5.5 Code Review Exercise

In this exercise, you apply Marick’s and the Open Laszlo code review checklists to the following code.

1. Prepare: Review the code, using Marick’s questions, Laszlo’s checklist, and any other C knowledge you have. Consider maintainability issues as you review the code. Document the issues you find.
2. Hold a review meeting: If you are using this book to support a class, work in a small group to perform a walk-through, creating a single list of problems.
3. Discuss: After the walk-through, discuss your findings with other groups and the instructor.

The solution to the first part is shown in the next section.

Here is the code that we are reviewing. This code performs a task by getting values from the user, performing a calculation, and then printing out the result. On subsequent pages, we will have a debrief for this exercise.

```
1.     getInputs(float *, float *, float* );
2.     float doCalcs(float, float, float);
3.     ShowIt(float);
4. main() {
5.     float base, *power;
6.     float Scaler;
7.     getInputs(&base, power, &Scaler);
8.     ShowIt(doCalculations(base, *power));
9. }
10. void getInputs(float *base, float power, float *S) {
11.     float base, power;
12.     float i;
13.     printf("\nInput the radix for calculation => ");
14.     scanf("%f", *base);
15.     printf("\nInput power => ");
16.     scanf("%f", *power);
17.     printf ("\\nScale value => ")
18.     scanf("i", i);
19.     *Base = &base;
20.     *P = &power; }
21. float doCalcs(float base, float power, float Scale){
22.     float total;
23.     if (Scale != 1) total == pow(base, power) * Scale;
24.     else total == pow(base, power);
25.     return; }
26. void ShowIt(float Val) {
27.     printf("The scaled power value is %f W.\\n", Val);
28. }
```

5.6 Code Review Exercise Debrief

This code is representative of code that Jamie frequently worked with when he was doing maintenance programming. Rex will let Jamie describe his findings here.

Let's start with some general maintainability issues with this code:

- No comments.
- No function headers. I have a standard that says that every callable function gets a formal header, explaining what it does, the arguments it takes, the return value, and what the value means. I also include change flags and dates, with an explanation for each change.
- No reasonable naming conventions are followed.

- I would prefer Hungarian notation¹¹ so we can discern the data type automatically.
- No particular spacing standards used, so code is not as readable as it might be.

Based on Marick's checklist and a general knowledge of C programming weaknesses and features, here are some specific issues with this code:

- Line 0: Not shown: We need the includes for the library functions we are calling. We would need stdio.h (for printf() and scanf()) and math.h (for pow()). These problems would actually prevent the program from compiling, which should be a requirement before having a code review.
- Line 1: Every function should have a return value; in this case, void.
- Line 2: No issues.
- Line 3: Once again, the function should have a return value.
- Line 4: This might work in some compilers, but the main should return a value (int), and if it takes no explicit arguments, it should have void. This is a violation of Marick's miscellaneous question, "Does the program have a specific exit value?"
- Line 5: The variable power is defined as a pointer to float, but no storage is allocated for it. Near as I can tell, there is no reason to declare it as a pointer, and it should simply be a local float declared. Note that these variables are passed in to a function call before being initialized. This could be seen as a violation of Marick's declaration question, "Are all variables always initialized?" Since no data has been allocated, this is a violation of Marick's allocation question, "Is too little (or much) data being allocated?" And, just to make it interesting, assuming that the code was run this way, it would be possible to try to dereference the pointer "*power", which breaks Marick's pointer question, "Can a pointer ever be dereferenced when NULL?"
- Line 6: Variable is passed in to a function call before being initialized. This is a violation of Marick's declaration question, "Are all variables always initialized?"
- Line 7: The function call arguments are technically correct since the variable power was defined as a pointer. However, the way it is written it will blow up

11. *Hungarian notation* is a term that originated with Microsoft thanks to its chief architect, Dr. Charles Simonyi. In Hungarian notation, the variable has a prefix that indicates the type. After its adoption within Microsoft, it spread to other companies, with the name "Hungarian notation" since it makes variables look foreign and because Simonyi was born in Hungary.

since there is no storage allocated. This is a violation of Marick's allocation question, "Is too little (or much) data being allocated?" Since I would change power to a float in line 5, this argument would have to be passed in as "&power" just like the other arguments.

- Line 8: Same issue with power; it should be passed by value as just "power". Also, the function doCalculations() does not exist. It should be doCalcs(). And, if they are meant to be the same function, the argument count is incorrect.
- Line 9: No issue.
- Line 10: "S" is not a good name for a variable.
- Line 11: The local variables have exactly the same name as the formal parameters passed in. I would like to think that this naming would prevent the module from compiling; I fear it won't. It certainly will be confusing. If you must name the local variables the same as the parameters (considering the way they are used, it makes a little sense), then change the capitalization to make them explicitly different. I would capitalize the local variables Base and Power.
- Line 12: While this is legal, it is a bad naming technique. The variable "I", when used, almost always stands for an integer; here it is a float. At the very least it is confusing. This should likely match the others and be renamed, "Scaler".
- Line 13: No issue, although the prompt message is weak.
- Line 14: No issue.
- Line 15: No issue.
- Line 16: No issue.
- Line 17: The line feed is backwards: should be \n and not /n.
- Line 18: We should be loading the value of S with this scanf() function. There is no need for the local variable i.
- Line 19: Let me say that I hate pointer notation with a passion. Here, we are assigning a pointer to the value pointed to by *Base. What we really want to do is assign the actual value; the statement should read "*base = Base" (assuming we made the change in Line 11 to its name).
- Line 20: Same as line 19, and I still really hate pointer notation. Also, we are not returning any value to the third argument of the getInputs() function. There should be a statement that goes, "*Scaler = scaler" (assuming we change the name of the variable as suggested in Line 12). *P is never declared; it is also a really poor name for a variable. Finally, the closing curly brace should not be on this line but moved down to the following line. That is the same indentation convention that we use for the other curly braces.

- Line 21: No issue.
- Line 22: No issue.
- Line 23: We are doing an explicit equivalence check on a float [if (Scale != 1)]. This is a violation of Marick's computation question, "Are exact equality tests used on floating-point numbers?" On some architectures, I would worry about whether the float representation of 1 is actually going to be equal to 1. The problem is that I really don't know what this scalar is supposed to do. It looks like the way the code is written, "Scale" is only there to save a multiplication if it is equal to 1. I would want to know if the user can scale at 5.3 (or any other real number) or if they could only use integers. If they could only input integers, I would change the data type to int everywhere it is used. If there is a valid reason to input a real number (i.e., one with a decimal), then I would lose the if statement and simply do the multiplication each time. Comments would help me understand the logic being used. The wrong operator has been used; it should be an assignment statement (=) rather than a Boolean compare (==).
- Line 24: Incorrect operator; need a single equal sign.
- Line 25: The calculation is being lost because we are returning nothing. It should return the local variable value, "total".
- Line 26: No issue.
- Line 27: No issue.

While this is not required in the exercise, here's the way Jamie rewrote the code to address some of these issues.

```
1.      #include <stdio.h> // Need for I/O functions
2.      #include <math.h> // Need for pow() function
3.
4.      // We need to start with valid function prototypes
5.      void getInputs(float *, float *, float * );
6.      float doCalcs(float, float, float);
7.      void ShowIt(float);
8.
9.      // This program will prompt the user for 3 inputs. It will
10.     // use those to calculate (Base ^ Power) * Scaler.
11.     // It will then print out the calculated value
12.     int main(void){
13.         float base, power, scaler;
14.         getInputs(&base, &power, &scaler);
15.         ShowIt(doCalcs(base, power, scaler));
16.     }
17.
```

```
18.     // This function prompts the user for 3 inputs and returns them
19.     void getInputs(float *Base, float *Power, float *Scaler){
20.         float base, power, scaler;
21.         printf("\nInput the radix for calculation => ");
22.         scanf("%f", &base);
23.         printf("\nInput power => ");
24.         scanf("%f", &power);
25.         printf ("Scale value => ")
26.         scanf("%f", &scaler);
27.         *Base = base;
28.         *Power = power;
29.         *Scaler = scaler;
30.     }
31.
32.     // This function performs the calculation
33.     float doCalcs(float base, float power, float Scale){
34.         float total;
35.         if (Scale != 1) {
36.             total = pow(base, power) * Scale;
37.         }
38.         else {
39.             total = pow(base, power);
40.         }
41.         return total;
42.     }
43.
44.     // This function prints out the returned value
45.     void ShowIt(float Val){
46.         printf("The scaled power value is %f W.\n", Val);
47.     }
```

Finally, Open Laszlo's checklist is concerned with changes, but there are some good rules there for any code. Let's go through this one explicitly.

■ Main questions

- Do you understand the code? No! No idea what this code is doing.
- Are there test cases for all changes? No! No test cases were defined at all.
- Were the changes formally documented as per guidelines? Unknown; this might be new code.
- Were any changes made without new feature or bug fix requests? Unknown.

- Coding standards
 - Do the code changes adhere to the standards and guidelines? Absolutely not. No comments. No function headers. Poor naming of variables.
 - Are any literal constants used (rather than parameterization)? Yes. On line 23, the literal constant “1” is used.
- Design
 - Do you understand the design? Unknown. No design document available.
 - Does the actual implementation match that design? Unknown.
- Maintainability
 - Are the comments necessary? Accurate? No comments.
 - Are variables documented with units of measure, bounds, and legal values? No.
- Documentation
 - Are command-line arguments and environmental variables documented? No.
 - Is all user-visible functionality documented in the user documentation? No documentation.
 - Does the implementation match the documentation? No documentation.

5.7 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 Technical Test Analyst exam. The questions in this section illustrate what is called a scenario question.

Walk-Through Scenario

Assume you are building an online application that allows for the secure transfer of encrypted financial data between banks, stock and bond trading companies, insurance companies, and other similar companies. This system will use public key infrastructure, and users will post their public keys on the system. The users’ private keys will be used by a thin client-side applet to decrypt the information on their local systems.

1. During preparation for a design specification walk-through, you notice the following statement:

A 10 MBPS or better network connection using TCP/IP provides the interface between the database server and the application server.

Suppose that the system under test will need to transfer data blocks of up to 1 gigabyte in size in less than a minute.

Which of the following statements best describes the likely consequences of this situation?

- A. The system will suffer from usability problems.
 - B. The system will suffer from performance problems.
 - C. The system will suffer from maintainability problems.
 - D. This situation does not indicate any likely problems.
2. During preparation for a code inspection, you notice the following header in a member function for the object "ubcd":

```
/* PURPOSE: Provides unsigned binary coded decimal integers,
*           via a class of unsigned integers of almost
*           unlimited precision. It can store a little over
*           4 billion 8 bit bytes, with each byte representing
*           an unsigned pair of decimal digits. One digit is
*           in each nybble (half-byte).
*/
```

Which of the following statements best describes why a programmer for a financial application would need to use such a binary coded decimal representation for data?

- A. This approach ensures fast performance of calculations.
- B. This approach indicates a serious design defect.
- C. This approach maximizes memory resource efficiency.
- D. This approach preserves accuracy of calculations.

3. During preparation for a peer review of the requirements specification, you notice the following statement:

The system shall support transactions in all major currencies.

Which of the following statements is true?

- A. The statement is ambiguous in terms of supported currencies.
- B. The statement indicates potential performance problems.
- C. The statement provides clear transaction limits.
- D. The statement indicates potential usability problems.

6 Test Tools and Automation

The intentions of a tool are what it does. A hammer intends to strike, a vise intends to hold fast, a lever intends to lift. They are what it is made for. But sometimes a tool may have other uses that you don't know. Sometimes in doing what you intend, you also do what the knife intends, without knowing.

—Phillip Pullman

The sixth chapter of the Advanced Technical Test Analyst syllabus is concerned with tools. The 2007 Advanced syllabus (Chapter 9) contained a discussion of test tools that might be used by all testers. In this 2012 syllabus, the tools chapter has been rescoped to discuss only the tools and techniques likely to be used by technical test analysts. We will discuss the following topics:

1. Integration and Information Interchange between Tools
2. Defining the Test Automation Project
3. Specific Test Tools

In addition, we have decided to keep some tool discussions from the first edition of our book that we believe are of interest to technical test analysts even though not necessarily in scope anymore for ISTQB.

6.1 Integration and Information Interchange between Tools

Learning objectives

TTA-6.1.1 (K2) Explain technical aspects to consider when multiple tools are used together.

Test tools can and should be made to work together to solve complex test automation tasks. 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, a defect management tool, and a graphical 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. As a technical test analyst, you should help the test manager plan for, design, and implement integration of disparate test tools.

Integration of test tools can be done automatically, if all the tools have compatible interfaces. If you buy a single vendor's test tool suite, the tools within the suite should be fully integrated out of the box. (Of course, you should consider *not* buying test suites that don't have such self-integration.) That test tool suite might not integrate with your other tools. In the more likely case that some of the interfaces aren't compatible—or maybe don't even exist—you should plan to create integration modules to tie the tools together. Such modules are sometimes referred to as glue or middleware. While there are costs associated with building integration modules, it provides significant benefits and risk-reductions.

First, by actively sharing information across tools, integrity constraints can be put in place to ensure consistency across the tools. Related information that is stored in different places quickly gets out of sync without an active process to keep it in sync. Test results reports that are created using information from disparate and inconsistent repositories will be confusing at best, and more likely plain wrong. For example, integrity constraints can ensure that test case ID numbers stored in two repositories are kept in sync if the master repository of test cases is updated in such a way that test case ID numbers are changed.

Second, if information must be manually merged across tools, not only is that prone to error, it's also very costly. Rex once had to spend about an hour every day merging data from two different test case tracking systems. With proper tool integration, this data merging happens regularly, automatically, invisibly, and without tester intervention or effort.

When integrating test tools, focus on the data, because getting consistent data across all the tools is the essential element here. The data should be merged in a fully automated fashion, happening on a regular basis and also available on demand if you need to force a synchronization. The repositories should be set up so that no inaccuracies or failed data copies can occur invisibly; in other words, in the rare (and it must be rare) circumstance that one or more pieces of

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, the logging of results, progress tracking, incident management, and test reporting.

information fails to synchronize, an administrator must be informed. If a network connection or server goes down, the integration facilities should recover automatically, including catching up any missed data synchronizations.

It's easy to focus on the user interface across the integrated tools, but the data is the important thing. Yes, it would be nice to have the tools work in a consistent way, but unless you are building custom tools, that will be hard to do after the fact. What's more important—and should be considered a must-have during tool acquisition—is the ability to ensure that across all tools, data is captured, stored, and presented consistently. Security of the data is also important. Otherwise, people can inadvertently or on purpose damage data in one weakly secured tool, then watch in horror or glee as that damaged data propagates across all the other tools.

If it sounds like we're describing a small development project, yes, that's right, we are. As such, it needs planning and design. Rex has found that such projects lend themselves well to iterative approaches such as Agile or spiral life cycle models, but there is a need for an overall design that identifies all the tool touchpoints, their interfaces, and the data to be shared. If you aren't used to doing this kind of work, you might want to involve a skilled business analyst in your organization. Just jumping in and starting work on integration of tools may turn out to be the start of an open-ended project where work is never done and no one is ever satisfied with the results.

Figure 6–1 is an example of an integrated automated test system built for an insurance company. The system under test—or, more properly, the system of systems under test—is shown in the middle.

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, claim history, accounts payables and receivables, and the like.

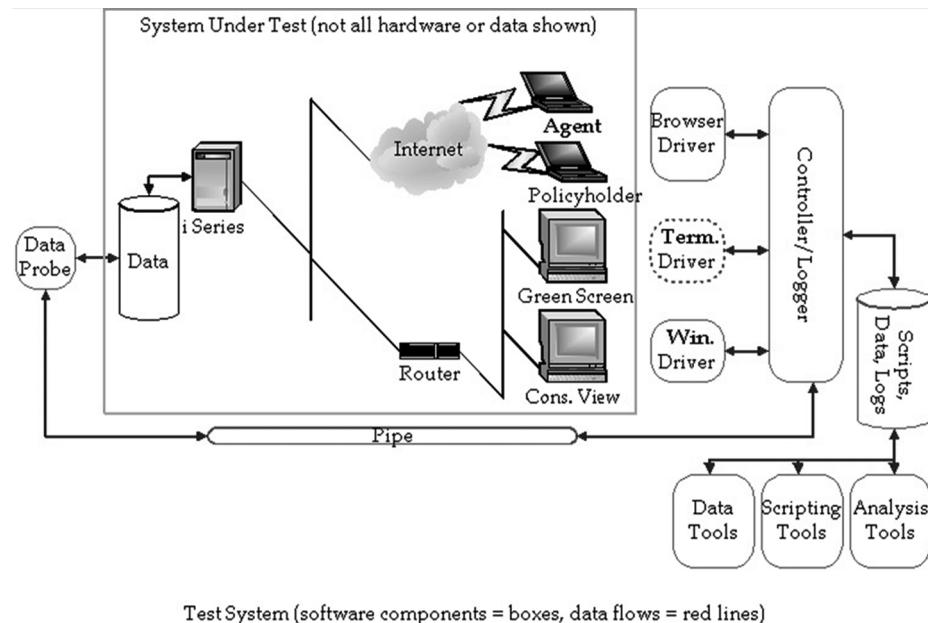


Figure 6-1 Integrated test system example

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 with a dotted line around it because there was some question initially about whether that would be needed. The controller/logger piece uses the drivers to make tests run, 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 figure, 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 us on this one; we both have the scars to prove it! If you let the tools drive the testing, you can end up not testing important things.

This brings us 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.

6.2 Defining the Test Automation Project

While we often think of test automation as meaning specifically the automation of test execution, we can automate other parts of the test process as well. You would be correct in thinking that most of the test automation that happens involves attempts to automate tasks that are tedious or difficult to do manually. These tasks include test and requirements management, defect tracking and workflow, configuration management, and certainly test execution tasks such as regression and performance testing.

Getting the full benefit from a test tool involves not only careful selection and implementation of the tool but also careful ongoing management of it. Too often, an automation project fails because the only guidance the automation team received was, “Get ‘er done!”

The success or failure of an automation project can often be determined very early in the project, based on the extent of preparation, the design, and the architecture built before starting to crank out tests. Too often as consultants and practitioners we’ve seen test teams saddle themselves with constraints due to poor design decisions made at the outset of automation. There is only one way to start an automation program: thinking long term. The decisions that you make at the beginning will be with you for years, unless, like many organizations, you paint yourselves into a corner and have to start fresh down the road.

6.2.1 Preparing for a Test Automation Project

Learning objectives

TTA-6.2.1 (K2) Summarize the activities that the technical test analyst performs when setting up a test automation project.

You should plan to use configuration management for all test tool artifacts, including test scripts, test data, and any other outputs of the tool, and remember to link the version numbers of tests and test tools with the version numbers of the items tested with them.

We will use two related terms, *architecture* and *framework*. These two terms are often used interchangeably, but they will not be in this book. An architecture is a conceptual way of building an automated system. Architectures we will discuss include record/playback, simple framework, data-driven, and keyword-driven architectures. A framework is a specific set of techniques, modules, tools, and so on that are molded together to create a solution using a particular architecture. We will discuss these differences in detail later in the chapter.

When automating test execution, you should plan to create a proper framework for your automation system. A good framework supports another important aspect of good test execution automation, which is creating and maintaining libraries of tests. With consistent decisions in place about the 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 libraries in which to store those.

Automated tests are programs for testing other programs. So, as with any program, the level of complexity and the time required to learn how to use the system means that you'll want to have some documentation in place about how it works, why it was built the way it was, and so forth. Eventually, the original architects will be gone; any knowledge not documented will be long gone too. Documentation doesn't have to be fancy, but any automated test system of any complexity needs it.

Remember to plan for expansion and maintenance. Failure to think ahead, particularly in terms of how the tests can be maintained, will reduce the scal-

ability of the automated system; that will reduce the possibility of getting positive value on your automation project.

ISTQB has moved the section discussing the business case for automation to the Advanced Test Manager syllabus. While we agree with that decision generally, we believe that a technical test analyst must also be aware of the “why” of automation in order to build a system that will generate the most value for the organization. Therefore, in this edition of our book, we will still include some discussion on the benefits of automation.

Test automation should occur only when there is a strong business case for it, usually one that involves shrinking the test execution period, reducing the overall test effort, and/or covering additional quality risks that could not be covered by manual testing.

When we talk about the benefits of automation, notice that these are benefits compared to the duration, effort, or coverage we would have with manual testing. The value of our automation has to be considered in terms of comparing it to other alternatives we might choose. Importantly, those alternatives must be alternatives that the organization actually would have pursued. In other words, if we automate a large number of tests but they are tests we would not bother to run manually, we should not claim a return on investment in terms of time savings compared to manual execution of those tests. As Rex often says, just because something's on sale doesn't mean that it's a bargain; it's not if you don't need it.

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, 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. Your test processes will have to change. If they don't change, then what benefit are you actually getting from the tool?

Recurring costs include the following:

- Maintaining the tool(s) 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 another way, to maximize test durability.
- Paying for ongoing license fees.
- Paying support fees for the tool.
- Paying for ongoing training costs.
- Porting the tests to new platforms (including OS upgrade).
- 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.

It's a natural temptation to skip thinking about planned quality improvement of the automation system. However, with a disparate team of people doing test automation, not thinking about it guarantees that the tool usage and scripts will evolve in incompatible ways and your reuse opportunities will plummet. Trust us on this; we saw a client waste well over \$250,000 and miss a project deadline because there were two automation people creating what was substantially the same tool using incompatible approaches.

In the Foundation syllabus, you'll remember that there was a recommendation to use pilot projects to introduce automation. That's a great idea. However, keep in mind that pilot projects based on business cases will often miss important recurring costs, especially maintenance. Pilots are often targeted using only short-term thinking. We suggest trying to leverage the pilot into considering long-term issues also.

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.

Due to the very high fixed costs of automation, we will usually have to worry about the scalability of the testing strategy. That is, we usually need to do a lot of testing to amortize the fixed costs and try to earn real value on our investment. The scalability of the system will determine how much investment of time and resources is needed to add and maintain more tests.

When determining the business case, we must also consider risks. The Foundation syllabus discussed the following risks:

- Dealing with the unrealistic expectations of automation in general. Management often believes that spending money on automation guarantees success: the silver bullet theory.
- Underestimating the time and resources needed to succeed with automation. Included in this underestimation are initial costs, time, and effort needed to get started and the ongoing costs of maintenance of the assets created by the effort.
- Overestimation of what automation can do in general. This often manifests itself in management's desire to lay off manual testers, believing that the automation effectively replaces the need for manual testing.
- Overreliance on the output of a single tool, misunderstanding all of the components needed that go into a successful automation project.
- Forgetting that automation consists of a series of processes—the same processes that go into any successful software project.
- Various vendor issues, including poor support, vendor organizational health, tools (commercial or open-source) becoming orphans, and the inability to adapt to new platforms.

In this Advanced Technical Test Analyst (ATTA) book, we must also consider these risks:

- Your existing manual testing could be incomplete or incorrect. If you use that as a basis for your automated tests, guess what, you're just doing the wrong thing faster! You need to double-check manual test cases, data, and scripts before automating because it will be more expensive to fix them later. Automation is not a cure for bad testing, no matter how much management often wants to think so.
- You 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 architectures, design for test script and data reuse, and other techniques can reduce the likelihood of this happening. When it does happen, it's a test automation project killer, guaranteed, because the test maintenance work will soon consume all resources available for test automation, bringing progress in automation coverage to a standstill.

- You experience an overall drop in defect detection effectiveness because everyone is fixated on running the scripted, invariable, no-human-in-the-loop automated tests. Automated tests can be great at building confidence, managing regression risks, and repeating tests the same way, every time. However, the natural exploration that occurs when human testers run test cases doesn't happen with automated scripts. Automated tests tend to present the ultimate pesticide paradox because they tend to run exactly the same each time. You need to ensure that an adequate mix of human testing is included. Most bugs will still be found via manual testing because regression test bugs, reliability bugs, and performance bugs—which are the main types of bugs found with automated tests—account for a relatively small percentage of the bugs found in software systems.

Now, 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.

Of course, the reason we incur the costs and accept the risks is to receive benefits. What are the benefits of test automation?

First, it must be emphasized 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 we say “invest heavily” what we 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—and expensive—failures.

Once they are in place, we can expect well-designed, carefully chosen automated tests to run efficiently 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.

Given the size of the initial investment, you have to remember that it will take many months, if not years, to pay back the initial costs. Understand that in most cases, there are no shortcuts. If you try to reduce the initial costs of development, you will create a situation where the benefits of automated test execution are zero or less than zero; you can do the math yourself on how long it takes to reach the break-even point 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?

- 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.
- The ability to quickly run regression and confirmation tests creates a by-product 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 might. Now, that's a dual-edged sword, for sure, because it can lead to recklessness, but used carefully, it's a nice capability to have for emergencies.
- Since test automation is seen as more challenging and more esteemed than manual testing, many testers and test teams find the chance to work on automated testing rewarding.
- Because of the late and frequent changes inherent in certain life cycle models, especially in Agile and iterative life cycles, the ability to manage regression risk without ever-increasing effort is an essential reason for using automation. In such life cycles, regression risk is higher due to frequent code changes.
- Automation is a must for certain test types that cannot be covered manually in any meaningful way. These include performance and reliability testing. With the right automation in place, we can reduce risk by testing these areas.

This section is not intended to give an exhaustive or universal list of costs, risks, and benefits associated with test automation. Be sure to assist the test manager in their analysis of the business case, including these three elements.

6.2.2 Why Automation Projects Fail

Learning objectives

TTA-6.2.3 (K2) Summarize common technical issues that cause automation projects to fail to achieve the planned return on investment.

At the risk of going in a different order than the ISTQB ATTA syllabus specifies, we believe that this section on why automation fails must come before we start discussing automation architectures. The main reason for that is that we use different architectures to solve many of the specific problems we present here.

In this section, we want to talk about some strategies for dealing with the common failures that afflict test execution automation.

The first topic has to be who is participating in the automation project. A person who just knows how to physically operate an automation test tool is not an automator any more than a person who knows a word processing program is an author. In our careers, we have met many people who claim to be automators—especially on their resumes when they are applying for jobs. However, when pressed on how to solve particularly common automation problems, they haven't a clue. Let us be clear: The automation tool is not an automation solution; it is only the starting point. An automator must know more than how to drive the tool.

In Jamie's career, he has used almost every popular vendor automation tool—and a great many of the open-source tools as well. He can fairly claim to have succeeded with almost every automation tool at one time or another but must also admit to having failed with just about every tool at least once also. Rex has built a larger number of automated testing tools, some of which were quite successful and some of which were abysmal failures. A person who claims to be an experienced automator but cannot intelligently discuss all of the times that they failed should be avoided. Every good automator that we have met became good by learning from their failures.

The most common question Jamie is asked when teaching a class or speaking to a group of people is, “Which tool do you recommend?” Rex gets this question a lot when talking about tools as well. Behind this question, we think most people are actually asking, “Which tool can we bring in that will guarantee our automation success?” The answer is always the same. There are no “right” tools for every situation!

Consider asking a race car driver, “Which is the best spark plug to use to win a race?” A good answer might be that there are a number of good spark plugs that can be used but none of them will guarantee a win (unless of course the driver is being paid to shill for one specific brand). The fact that the car has spark plugs is certainly essential, but the brand is probably not. And so it is with automation tools.

Purely on-the-fly, seemingly pragmatic automation using any tool can work for a brief time; as problems crop up, the automator can fix them—for a short time. Eventually, such an automation program will fall over from its own weight. This failure is a certainty. The more test cases, the more problems, and the more problems, the more time will be needed to solve them. There are so many problems inherent with automation, a fully drawn-out, strategic plan is the only

chance an organization has to succeed. As an automator who has been doing automation for over 20 years, Jamie has never—NEVER—seen an automation program succeed in the long term without a fully planned-out strategy up front. As someone involved in testing and test management for almost 30 years, Rex concurs and can add that explaining this fact to management is often very difficult indeed.

So, here are some of the ingredients needed for a successful test automation strategy. First and foremost, automate functional testing for the long term. Short term thinking (i.e., we need to get the current project fully automated by next month) will always fail to earn long-term value. Judging from the number of requests we get for exactly this service—get the automation project working to test the current release we are already behind on—this chestnut is almost universally ignored.

Build a maintainable automated test framework. Think of this as the life support system for the tests. We will discuss how to do this in an upcoming section. Remember that the most important test you will ever run in automation is the next one. That is, no matter what happens to the current test—pass, fail, or warning—it means little if the framework—without direct human intervention—cannot get the next test to run. And the next test after that. The framework supports the unattended execution ability of the suite.

Unless there is an overwhelming reason to do otherwise, only automate those tests that are automatable; that is, they can run substantially unattended and human judgment is not required during test execution to interpret the results. Having said that, we have found sometimes there were good business reasons to build manual/automated hybrids where, at certain points in the execution, a tester intervenes manually to advance the test.

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 where 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. If the test only needs to be run once in a blue moon, it may be better left as a manual 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. As a good example of this, you can see the case study that Rex and a client wrote about constructing an automated monkey test from freeware components in a matter of a few weeks.¹

That said, be careful with your test tool portfolio, both open source and commercial. It's easy to have that get out of control, especially if everyone downloads their own favorite open-source test tool. Have a careful process for evaluating and selecting test tools, and don't deviate from that process unless there is a good business reason to do so.

To enable reuse and consistency of automation, make sure to provide guidelines for how to procure tools, how to select tests to automate, how to write (and document) maintainable scripts, and other similar tasks. This should entail a well-thought-out, well-engineered, and well-understood process.

Most test automation tools—at least those for execution—are essentially programming languages with various bells and whistles attached. Typically, we are going to create our testing framework in these languages and then use data or keywords in flat files, XML files, or databases to drive the tests. This supports maintainability. We will discuss this further when we talk about architectures later in this chapter.

Every tester has run up against the impossibility of testing every possible combination of inputs. This combinatorial explosion cannot be solved via automation, but we are likely to be able to run more combinations with automation than manually.

Some tools also provide the ability to go directly to an application's application programming interface (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. Tests written to the API level tend to be much more stable than those written to the GUI level because the API tends to evolve much more slowly than the GUI.

Scripting languages and their capabilities vary widely. Some scripting languages are like general-purpose programming languages. Others are domain specific, like TTCN-3 (used predominately in the conformance testing of communication systems). 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. Many modern tools support widely understood programming languages (e.g., Java, Ruby, and VBA) rather than the custom, special-purpose languages of the early tools (e.g., TSL, SQA Basic, and 4Test).

1. See “Quality Goes Bananas” on the RBCS articles page, www.rbcstech.com.

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. We've used some very solid open-source test tools, and we've also run into tools that would have to be improved to call them 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 it establishes nothing more than basic platform compatibility.

In addition to quality considerations, with open-source tools that have certain types of licenses, such as Creative Commons and the GNU Public License, you might be forced to share enhancements you create. Your company's management, and perhaps the legal department, will want to know about that if it's going to happen.

If you can't find an open-source or commercial tool, you can always build your own. Plenty of organizations do that. You might consider it if the core competencies of your organization include tool building and customized programming. However, it tends to be a very expensive way to go. In one recent engagement, Rex saw a team of three or four test engineers trying to build a test management system (with all the features of commercial test management systems) in the space of a year or so. Rex had to explain that most commercial test management systems represent person-centuries or even person-millennia of accumulated development effort, and so they were unlikely to succeed in any semblance of a realistic time frame.

Sometimes the creation of a custom tool is much easier than Rex's example in the previous paragraph. However, when building a tool is a reasonable task, there are other risks to deal with. Since one or two people develop these light-weight tools as a side activity, there's a high risk that when the tool developer leaves, the tool will become an orphan. Make sure all custom tools are fully documented.

Be aware that 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, unless you are ready to perform the certification yourself. Don't let self-certification scare you away from custom or open-source tools,

though. Rex has had a number of clients tell him that this process is relatively straightforward.

We would like to take a moment to discuss the deployment of test tools. Before deploying any tool, try to consider all of its capabilities. Often while doing a tool search for a particular capability, we have found that the organization already had a tool that incorporated that capability but no one considered it. Historically, automation tools have a very high “shelf-ware index.” Many are the times that we have found multiple automation tool sets with valid licenses sitting in the back of the lab closet. Closely related to finding unused tool sets on a shelf is to learn that a currently used tool may be extensible to deliver a needed capability. In other words, the tool currently does not have the sought-after capability, but with a little programming, configuration, and/or extension it could. Of course, in order to achieve that aim, it is essential to first understand how the tool works.

Various tools might require different levels of expertise to use. For example, a person without strong technical skills, including programming, is not likely to be successful using a performance tool. Likewise, without programming skills, the possibility that a person can be a successful automator is negligible. A test management tool should be managed by someone with strong organizational skills. A requirements management tool will tend to work much better when managed by a person with an analyst background. Make sure you match the tool to the person and the person to the tool.

When we use certain tools, we are creating software. Automation and performance tools come immediately to mind. When we create software, the output needs to be managed the way we manage other software. That includes configuration management, reviews, and inspections and testing! It always amazes us how often testers want the programmers to completely manage the software that comes out of the development team while totally ignoring all good software practices for the software that comes out of the test team.

Let's take that a little further. We are going to discuss creating different architectures for automation later. When we create an architecture for test automation, it should not be done haphazardly. We should have architecture design documents, detailed design documents, and design and code reviews as well as component, integration, and system testing. What we are building is no less of a product than the product we are going to test.

And one final note. We mentioned earlier about auditing the capabilities of the tools you are using. Audit the automation itself. What tests do you already have and what do they test? As consultants, we have often been called in to audit

an automation department to find out why they do not seem to be adding value to the test team. We have often found that they have hundreds and hundreds of scripted tests that are not doing good testing. Some of these scripts are poor because they directly automated manual tests without considering whether the tests were actually automatable (not every test is). We have found automated tests that had no way of matching expected with actual results. The assumption was made that if the script did not blow up, it must have passed. That might be clever, but it is not testing.

Earning value with automation is rarely easy and never accidental. Before deploying a tool, it is essential to understand that.

6.2.3 Automation Architectures (Data Driven vs. Keyword Driven)

Learning objectives

TTA-6.2.2 (K2) Summarize the differences between data-driven and keyword-driven automation.

Many years ago, Jamie attended a workshop on automation that was attended by many of the most experienced automators in the country. Several of the attendees got into a somewhat heated discussion as to who invented data-driven and keyword-driven automation concepts. They all claimed that they personally had come up with and developed these techniques.

They finally came to the realization that, indeed, they all had. Independently. Since then Jamie has discovered that there have been many cases in history where multiple people, needing a solution to a particular set of problems, came up with similar solutions.

In the early 1990s, everyone trying to automate testing had a common problem. Automation of testing was a meme complex² that had spread like wildfire, and there were a lot of tools being developed for it. Unfortunately, the basic capture/replay process just did not work. The basic model of capture/replay was really an unfunny joke. Virtually every person who wanted to be a serious automator, who saw the possibilities in the general idea while failing miserably in the execution, tried to come up with solutions. Many would-be automators fell by the wayside; but many of us persevered and eventually came up with solutions that worked for us. Many of these solutions looked very similar. Look-

2. A meme is an idea, pattern of behavior, or value that is passed from one person to another. A meme complex is a group of related memes often present in the same individual's mind.

ISTQB Glossary

data-driven testing: A scripting technique that stores test input and expected results in a table or spreadsheet so that a single control script can execute all of the tests in the table. Data-driven testing is often used to support the application of test execution tools such as capture/playback tools.

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.

record (capture)/playback tool: A type of test execution tool where inputs are recorded during manual testing in order to generate automated test scripts that can be executed later (i.e., replayed). These tools are often used to support automated regression testing.

test execution tool: A type of test tool that is able to execute other software using an ATA automated test script (e.g., capture/playback).

ing back, we guess it would have been strange if we did not come up with the same solutions—we all had the same problems with the same tools in roughly the same domain.

In the next few paragraphs, we want to discuss the natural evolution in automation. We call it natural because there was no sudden breakthrough; there was just a step-by-step progression that occurred in many places.

The driver of this natural evolution is traditionally thought of as return on investment (ROI). Test automation may not fit within the traditional concept of an investment, but it must add value to the organization that wants to use it. If done correctly, automation can add value. When done well, it can add great value. But when an automation program is set up or executed poorly, the organization would be better off throwing their money into an active volcano than spending it on automation.

The costs of an automation program are typically very high. Therefore, the value that program must provide must also be very high. One way to provide value is to allow the organization to run many tests it could not otherwise run.

Back in the early days of automation, many automators could create large numbers of tests. Unfortunately, the ability to run the tests effectively was spotty—for a variety of reasons we will discuss later. Having a large number of tests that could not be run successfully was a problem all automators faced.

We needed a solution. The most common solution, as you will see, was a logical progression of architectures. Many automators went from capture/replay to the framework architecture, then to data driven, and some of us went finally to keyword/action word architectures. Our terminology is not definitive; like so much of testing, there is little commonality in naming conventions. Even the ISTQB glossary has a superficial set of definitions when it comes to automation. Therefore, we will try to define our concepts with examples and you can feel free to call the concepts whatever you like.

Incidentally, the evolution of automation is still occurring. Often we have walked into an organization as consultants and found the test group reinventing the automation wheel. If your organization wants to use automation, and you do not bring in an experienced automator who has already walked this long path, you will tend to go through the same evolution, making the same mistakes. However, since the wheel has already been invented, you might consider hiring or renting the expertise. If you do bring in an automator, make sure they know what they are doing. Too often, we have seen people with resumes claiming expertise at automation when the only thing they know is a single capture/playback tool.

In his book *Outliers*, Malcolm Gladwell suggests that it takes 10,000 hours of doing something to become an expert. Ten thousand hours works out to five years of continuous employment; that is, 40 hours per week, 50 weeks a year, for five years. So, Rex has a rule of thumb that to be considered an expert, an automator should have at least five continuous years performing automation. Jamie would tend to agree, but he would include the requirement that it be *good* automation for at least four of those years (as compared to five years of fumbling around).

6.2.3.1 The Capture/Replay Architecture

So, let's take a look at the problem. You buy a tool (or bring in an open-source tool) that does capture/replay. These tools are essentially a wrapper around a programming language (the playback part) and have a mechanism to capture interface actions (keystrokes and mouse actions) and place them in a script using that programming language. At a later date, when you want to run the test again, you submit the script to an execution machine that re-creates the interface actions as if the human tester were still there.

Note that the script that was created essentially encapsulates everything you need—it has both the data and the instructions as to what to do with the data all in one place.

What could go wrong with that?

Capture/playback automation actually is a brilliant idea (other than the huge logic problems involved). It can be used, occasionally successfully, as a short-term solution to a short-term problem. If we need quick and dirty regression testing for a single release, it might work. Jamie once recorded a quick script that could be run multiple times at 3 a.m. to isolate a problem with a remote process. When Jamie came in the next morning, the script had triggered the failure, and they had a good record of it due to the recorded script. If you have a lab full of workstations and want to record a quick and dirty load test to exercise a server, you can do that.

But, if we want a stable, long-term testing solution that works every time we click the go button, well, the capture/replay tool won't do that. It is our experience, and that of every automator that we have ever spoken to, that the capture/replay architecture is completely worthless as a long-term testing solution.

In Figure 6–2 you see a recorded script from one of the all-time popular capture/replay tools, called WinRunner from Mercury-Interactive (currently owned by HP). We have removed the spaces to save room, but this is pretty much what was captured when recording. This [partial] script was generated to exercise a medical software package that was used to allow doctors to prescribe drugs and treatments for patients directly.

```
1. workstationset_window ("FREDAPP", 11);
2. edit_set ("edt_MR number", "MRE5418");
3. obj_type ("edt_MR number","<kTab>");
4. password_edit_set("edt_Password", "kzisnyixmynxfy");
5. edit_set ("Edit_2", "VN00417");
6. obj_type ("Edit_2","<kReturn>");
7. set_window ("FREDAPP", 7);
8. button_press ("No");
9. set_window ("FREDAPP", 5);
10. edit_set ("edt_MR number", " BC3456 ");
11. obj_type ("edt_MR number","<kTab>");
12. password_edit_set("edt_Password", "dzctmzgtdzbs");
13. obj_type ("edt_Password","<kReturn>");
14. set_window ("FREDAPP97 Msg", 5);
15. button_press ("Yes");
```

Figure 6–2 Partial WinRunner script

First of all, it is a little difficult to read. This script exemplifies why we have code guidelines and standards. But because it is meant to execute directly, maybe we aren't supposed to be able to read it.

Let's discuss for a moment how a human being interacts with a computer. After all, we are really trying to simulate a human being when we automate a test.

When a human being wants to interact with a GUI (in this case a Windows application), they sit down, look at the screen, and interact with what they see on the screen using the keyboard and the mouse. As mentioned earlier, GUI objects seen on the screen are generally metaphors: we see files to edit, buttons to press, tree lists to search, menus to select. There is always an active window (the one that will get input); we make it active by clicking on it. There is an active object in the window; when it is active we can tell because it usually changes somehow to let us know. There may be a blinking cursor in it, it may be highlighted, or its color may change. We deal directly with that active object using the keyboard and/or mouse. If the object we want to deal with is not active, we click on or tab to it to make it active. If we don't see the object, we don't try to deal with it. If something takes a little too long to react to our manipulation, we wait for it to be ready. If it takes way too long, we report it as an anomaly or defect.

Essentially, a manual test case is an abstraction. No matter how complete, it describes an abstract action that is filtered through the mind and fingertips of the manual tester. Open a file, save a record, enter a password: all of those are abstract ideas that need to be translated. The human tester reads the step in the test procedure and translates the abstract idea to the metaphor on the screen using the mouse and keyboard.

In this script, you see a logical translation of those steps. The first line is identifying the window we want to make active, to interact with. The second line details the control we want to deal with, in this case a specific text box. We type a string into that (the "edit_set") and then press the Tab key to move to the next control. Step-by-step, we deal with a window, a control, an action. The data is built right into the script.

So where is the problem? The script is a little ugly, but programming languages often are. We don't expect them to read as if they were to be awarded a Pulitzer Prize in literature. As long as it drives the test to do the right thing, does it matter if it's ugly? What if it does not work like a human would, however?

A recorded script is completely, 100 percent literal. It tries to do exactly what the tester did—and nothing else. That is really the crux of the problem; it

models the human tester completely wrong. Think about what a capture/replay tool is actually saying about the tester it tries to model. The tester is just a monkey who mindlessly does what the manual test case tells them to do. Click here, type there.

But modeling a tester as a repeatable monkey is not valid. A manual tester—at least one that knows what they are doing—adds important elements to that abstract list of steps we call a test procedure. We can narrow it down to two important characteristics added by the human tester to every line of any test: context and reasonableness.

Regarding context, the tester can see and understand what is going on with the workstation. A recorded script cannot add context other than in a really limited way. Look back at the script in Figure 6–2. It has the tester tab from the ID text field (edt_MR Number) to the password text field (edt_Password). If the tab order had changed, a human would see that and tab again, or pick up the mouse and move to the right place with a single click. The script expects the password text field to forever be one tab after the ID text field. Change kills automation when relying on capture/replay. When a failure occurs, it is often signaled by something out of context. A human being is constantly scanning the entire screen to understand the current context. If something incorrect happens—something out of context—the human sees it, evaluates it, and makes a decision. Is it an anomaly that we need to document but then we can continue on? Is it an unrelated failure that we must stop for? The automation tool has no such contextual capability.

If the scripter puts in a check for a particular thing, and that thing is incorrect, the tool will find it. But nothing else will be found. If the script tries to do something—say, type in the password text field—and it does not find the field, it can report in the log that a control was not found. But that test has just failed, often for a superficial nit that a human could have dealt with gracefully.

The other characteristic added by a human is reasonableness. It is clear that there has to be some kind of timing to an automated script. If the script is told to do something to a control that is not currently visible, it will wait for a short amount of time (typically 3 seconds). If the control does not show in that time, boom, the test just failed! Suppose it is a control on a web page that is slow loading? Fail! Suppose it is a control that is out of view due to scrolling? Fail (with most tools). Suppose the developer changed the tag on the control? Fail. A human can sit and look at the screen. It takes 4 seconds rather than 3 seconds? We'll wait—and we might just note in the test log that the control took a long time showing up. Not on the screen? A tester will scroll it. Renamed? A tester will find it.

The flip side of reasonableness is that human beings will often see when there is an issue; for example, when the screen does not render correctly but the automation tool misses it. Here is a funny story from Jamie's early automation career. Jamie was to the point that he was automating the testing of complete dialogs in a Windows 3.1 application. He must have bragged too much because one of the developers decided to teach him a lesson. One small, rarely used modal dialog was modified such that the foreground colors of the dialog were changed to match the background colors. The automation never noticed it—the script testing the dialog worked flawlessly as it identified objects by their listed properties, one of which was *not* color. As a tester, it was Jamie's duty to at least run a sanity test on the dialog. As an automator, he didn't. They shipped the code that way. Support said they got very quizzical questions as to why there was a completely empty dialog in the application. Oops! The truth is that some modern automation tools solve some of these problems. Others don't. There is no capture/replay tool that solves every problem; there are no tools that can always add context and reasonableness except through programming.

Error recovery is virtually always a problem with capture/replay. Early tools had no ability to recover from an error; many modern tools have a limited ability on their own. So, assume that we do have a failure in a test. A human discerns there was an error, gracefully shuts down the application, restarts it, and moves on to the next test. What does the recorded automation script do in case of a failure? The early ones mainly just stopped. Most long-time automators wish they had a nickel for every morning they came in and the suite was stuck on the second test and had not moved all night. Some of the modern tools can, in limited cases, shut down the system and continue on to the next test. Sometimes... But suppose a dialog box pops up that was unexpected? We'll see you Monday morning.

6.2.3.2 Evolving from Capture/Replay

The sad but true fact is that change is the cause of most capture/replay failures. Jamie once had an executive rail at him because the automation was always broken. Every time they ran the scripts, the tests failed because the developers had made changes to the system under test that the automation tool could not resolve. (While we were writing this book, Rex had a number of programmers and testers make the exact same complaint about tests created with the tools QuickTest Pro and Test Complete.) Jamie told the executive that he could fix the problem, easy as pie. When the executive asked how, Jamie told him to have the

development team stop making changes. No changes, no failures. As you might expect, he did not take Jamie's advice...

So, the developer changes the order that events are handled. Boom, automation just failed. A human tester: no problem.

A human being sees a control and identifies it by its associated text, its location, or its context. A tool identifies an object by its location, or its associated tag, or by its index of like fields on the screen (from the top-left to bottom-right) or possibly by an internal identifier. If the way the control is identified changes at all, the tool likely does not find the control. The control was moved a few pixels? If location was the way the automator identified the control, boom, automation just failed. A human tester tends not to have the same problem.

Timing changes. Boom, automation is likely to fail. A human has no such problem. Dynamic content where controls may be enabled/disabled or made visible/invisible based on business rules that are unknown until runtime. Boom, automation is likely to fail sometimes. The human tester simply evaluates what is showing at the time and decides on the fly whether it is correct or not.

System context change? Suppose the recorded test saves a file. The next time the test is run, unless the file was physically removed, when the file save occurs, we are likely to get an extra dialog box popping up: "Do you want to overwrite the file?" Boom, the automation just failed. A human would simply click Yes to clear the message and then move on. The more clever the programmers are, putting up reminder messages or asynchronous warning messages, the more it fouls up the automation. Already created that record in the database? Sorry, you can't create it again.

Frankly, good automators using good processes can minimize these kinds of problems. Working hand in hand with the developers can minimize some. Modern automation tools can minimize some. But, even with the best of everything, you still have testing that just barely limps along. The testing is brittle, just waiting for the next pebble to trip over.

And scalability—the ability to run large numbers of tests without much added cost—is the ultimate capture/replay automation project killer! When the automators have to spend all of their time repairing existing scripts rather than creating new ones, the automation project is already dying.

Let's work through a theoretical situation, one that every automator who has made the jump from capture/replay to the next step has endured.

You have built a thousand test cases using capture/replay. Each one of those test cases at one time or another has to open a file. Each one of them recorded

the same sequence: pulling down the File menu and clicking the Open File menu item.

You get Wednesday's build and kick off the automation. Each automated test case in turn fails. You analyze the problem and there you find, for no particular reason, that the developer has changed the menu item from Open File to Open. Okay, you grab a cup of coffee and start changing every one of your scripts. Simple fix, really. Open each one up, find every place it says Open File, remove the word *File*. If you are really smart, you might do a universal change using search-and-replace or a GREP-like tool; watch that, though, because Open File may show up in a variety of places, some of which did not change. Work all night, get all 1,000 edited, kick them off, find the 78 you edited incorrectly, fix those, and by Friday morning all is well with the world. Of course, you did just totally waste two days...

In Monday's build, the developer decided that change just wasn't elegant, so he changed it back to Open File. You slowly count to 10 in three languages under your breath to avoid saying something to the programmer you cannot take back.

Scalability is a critical problem for the capture/replay architecture. As soon as you get a non-trivial number of test scripts, you get a non-trivial amount of changes that will kill your productivity. The smallest change can—and will—kill your scripts. You can minimize the changes by getting development to stop making changes for spurious reasons, but change is going to come and it is going to kill your scripts, your productivity, and hence your value.

6.2.3.3 The Simple Framework Architecture

Every developer can easily see the solution to this problem. Two generations ago, when spaghetti code was the norm, programmers came up with the idea of decomposition, building callable subroutines for when they do a task multiple times.

As noted earlier, the automation tool the tester is using has—at its heart—a programming language. It should not be a surprise that programming is the solution to these problems.

For the programmer change noted earlier, you can create a function called `OpenFile()` and pass in the filename you want to open as a parameter. You could even just put the recorded code into the function if you wanted. In each one of your 1,000 scripts, replace the recorded code with the function call, passing in the correct filename.

Oops. You get all this done and the developer has [another] change of heart. You get the new build, every test case fails. Ah, but now you go in and change the body of the function itself, recompile all of the scripts, and voila! They all run. Elapsed time: maybe 10 minutes.

Scalability is an important key to successful automation. We need to run hundreds if not thousands of automated tests to recoup our fixed automation costs, much less get positive value. If the automation team cannot reliably run lots of tests, automation will never add a positive value.

Notice now, however, that this is no longer a complete capture/replay architecture. The architecture is now partially recorded and partially programmed. And there are a lot more failure points than just the Open [File]. We could create lots of different functions for other places liable to change. And, come to think of it, we could do more than just open a file using recorded strokes. As long as we are programming a function, we can make it elegant. Perhaps add error handling with meaningful error messages. If the file is stored on a drive that is not mapped, we can add automatic drive mapping inside the function. If the file is large or remote, we can allow more time for it to open without letting it fail. If it takes too long to open but does finally open, we can put a warning message in the log without failing the test. If something unexpected happens, we can take a snapshot of the screen at the failure point so we have an image for the defect record. We can put multiple tasks in a single function, giving us aggregate functionality. Rather than separate keystrokes, we could have a LogIn() function that brings up a dialog, types in the user ID and password, presses the go button, and checks the results.

The automator is limited only by imagination. The more often a function is going to be used, the more value there is in making it elegant.

This leads to what we call the Simple Framework architecture. Other people have other names, and there does not appear to be any standard name used yet. The architecture is defined by decomposing various tasks into callable functions and adding a variety of helper functions that can be called (e.g., logging functions, error handling functions, etc.).

We said earlier that we would differentiate between the terms *framework* and *architecture*. That becomes a little cloudier when the architecture is named “Simple Framework.” Sorry about that.

The architecture is the conceptual or guiding idea that is used in building the framework. Perhaps this metaphor might help. Consider the architecture to be an automobile. It has four wheels, two or four doors, seats, and an engine. There are many different versions of automobiles, including Saab, Toyota,

Chevy, Ford, Dodge, and so on. Each of them is seen as an automobile (as compared to a truck or an airplane, say).

We build an instance of an architecture, calling it a framework. We might build it with a specific tool, building special functionality to make up for any shortcomings that tool might have. We may add special logging for this particular project, special error handling for that. There are several open-source frameworks available that fit certain architectures (e.g., data-driven frameworks, keyword-driven frameworks).

This particular architecture we have named the Simple Framework. The specific details of how you implement it are up to you and your organization. Those decisions should be based on need, skill set, and always—ALWAYS—with an eye toward adding long-term value to the automated testing.

Functionality that is used a lot gets programmed with functions. The more likely a function is to fail, the more time we spend carefully programming error handling and specialized logging. Some stuff that is rarely done might still be recorded using the tool's facilities. A script may be partially recorded, partially scripted. We might add functionality outside the tool; Jamie likes to add custom-written DLLs to integrate more complex functions into the framework.

In the capture/replay architecture, we could allow anyone with any skill set to record the test scripts. Note that now we need one or more specialists: programming testers that many people just call automators. Without programming skills, the framework does not get built. Without excellent software engineering skills, a framework may get built that is just as failure prone and brittle as the capture/replay architecture was. Because, in the final analysis, we are creating a long-term project: building a software application we call a framework.

There are still some risks that we have now that we must consider. Scalability is better than the capture/replay framework, but it's still not great; for each test case, we still have a separate script that must be executed. That may mean thousands and thousands of physical artifacts (scripts) that must be managed. We have seen automation frameworks where the line-of-code count is higher than that of the system under test.

In addition, test data is still directly encoded in each script. That is a problem when we want to run a test that covers the same area with different data.

And we must ask the question, Who is going to write the tests? Too often, we have seen where an organization refuses to hire testers who are not also programmers. They insist that every tester must be able to also write automation code.

Frankly, we think this is a huge mistake. A tester may have some programming skills, or they may not. Are you going to fire every tester in your organization who came from support? All the domain experts who don't know anything about programming? We look at the skill sets of tester and automator as potentially overlapping but not necessarily the same. Not every tester wants to be a programmer—that may even be why they are testers. Testing is much more about risk than it is about programming. If all of our testers are consumed with worrying about the automation architecture, when are they going to be able to think about the risk management tasks that are their real value add?

We believe the best organizational design is to have a test subgroup made up of automation specialists. This would include, like any development team, both designers and programmers (or in a small group, it might be the same person filling both roles). This automation group provides a service to the testers, negotiating on the specific tests that will get automated. Automation is done purposely, with an eye toward adding long-term value. Each project team may have its own automation team, but it is generally our experience that a centralized team, shared between projects, is much more cost effective.

We have solved some of our automation problems with our simple framework architecture, but we are not done yet. We still have some automation risks that we might want to mitigate.

6.2.3.4 The Data-Driven Architecture

The number of scripts in our simple framework architecture that we have to deal with is problematic. As we mentioned earlier, we are going to need a lot of testing to help recover our fixed costs, much less our variable costs. More test cases, more scripts. But there is more overhead (i.e., variable costs) the more scripts we have. And the really annoying thing is that so many of our test cases tend to do the same things, just using different data.

This is the situation that tends to drive automators from the simple framework architecture to the data-driven architecture.

Consider the following scenario. We are testing a critical GUI screen with 25 separate fields. There are a lot of different scenarios that we want to test. We also need to test the error handling to make sure we are not entering bad data into the database. Manual testing this is likely to be ugly, mind-numbing, brain-deadening, soul-sucking testing of the worst sort. Enter all the values. Click OK. Make sure it is accepted. Go back. Enter all the data. Click OK. Go back. Repeat until we want to find a bridge to jump off. This is the exact reason automation

was invented, right? But we could easily have 100 to 200 different scripts, one for each different (but similar) test case. That is a lot of potential maintenance.

To automate this in our simple framework architecture, we create a script that first gets us to the right position to start entering data. Then we sequentially fill each field with a value. After all are filled, click the OK button (or whatever action causes the system to evaluate the inputs). If it is a negative test, we expect a specific error message. If it is a positive test, we expect to get ... somewhere, defined by the system we are testing. Each script looks substantially the same except for the data.

This is where a new architecture evolved. Most people call it data-driven testing, and Jamie invented it. To be fair, so did just about every single automator in the mid-1990s who had to deal with this kind of scenario. In Jamie's case, he realized that he could parameterize the data and put it into a spreadsheet, one column per data input. Later he used a database; it does not really matter where you put the data as long as you can access it programmatically. Some automators prefer flat file formats like comma-separated value (CSV). One column per data input, and then we might add one or more extra columns for the expected result or error message. Each row of data represents a single test. We simply create a single script and build into it a mechanism to go get the appropriate row of data. Now, that single script (built so it's almost identical to all of our framework scripts except for the ability to pick data from a data store) represents any number of actual tests. Have one dozen, two dozen, a hundred rows of data, it doesn't matter. It is still only one script.

Want a new test? Add a new row of data to the data store. Assuming you built the framework correctly, it will pick up the number of tests dynamically, so there are no other changes needed. Next time the automation runs, the new test is automatically picked up and run.

Remember that earlier we said that scalability is an important key to success. Now, to thoroughly test a single screen, we can conceptually have one script and one data store. Compare that to the possibility of 100 to 200 or more scripts just to test that GUI screen.

So now we have a data-driven architecture. Notice that nothing precludes us from having a framework script—or 100—that are not data driven. We might even have a mostly recorded script or two for things that don't need to be tested repetitively. It takes an automator to build in the ability to pick up data from the data store, to parameterize the functions. Or perhaps the tool might have that ability built in. We might also add some more error handling, better logging, etc.

Jamie has a basic rule of thumb when dealing with automation. Have the automators hide as much as possible of the complexity inside the automation so that testers do not have to worry about it. We want the testers to be concerned about risk, about test analysis and design, and about finding failures. We do not want them worrying how the automation works. Need a completely new scenario? The tester needs to give the automator enough information that they can script it—a good solid manual test procedure is optimal. If it is something they need to test a lot, say so. After that, want a new test, same scenario? Add a row of data to the data store and voila...

At this point, the number of tests is no longer proportional to the number of scripts. And, as Martha Stewart used to say, “That is good.” Scalability of maintenance becomes nonlinear where repairing one script may fix dozens or hundreds of tests.

But we are not yet done. Perhaps we can get rid of scripts all together.

6.2.3.5 The Keyword-Driven Architecture

Let’s think about a perfect testing world for a second. A tester, sitting on a pillow at home (we said perfect, right?) comes up with the perfect test scenario. She waves her magic wand and, presto chango, the test comes into being. It knows how to run itself. It knows how to report on itself. It can run in conjunction with any other set of tests or it can run alone. It has no technology associated with it, no script to break. If the system changes, it automatically morphs to “do the right thing.”

Okay, the magic wand may be a little bit difficult to achieve. But the rest of it might just be doable—kind of.

We are going to talk about what is now known as keyword-driven (or sometimes action-word) testing. Keyword-driven testing has been described as a way to use a metalanguage that allows a tester to directly automate without knowing anything about programming. But, if the *meta* thing bothers you, don’t worry about it. We’ll sneak up on it.

Let’s forget about automation for a second. Instead, let’s just look at something we all have seen. Table 6–1 contains a partial manual test procedure—not a full-blown IEEE 829 test procedure specification but a nice minimalist test procedure.

Now, let’s think what we are really seeing in this table. A test procedure step can often be described in three columns. The first column has an abstract task that we want to perform to the system under test—abstract in that it does not

tell you how to do it; it is really a placeholder for the knowledge and skill of the manual tester. The tester knows (we hope) how to start the system, how to log in, how to create a record, how to edit a record. That's why we pay testers rather than train monkeys, right?

Table 6–1 A minimal manual test procedure

Task to perform	Data to use	Expected Result
Start system under test	C:\...\SUT.exe	Starts up correctly
Log in	User ID / Password	Logs in correctly
Create a new record	Pointer to spreadsheet row of data	Key to new record
Edit a record	Record key: a pointer to spreadsheet row of data	Expect change notification

The second column is not abstract; it is very tangible. This is the exact data that we will be using. ISTQB says that this data is the test case, along with the expected result in column three. But, note that column three is also kind of abstract. Start up correctly, log in correctly, record created correctly (key returned), change notification. Again, we are expecting the tester to know what the right thing is and how to determine it.

Remember the discussion we had earlier about context and reasonableness? As shown above, a manual test procedure—at least columns one and three—is just an abstract shell to which a manual tester pours in context and reasonableness when they run the test. Certainly column two is not abstract; that is the concrete data of the test case. With a good tester, we usually do not have to go into excruciating detail for columns one and three; they know the context and what is reasonable for their domain.

Almost every manual test procedure we have ever written kind of looks like this or at least could be written like this. Now imagine that you already have a framework with functions for common items like starting up the system being tested, logging in, creating a new record, and so on. Each framework function has been programmed to contain both context and reasonableness. A script is merely a stylized way to string those executable functions together. So we should ask, “Do we really need a script?”

As you can guess, the answer is no. The script is there for the benefit of the tool, not the tester. If we had a way to make it easy for a tester (with no programming experience needed) to list the tasks they wanted to do in the order they

wanted, and to pick up the data they wanted to use for those tasks, we could figure out a way to scan through them and execute them without a formal script.

This is what a keyword language is. A metalanguage (in this case, *meta* means high level) that has, as a grammar, the tasks that a tester wants to execute. It likely does not have the normal structures (loops, conditionals, etc.) that a standard, procedural programming language has—although in some cases those have been built in. It is actually a lot more like SQL, a descriptive language rather than a procedural language.

Here you see some framework functions that could exist from the simple framework architecture automation already existing (or they could be built completely from the ground up if we are just starting):

- StartApp(str Appname)
- Login(str UserID, str Pwd)
- CreateNewRec(ptr DataRow)
- EditRec(int RecNum, ptr DataRow)
- CloseApp(str Appname)

The actual keywords here are <StartApp>, <Login>, <CreateNewRec>, <EditRec>, and <CloseApp>. Notice the keywords are selected to have some kind of domain-inspired meaning to the testers who will be using them. In the parentheses, you see the data parameters that must be passed in. It still looks like a programming language, right? Well, we need to do a little more to make this user friendly.

The reason this entire keyword mechanism exists is to make it easy for non-programming testers to directly build executable test cases. The easier we make the metalanguage to use, the lower we can drive our variable costs, like training and support.

6.2.4 Creating Keyword-Driven Tables

Learning objectives

TTA-6.2.4 (K3) Create a keyword table based on a given business process.

In our experience, the best way to create a keyword grammar is to mine the manual test cases that already exist. The abstract tasks that will eventually end up as keywords are already likely defined there. The scope of the keywords' actual functionality will usually come from the knowledge of the actual testers who run

those tests. This will ensure that the testers who are going to be tasked with building keyword-driven tests will understand intuitively how to use them.

The actual granularity of the keywords should be debated by the team designing the architecture; that is, how much functionality should a keyword actually entail? There are no absolute answers to the question, but different levels of granularity will require different technical solutions.

For example, at the lowest granularity, you could have keywords that say ClickButton() or TypeEdit(). This would clearly be tied very closely to the interface, and would therefore be problematic when the interface changes. However, there have been keyword tools that consisted of such low-level granularity. Consider how these might be used to automate the login process (see Table 6–2).

Table 6–2 Low-granularity keywords

Task	Screen	Control	Data	Expected
TypeEdit	PwdDialog	UserNameBox	“Samuel Clemons”	Accepted
TypeEdit	PwdDialog	PasswordBox	xxxxxxxxxx	Accepted
ClickButton	PwdDialog	Login		Dialog closes, Main window active

This scheme would require more columns than we discussed in our previous example because we would be required to identify specific screen objects. Any changes to the interface would likely break the keyword scripts (although to be fair, there are ways of dealing with that by using GUI maps or other indirection schemes.) While this gives ultimate control to the tester, it does not really save them time or allow the abstraction we see as a benefit to the architecture.

Consider how, by setting our granularity level a bit higher, we can make it easier for the testers to use the keywords without knowing anything about the actual interface (i.e., using more abstraction). We could conceivably have a Login() keyword that takes two arguments as seen in Table 6–3.

Table 6–3 Higher-granularity keywords

Task	Data	Expected
Login	“Samuel Clemons”, “xxxxxxxxxx”	Logged in

Note that the data passed in could be indirectly referenced by passing in pointers to data stores.

Using higher granularity may make it easier to abstract certain tasks, but it can certainly be taken too far. For example, Jamie was involved in a keyword project conversion once where the starting point was a simple framework-based architecture. To avoid losing the existing automation during the conversion, he created a keyword as shown in Table 6–4.

Table 6–4 A step too far

Task	Data	Expected
ExecuteScript	F:\xxx\xxx\xxx\Script_B4	Pass

This keyword does exactly what you would expect: runs the entire existing script found out on the server. When Jamie checked back with the group 11 months later, they had not graduated to actual keywords yet. It was working, so they did not want to invest further resources.

One thing that can be done is allow aggregation of keywords. Suppose we have a number of tasks involved in creating a record in a database. At times we might want to perform each of these tasks discretely, so we have a different keyword for each task. The following tasks might be covered:

- CreateNewRecord()
- AddPersonalInfo()
- AddBusinessInfo()
- AddShippingInfo()
- SaveExistingRecord()

While we may have started with a single high-granularity keyword that encapsulated all of these, it might make business sense to allow each to be a separate keyword. However, we might still want to use them all together at once. Rather than create a whole new keyword, some implementations allow a new keyword to be created through aggregation. A new keyword, CreateAndPopulateRecord(), could be defined as the combination of the previous five keywords. This obviously requires some intelligent finessing of the inputted data, but it is a reasonable way to allow new keywords to be created.

6.2.4.1 Building an Intelligent Front End

Whatever the level of granularity the keywords are going to have, in our mind the most important design choice to make this architecture easily usable is to have an intelligent front end that can lead a tester through the process of

building the test. It should have drop-down lists that are contextually loaded with keywords so the user does not need to remember the keyword grammar. Such a front end can be built in Excel (on the low end) or in just about any rapid application development (RAD) language that the automator is comfortable with (Jamie historically tends to use Delphi) on the high end. It is our belief that the better we build the front end, the more we can expect to gain from using it long term.

Consider one way such a front end might work. We start creating a new test with a screen that shows three columns and rows that are all blank. Logically, there are only a few things a tester could do; the most logical step would be to start the system under test. So, the first column of the blank test would have a drop-down list that would include the keyword <StartApp>. If the user clicks on <StartApp>, the front end knows that it takes one argument and therefore prompts the tester to enter that in the second column. The drop-down list in the second column would include all of the applications for which there are keywords available. The list is loaded dynamically as soon as the first column is selected (i.e., <StartApp> is chosen). The existing StartApp() function in the simple framework already knows how to check to make sure the application started correctly, so column three is not strictly needed here.

Moving on to the next keyword (the second row of the test), there are only a small number of logical steps that a tester might take next, so the drop-down list in the first column would automatically load keywords representing those steps. In other words, the next keyword drop-down list is populated with only those keywords that logically could be called based on where the previous keyword left us (assuming that the execution actually succeeded). The keyword <Login> would be one of the possible tasks, so the tester selects that. <Login> takes two parameters, so the front end prompts for the user ID and password to use. The third column drop-down might have all known valid and nonvalid results that the framework function may return. If the expectation is that the login should work, that would be chosen. If this is a negative test, a specific expected error message could be selected.

Each step of the way, the tester is guided and helped by the front end. We do not assume that the tester knows the keywords, nor do we assume that they will remember the argument number or types. We do assume that the tester knows the domain being tested, however. The keywords that are loaded in the column one drop-down list at any given time should be a subset of all the keywords, where the starting point is the ending point of the previous keyword. Or, the drop-down may show a tree of all available keywords possible in a hierarchical

structure. The important point we are trying to make is that the more help this front-end tool can give to the test creator, the less training needed and the higher the productivity of the testers.

6.2.4.2 Keywords and the Virtual Machine

This scheme assumes that a keyword always executes successfully to the end so that it leaves the application being tested in the expected state. Of course, any tester can tell you that is not likely to always be the case. Errors might occur or the GUI could be changed. This is where the simple framework comes in. If there is any failure during the execution of the keyword, or if the keyword execution does not drive the system to the expected state, the framework must document the occurrence in the log, clean up the environment (including backing out and perhaps shutting down the application being tested), and moving the automation suite to the next test to be run.

This recovery is transparent to the tester. When correctly built, the keyword architecture allows the testers using it to make the assumption that every test will pass, thus simplifying the testing from the viewpoint of the tester. After all, that is exactly how manual tests are written; each test is expected to pass and, if it doesn't, the manual tester will be able to handle the cleanup and logging tasks.

We might not have figured out the hypothetical magic wand mentioned earlier yet, but the more intelligence automators can add to the framework, the closer we get to it.

If we assume that we can build an intelligent front end to help guide the user, then what we need are keywords. Each keyword should be modeled on a physical task that a tester would be expected to perform, as we had discussed for our manual test procedure. The tester arranges the keywords in the order of execution, exactly the way they did for the manual test procedure. Instead of having a manual tester supply the context and reasonableness during execution, however, the automator supplies it by programming a framework function to do the task. The automator builds into the keyword function the data pickup, error handling, synchronization, expected result handling, and anything else that a manual tester would have handled.

The automator also must build a virtual machine to execute the keywords. Reading the definition for keyword testing in the ISTQB glossary, it references the word *script* three times, calling this a scripting technique run by a control script. We believe this definition is too limiting. There is absolutely no reason that we have to use scripts or scripting techniques to build a successful key-

word-driven tool. We are going to simply call this process a virtual machine. A given automation framework may use scripts, but the tools that Jamie has built have not always used scripts.

This virtual machine will handle all of the logging tasks, exception handling, failed test handling, and execution of the keywords. Essentially, this virtual machine is the framework. Had we started with a simple framework architecture, much of the complexity needed for the keyword architecture is already there. When started, the framework likely sets up the environment, initializes the log, picks up the first test to be run, picks up the first keyword, executes it, picks up the next keyword, and so on. At any time, a failure causes the virtual machine to back out of the current test, log the error (possibly including taking and storing a snapshot of the screen), clean up the environment, and prepare to run the next test case.

As part of the front end, there also should be a business process by which a tester can request a new keyword to perform a certain task. Suppose we were automating MS Word testing. A tester might ask for a keyword that allows them to create a table in a document, with row and column count as parameters. Once the tester asks for it, the automator would make the keyword available so that testers could immediately start using it in their test cases.

Note that we have complete separation of the keyword language, which is abstract and descriptive, and the virtual machine, which has all of the actual execution facilities. A new keyword can be created and used weeks before the keyword execution functionality is built into the virtual machine. Certainly the automator would need to create the functionality behind the keyword to make it executable before the tests are to be run.

This ability for functional testers to create keywords well before the functionality providing for execution of them is written fits well into the model we are trying to describe here. After all, manual test cases can be created well before the code is available to run them, following the ISTQB processes of analysis, design, and implementation. All of this can be as complex as needed by the organization. Note that we are discussing building a front end, a virtual execution environment, exception handler, and all of the other bells and whistles: all of this work means the automators must be good developers. This is a real software system that we are talking about and it likely will take a small team of automators/programmers to build it.

6.2.4.3 Benefits of the Keyword Architecture

Consider the implications of this architecture. We have now effectively built a truly scalable test system. Suppose you have five automators building keyword systems for several different projects in an organization. There are some serious costs there. But notice that any number of projects with any number of testers can be supported; 10 or 20 or 100 testers could all use the keyword systems. There is no limit to the number of testers, nor is there a limit to the number of tests. Whether we have 1,000 or 100,000 test cases, we still need the same number of automators. Compare that to when we were using scripts and there was a finite number of scripts that a single automator could support.

This architecture also allows us to break away from the GUI. Want to test 3,270 or 5,250 dumb terminal emulators? UNIX or AIX interfaces? It takes some intelligent programming on the part of the automator, but the sky really is the limit.

Reviewing where we are now:

- Domain experts (i.e., test analysts) describe the keywords needed and build the test cases using them.
- Automation experts write the automation code to back the keywords and a mechanism for putting them in order easily (the front end).
- An unlimited number of testers can be supported by relatively few automators.
- Test creation is essentially point and click from the front end.

Incidentally, for most of the keyword systems that Jamie has built, he has included the execution module in the front end. This allows anyone to graphically choose what test suites to run and where and when to run them using the same tool as the tests were built in. Some of these also had log browsers built in to provide the ability to walk through the results.

There are several benefits to the keyword architecture that we have not yet addressed.

Throughout this entire discussion of automation, we have stressed what a problem change is. Anytime we are dealing with scripts and the underlying system under test changes, the scripts stop working and they must be repaired. That means the more tests we have, the more effort it takes to fix them. The simple framework architecture and data-driven architectures helped some, but there was still this issue with the scripts.

Manual test procedures, on the other hand, are rarely brittle—at least if we are careful to write them toward the logical end of the detail spectrum rather than as rigid concrete tests with screens, inputs, and outputs all hardcoded into the procedure. As long as we do the same tasks, change does not break them. The manual test procedures—at least when logical rather than concrete—are essentially abstract. Of course, the amount of detail in the test procedure is going to be governed by more than how changeable they need to be. Testing medical, mission-critical, and similar types of software is likely to require extremely detailed procedures.

Consider a common task like opening a file in Microsoft Word. We did it one way in Word for Windows 2000, another way in Word for Windows 2008. Each different version of Word changed the way we opened a file; the interface changed radically during that time period. The differences are mostly subtle—insurmountable to capture/replay, but easy to manual testers. Any automated function would have to change each time. But a manual test procedure that said, “Open a file in Word” would not need to change; the details are abstracted out.

A keyword test is very much like a manual test procedure. It contains abstract functionality, like `<OpenFile>`. Well-designed keyword tests are very resistant to change. The interface can change a lot, but the abstract idea of what a step is doing changes very slowly.

With keyword-based testing, our main assets—the keyword tests themselves—are resistant to change. The (perhaps) thousands of test cases do not need to change when the interface changes because they are abstract. The code backing each keyword will certainly be likely to change over time. And, the automators will need to make those changes. But the main assets, the tests themselves, will likely not need to change. The number of keywords tends to be relatively stable, so the maintenance on them will be relatively small (especially when compared to the amount of testing that is supported).

When you look at the advantages, if an organization is going to start an automation project, keywords are often the way to go. Not every domain is suitable for this kind of automation, but many are.

One disadvantage of capture/replay automation has always been how late in a cycle it can be done. In order to record a script, the system has to be well into system test so a recording can be successfully made. The earlier you record a script, the more likely change will invalidate it. The simple framework architecture mitigates that a little, but by the time the simple framework functions can be completed, plus all of the scripts created, it is again late in the cycle. Data-

driven testing still uses scripts so we still have the same problem (to a lesser degree).

But keywords are abstract. As long as we have a good idea what the workflow is going to be for our system, we can have the keywords defined *before* the system code is written. Our testers could be developing their keyword test cases before the system is delivered, exactly the way they could when they were creating manual test cases and procedures. Conceivably, all keyword automated test cases could be ready on first delivery of the code. Certainly the functionality of the keywords will not be there yet. However, the automators could plausibly have the functionality close to ready based on early releases from development. Then, once the code is delivered, some minor tweaks and the automation might be running—early in system test when it could really make a difference.

Historically, automation has mostly been useful for regression testing during system test (or later). With keyword testing, it is possible to push automation further up in the schedule. It is thinkable that a suite of automated tests could be applied to functionality testing the first time that functionality comes into test if the automators were supplied with early code releases to write their functions. Of course, in Agile life cycles, people use approaches like acceptance test driven development (ATDD) and behavior driven development (BDD) to create automated tests that can be run the first time the functionality is available for testing.

Data-driven techniques can be used to extend the amount of testing also. Rather than inputting specific data into column two, we could input pointers to data stores where multiple data sets are stored. This would effectively give us a data-driven keyword architecture.

There are a number of options available for keyword-driven tools. There are several open-source frameworks available as well as many commercial tools that incorporate keyword technology. Some of these sit on existing automation capture/replay tools, so you can adapt to keywords without losing your investment in frameworks that you have already written. If your organization has the skill set, you might also want to build your own framework. The front end could then be customized for higher productivity and to fit your own needs.

Over his career, Jamie has created a number of these, starting back in the middle 1990s. While they take a lot of time and effort, he has generally found it to be well worth the investment.

Note that the keyword tests themselves, which are really the testware assets for the organization, are completely abstracted from the execution layer—the virtual machine we have discussed. That means that the automators, if they had

to, could completely switch automation tools without having to change any tests. Simply rewrite the framework functions that make up the virtual machine and the organization is back in business. Contrast that to all of the architectures we have discussed where all of the tests (scripts) are intimately tied into the tool they were written in.

It is possible to have several specialized automation tools (vendor and/or open source and/or home brewed) all working to support the abstract layer where the value resides. No longer is the automation team held hostage by a particular tool that may become an orphan because a tool company decided to deprecate it.

6.2.4.4 Creating Keyword-Driven Tables Exercise

Refer to the pseudo code recorded script as seen in Figure 6–3. This is the same code we saw in Figure 6–2.

Devise a keyword grammar that could be used to test this portion of the application. Note that you will need to use your imagination a bit to figure out what the recorded script is doing. The important thing is to look for the underlying business logic while reading the actions.

1. Make "FREDAPP/11" the active window
2. Type "MRE5418" into edit box "edt_MR number"
3. Press the Tab button to move to next control
4. Type "kzisnyixmynxfy" into edit box "edt_Password"
5. Type "VN00417" into edit box "Edit_2"
6. Press the Return key [should trigger popup window]
7. Make "FREDAPP/7" the active window
8. Press the No button [should trigger popup window]
9. Make "FREDAPP/5" the active window
10. Type " BC3456 " into edit box "edt_MR number"
11. Press the Tab button to move to the next control
12. Type "dzctmzgtdzbs" into the edit box "edt_Password"
13. Press the Return key [should trigger popup window]
14. Make "FREDAPP97Msg/5" the active window
15. Press the Yes button

Figure 6–3 Recorded WinRunner script

6.2.4.5 Keyword-Driven Exercise Debrief

The first six lines of this script could be seen as a single action. Note that the following would be a description of a human being (assume a doctor) interacting with the application using the keyboard and mouse:

- The doctor must have started the application earlier, causing a password dialog screen to pop up. So the first thing is to mouse click on that screen, causing it to become active (line 1).
- Since line 2 consists of us typing into an edit box (MR_number), we have to assume it was the default control. So we need to type in the value ("MRE5418")
- In line 3, we tab from the first edit box to the password edit box.
- We type in the password in line 4 ("kzisnyixmynxfy").
- Notice that in line 5 we are setting the value in another edit box. How did we get there? It may have been a mouse click; it may have been automatically handled by a Return press that was handled directly by the control. It really does not matter. The important part is it tells us that we have a third value that needs to be entered. In line 5, we type into Edit_2 (which is actually the domain edit field) the domain value (VN00417.)

The sum result is a single logical action. Since keyword names should be self-documenting, we are going to call this one

<LoginDomain>

It will take three arguments:

User ID, Password, and Domain name

This would be where the intelligent front end would need to come in to help the tester. There would have been a <StartApp> keyword that would have initialized the system, leaving it at the screen needed for <LoginDomain> to be called. After startup, the Task To Do drop-down list would contain <LoginDomain>. When selected, it would automatically pop up three edits so the user would see that three arguments must be passed in when using the keyword. After filling them in, the user would go on to enter the next keyword.

Lines 7 and 8 would be performed by the doctor as follows: After the doctor logs in to the domain, some kind of dialog pops up with a question. Since the script shows the user selected No when it popped up, we can reasonably assume that there are two separate paths we could take (Yes or No).

So the keyword we need is going to take a single Boolean argument. In this case, the question that was asked was, Do you want to prescribe a treatment? A

logical keyword should show that a decision is being made. Therefore, we will call it

<DoTreatment> or perhaps <DoTreatment?>

This keyword will take a single Boolean argument that the front end would likely show as a check box.

The next step would appear to be redundant but was part of the security on the multilevel system. Lines 9 through 13 are the same actions as 1 through 6 on a different window. This is another layer of security allowing the doctor to get into the section of the system that allows prescribing drugs. Notice that we cannot use the same keyword <LoginDomain> because it is a different window that we are logging in to. So we need another keyword:

<LoginSection>

This will take three arguments: user ID, password, and a Boolean value. Note that in this case we are handling the box that pops up after the login by passing in whether we will click Yes or No. In this case, the message is a nag (informational) message that we want to ignore—hence the Yes answer.

Our keyword script (so far) would look like Table 6–5.

Table 6–5 Possible keyword solution

	Keyword	Arg1	Arg2	Arg3
1	LoginDomain	MRE5418	kzisnyixmynxfy	VN00417
2	DoTreatment	No		
3	LoginSection	BC3456	dzctmzgtdzbs	Yes

Finally, notice that we could have conceivably passed four arguments to <LoginDomain>, with the fourth being whether to answer Yes or No to the screen that pops up after we log in to the domain. That would have eliminated the <DoTreatment> keyword. We think this is strictly a matter of taste. We need to balance the amount of things that a single keyword does by the number of keywords we have and the complexity of trying to understand each task.

6.3 Specific Test Tools

While automation is likely to be a very large portion of the tasks for a technical test analyst, there are other tool sets a TTA may be required to use. The following sections describe several different types of tools that may be seen. Although this is not an exhaustive list, it is representative of what we have seen.

6.3.1 Fault Seeding and Fault Injection Tools

Learning objectives

TTA-6.3.1 (K2) Summarize the purpose of tools for 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 is typically done 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 sometimes called mutation testing, where an effort is made to generate all possible mutants.³ NASA uses this technique to help in reliability testing. This technique can only be used when it is practical to generate a large number of fault seeded (or “be-bugged”) variants of a program and then to test all those variants (or “mutants”). This generally requires automation not only of the fault seeding but also of the testing of the variants.

Fault injection is usually about injecting bad data or events at an interface. This has the effect of forcing the system to execute exception handling code. For example, Rex has a tool that allows him to randomly corrupt file contents. The core logic of that tool is shown in Figure 6–4. Similar techniques can be used to bombard the software with a stream of incorrect inputs, failed services from the network or operating system, unavailable memory or hard disk resources, and so forth.

These types of tools are useful to both developers and technical test analysts.

3. There’s a discussion about the use of mutation testing at Google in Chapter 18 in the book *Beautiful Testing*, edited by Tim Riley and Adam Goucher.

ISTQB Glossary

fault seeding tool: A tool for seeding (i.e., intentionally inserting) faults in a component or system.

```
for (i_cnt=0; (i_cnt<i_corrupt); i_cnt++) {  
  
    /* Figure out where we'll corrupt and how. */  
    b_badbyte = (unsigned char) (rand() % 0x100);  
    i_pos = (long) ((double) i_fend * ((double) rand()/(double) RAND_MAX));  
    printf("Corrupting byte %d to be %x now.\n", i_pos, b_badbyte);  
  
    /* Go to the right place in the file. */  
    if (fseek(fp_fcorrupt, i_pos, SEEK_SET) ) {  
        fprintf(stderr, "Seeking position to corrupt failed...exiting.\n");  
        exit(4);  
    }  
  
    /* Corrupt the file by writing random data. */  
    i_wrote = fwrite(&b_badbyte, sizeof(char), 1, fp_fcorrupt);  
    if (i_wrote != 1) {  
        fprintf(stderr, "Corrupting failed...exiting.\n");  
        exit(5);  
    }  
}
```

Figure 6–4 Core loop for a file corrupter utility

6.3.2 Performance Testing and Monitoring Tools

Learning objectives

TTA-6.3.2 (K2) Summarize the main characteristics and implementation issues for performance testing and monitoring tools.

Before performance tools were readily available, Jamie used to simulate performance testing using a technique they called sneaker-net testing. Early Saturday or Sunday morning, a group of people would come into the office to do some testing. The idea was to try to test a server when few (or no) people were on it.

Each person would get set up on a workstation and prepare to connect to the server. Someone would give a countdown, and at go! everyone would press the connection key. The idea was to try to load the server down with as many processes as they could to see what it would do. Sometimes they could cause it

ISTQB Glossary

performance testing tool: A tool to support performance testing 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.

to crash; usually they couldn't. Occasionally, if they could get it to crash once, they could not crash it a second time.

This nonscientific attempt at testing was pretty weak. They could not get meaningful measurements; it was essentially binary: failed | didn't fail. If they did get a failure, it was rarely repeatable, so they often did not understand what it told them.

Luckily we don't have to do this anymore; in today's world, real performance test tools are ubiquitous and relatively cheap compared to even 10 years ago. There are now open-source and leased tools and tools in the cloud, as well as some incredible vendor tools available.

We talked about the testing aspects of performance testing in Chapter 4. In this chapter we want to discuss the tool aspects. For the purposes of this discussion, let's assume that we have done all of the modeling/profiling of our system, evaluated the test and production environments, and are getting ready to get down to actually doing the testing.

Often, the test system is appreciably smaller than the production system. Be aware that you may need to extrapolate all of your findings before they are actually meaningful. Extrapolation has been termed "black magic" by many testers. If the system model you are working from is flawed, extrapolating it to a full-size system will push the results that much further off. On top of that, there could easily be bottlenecks in the system that will only show up when the system is running at higher levels than we might be able to test in a lab. We can check for that on the system we test but have no way of knowing which problems will show up in production, where the hardware is likely different or at least set up differently.

6.3.2.1 Data for Performance Testing

At this point we need to define the data we are going to use, generate the load, and measure the dynamic aspects that we planned on. It is pretty clear that performance testing with poor or insufficient data is not going to be terribly useful; coming up with the right data is a non-trivial task.

We could use production data, but there are a couple of caveats that we need to consider. There are likely laws that are going to prevent us from using certain data. For example, in the United States, testing with any data that is covered by the Health Insurance Portability and Accountability Act (HIPAA) is problematic. Different countries have different privacy laws that might apply; testers must ensure that what they are using for data is not going to get them charged with a crime. There are tools available for data anonymization (also called data scrambling) that can render production data safe to use by changing values to make them still realistic but no longer personally identifiable.

Generally, production data is much bigger than the available data space in the test environment. That raises many questions about how the production data (assuming we are allowed to use it) can be extracted without losing important links or valuable connections between individual data pieces. If we lose the context that surrounds the data, it becomes questionable for testing (since that context might be needed by the system being tested).

Fortunately there are a variety of tools that can be used to generate the data you need for testing. Depending on the capabilities needed, possibilities range from building your own data using an automation tool to building it using really pricey vendor tools. Jamie's suggestion, based on getting lost several times in his career by underestimating the effort it would take to generate enough useful data, is to plan lots of time, effort, and resources on the creation of your data.

There are three distinct types of data that will be required.

There is the *input* data that your virtual users will need to send to the server:

- User credentials (user IDs and passwords): Reusing credentials during the test may invalidate some of the findings (due to caching and other issues). In general, you should have a separate user ID for each virtual user tested.
- Search criteria: Part of exercising the server will undoubtedly include searching for stuff. These searches should be realistic, for obvious reasons. Searches could be by name, address, region, invoices, product codes, and so on. Don't forget wild card searches if they are allowed. You should be familiar with all of the different kind of searches your system can do; model and create data for them.

- Documents: If your system deals with attached documents (including uploading and downloading) then those must be supplied also. Different file types should be tested; once again, if the server is going to do a particular thing in production, it probably should be modeled in test.

Then we have the *target* data. That would include all of the data in the back end that the server is going to process. This is generally where you get into huge data sets. If the data set is too small, some of the testing will not be meaningful (timing and resources needed for searches and sorts, for example). You will likely need to be able to extrapolate all of your findings (good luck with that) if this data set is appreciably smaller than in production.

Jamie's experience in performance testing is that the test data is usually smaller than in production. Rex has seen many times where the amount of data is similar to in production. Clearly, either case is possible. If you have lots of data, great. If not; well, you must do the best you can with what you have. It is important that you document any perceived risks in either case.

Note that the backend data will need to support the input data that we discussed earlier.

You will most certainly need to have a process to roll the data back after a test is completed. Remember that we often will run a number of performance tests so that we can average out the results. If we are not testing with the database in the same condition each time, then the results may not be meaningful. Don't forget to budget in the time it takes to reset the data.

Finally, we have to consider the *runtime* data. Simply getting an acknowledgment from the server that it has finished is probably not sufficient. To tell whether the server is working correctly, you should validate the return values. That means you need to know beforehand what the returned data is supposed to be. You could conceivably compare the return data manually; don't forget to bring your lunch! Clearly this is a spot for comparator tools, which of course, requires you to know what is expected.

6.3.2.2 Building Scripts

One good way to get reference data is to run your performance transactions before the actual performance test begins. You are going to check the transactions before using them in a test, aren't you? Make sure the data is correct, save it off, and then you can use a comparator tool during the actual run. Since we know what kinds of transactions we are going to be testing—we identified them in the modeling step—we now need to script them. For each transaction, we

need to identify the metrics we want to collect if we did not do that while modeling the system. Then the scripting part begins.

Some performance tools have a record module that captures the middleware messages that connect the client to the server and places them in a script while the tester runs the transaction scenario by hand from the client. This is what Jamie has most often seen. More complex and time consuming would be to program the transaction directly. This will entail more complexity and effort, but it may be needed if the system is not available early enough to do recording.

Once we have a script, we need to parameterize it. When it was recorded, the actual data used were placed in the script. That data needs to be pulled out and a small amount of programming needs to be done so that in each place the script used a constant value, it now picks the data up from the data store.

Once that's done, the script itself must be tested. Try running it by itself and then try running multiple instances of it to make sure the data parameterization was done correctly.

6.3.2.3 Measurement Tools

By the time we get to this point, we are almost ready. However, we still need to set up our measuring tools. Most servers and operating systems have a variety of built-in tools for measuring different aspects of the server performance.

There are two types of metrics we need to think about. As a reminder from Chapter 4, the first metrics are response time and throughput measures, which compute how long it takes for the system to respond to stimuli under different levels of load and how many transactions can be processed in a given unit of time. The second set of metrics deals with resource utilization; how many resources are needed to deliver the response and throughput we need.

Response time generally consists of the amount of time it takes to receive a response after submitting a request. This is often measured by the server (or workstation) that is submitting the requests to the server and is usually performed by the tool submitting the requests. Any other time before the request is submitted and after the response is received is called "think time." Throughput measures are also generally measured by the performance tool by calculating how many requests were submitted where responses were received within a given measure of time.

For the resource usage measurements, most are done on the servers. If we are dealing with a Windows server, then monitoring is relatively simple. *Perfmon* is a Microsoft tool that comes with the server operating system; it allows hundreds of different measurements to be captured.

If you are dealing with Linux or UNIX, there are a bunch of different tools that you might use:

- Pmap: process memory usage
- Mpstat: multiprocessor usage
- Free: memory usage
- Top: dynamic real-time view of process activity on server
- Vmstat: system activity, hardware and system info
- SAR: collects and reports system activity
- Iostat: average CPU load and disk activity

For mainframe testing, there are a variety of built-in or add-on tools that can be used.

Explaining how each of these tools works is, unfortunately, out of scope for this book. However, you can find each one and uncover extensive knowledge by performing a web search.

When dealing with these measurements, remember that we are still testing. That means that it is important to compare expected results against the actual values that are returned.

6.3.2.4 Performing the Testing

So now we are all set. We can start our performance test, right? Well, maybe not quite yet. We need to try everything together in a dress rehearsal; that is, we have to smoke test our performance test. It is our experience that all of the different facets almost never work correctly together on the first try. When we start ramping up a non-trivial load, we often start triggering some failures.

This is a great time to have the technical support people at hand. The network expert, database guru, and server specialist should all be handy. When—as inevitably happens—the system stabs its toe and falls over, they can do immediate troubleshooting to find out why. Often it is just a setting on the server or a tweak to the database that is needed and you can get back to the smoke test. Sometimes, of course, especially early in the process, you find a killer failure that requires extensive work. You need to be prepared because that does happen fairly often.

Once you get it to all run seamlessly, you might want to capture some baselines. What kind of response times are we getting with low load—and what did we expect to get? These will come in handy later when we're ramping up the test for real.

There are some extra things to think about when setting up a performance test.

In real life, people do not enter transaction after transaction without delay. They take some time to think about what they are seeing. This kind of information should have been discussed when we were modeling the system, as well as how many virtual users we wanted to test.

Depending on the kind of testing we are doing, we can ramp up the virtual user count in several different ways:

- The big bang, where we just dump everyone onto the system at once (a spike test scenario)
- Ramp up and down slowly, throwing all of the different transactions into the mix
- Delay some users while using other ones

The way we ramp up is often decided based on the kind of testing we are doing.

Likewise, the duration of the test will also depend on the type of test. We may run it for a fixed amount of time, based on our objectives. Some tests we might want to run until we are out of test data. Other tests we might decide to run until the system fails.

After we shut down the performance test, we still have some work to do. We must go grab all of the measurements that were captured. One good practice is to capture all of the information from the run so we can analyze it later. Inevitably, we forget something on our initial sweep; if we don't save it off, it will be lost forever.

After all this, draw a deep breath, step back, and compare what you found with what you expected. Did you learn what you wanted to? Or in other words, did the results show that the requirements are fulfilled? This is not a silly question. Jamie read a fascinating article in a recent science magazine that discussed how often researchers run an experiment and then don't believe the results they got because it did not fit in with their preconceived mind-set. As testers, we are professional pessimists. We need to learn to identify a passing test—and a failing test.

If the test passed, then you are done. Write the report and go get a tall refreshing beverage; you deserve it. If there are details that weren't captured...well, welcome to the club. Tomorrow is another day.

6.3.2.5 Performance Testing Exercise

Given the efficiency requirements in the HELLOCARMS system requirements document, determine the actual points of measurement and provide a brief description of how you will measure the following:

1. 040-010-050
2. 040-010-060
3. 040-020-010

The results will be discussed in the next section.

6.3.2.6 Performance Testing Exercise Debrief

040-010-050

Credit-worthiness of a customer shall be determined within 10 seconds of request. 98% or higher of all Credit Bureau Mainframe requests shall be completed within 2.5 seconds of the request arriving at the Credit Bureau.

In this case, we are testing time behavior. Note that this requirement is badly formed in that there are two completely different requirements in one. That being said, we should be able to use the same test to measure both.

The first is 2.5 seconds to complete the Credit Bureau request.

Ideally, this measurement would be taken right at the Credit Bureau Mainframe, but that is probably not possible given the location of it. Instead, we would have to instrument the Scoring Mainframe and measure the timing of a transaction request to the Credit Bureau Mainframe against the return of that same transaction. That would not be exact because it does not include transport time, but since we are talking about a rather large time frame (2.5 seconds), it would likely be close enough.

The second is 10 seconds for the determination from the Telephone Banker side.

This measurement could be taken from the client side. Start the timer at the point the Telephone Banker presses the Enter button and stop it at the point the screen informs the banker that it has completed. Note, in this case, that we would infer that the client workstation must be part of the loop, but since it is single threaded (i.e., only doing this one task), we would expect actual client time to be negligible. So, our actual measurement could conceivably be taken from the time the virtual user (VU) sends the transaction to the time the screen is fully loaded. That would allow the performance tool to measure the time.

Clearly, this test would need to be run with different levels of load to make sure there is no degradation at rated load: 2,000 applications per hour in an early release (040-010-110) and later at 4,000 applications per hour (040-010-120). In addition, it would need to be run a fairly long time to get an acceptable data universe to calculate the percentage of transactions that met the requirements.

040-010-060

Archiving the application and all associated information shall not impact the Telephone Banker's workstation for more than .01 seconds.

Again, we are measuring time behavior.

Because the physical archiving of the record is only a part of this test, this measurement would be made by an automation tool running concurrently with the performance tool. Our assumption is that the way the requirement is worded, we want the Telephone Banker ready to take another call within the specified time period. That means the workstation must reset its windows, clear the data, and so on while the archiving is occurring.

We would have a variety of scenarios that run from cancellation by the customer to declined applications to accepted. We would include all three types of loans, both high and low value. These would be run at random by the automation tool while the performance tool loaded down the server with a variety of loans.

The start of the time measurement will depend on the interface of HELLO-CARMS. After a job has completed, the system might be designed to reset itself or the user might be required to signal readiness to start a new job. If the system resets itself, we would start an electronic timer at the first sign of that occurring and stop it when the system gives an indication that it is ready. If the user must initiate the reset by hand, that will be the trigger to start the timer.

040-020-010

Load Database Server to no more than 20% CPU and 25% Resource utilization average rate with peak utilization never more than 80% when handling 4,000 applications per hour.

This requirement is poorly formed. During review, we would hope that we would be able to get clarification on exactly what resources are being discussed when specifying percentages. For the purposes of this exercise, we are going to make the following assumptions:

- 25% resource utilization will be limited to testing for memory and disk usage.
- Peak utilization applies to CPU alone.
- The network has effectively unlimited bandwidth.

This test will be looking at resource utilization, so we would monitor a large number of metrics directly on the server side for all servers:

- Processor utilization on all servers supplying the horsepower
- Available memory
- Available disk space
- Memory pages /second
- Processor queue length
- Context switches per second
- Queue length and time of physical disk accesses
- Network packets received errors
- Network packets outbound errors

The test would be run for an indeterminate time, ramping up slowly and then running at the rated rate (4,000 applications per hour) with occasional forays just above the required rate.

After the performance test had run, we would graph out the metrics that had been captured from the database server. Had the server CPU reached 80 percent at any time, we would have to consider the test failed. Averaging out the entire time the test had been running, we would look for memory, disk, and CPU usage on the database server to make sure that they averaged less than the rated value.

Note that this test points out a shortcoming of performance testing. In production, the database server could easily be servicing other processes beyond HELLOCARMS. These additional duties could easily cause the resources to have higher utilization than allowed under these requirements.

In performance testing, it is always difficult to make sure we are measuring apples to apples and oranges to oranges. We fear without more information, we might just be measuring an entire fruit salad in this test.

6.3.3 Tools for Web Testing

Learning objectives

TTA-6.3.3 (K2) Explain the general purpose of tools used for web-based testing.

Web tools are another common type of test tool.

A frequent use of these tools is to scan a website for broken or missing hyperlinks. Some tools may also provide a graph of the link tree, the size and speed of downloads, hits, and other metrics. Some of these are used before going live; others can be used to run regular checks on a live website to minimize the time that broken links are exposed or to optimize the user experience.

Some tools can do a form of static analysis on the HTML/XML to check for conformance to standards. This validation is often done against W3C specifications, but some tools also might validate a website against Section 508 (the accessibility standard) of the US Rehabilitation Act of 1973.

Spell checking and validation against specific browsers can also be done by some tools.

There are a wide variety of web testing tools that fall into the category of test automation and/or performance tools:

- Selenium: An open-source suite of tools that run in several different browsers across different operating systems.
- Latka: An end-to-end functional testing automation tool implemented in Java. It uses XML syntax to define HTTP/HTTPS requests and a set of validations to ensure that the requests were answered correctly.
- Watij: A Java-based open-source tool that automates functional testing of web applications through a real browser.
- SlimDog: A simple script-based web testing tool based on HttpUnit.
- LoadSim: A Microsoft-supplied tool that simulates loads on Microsoft Exchange servers.
- Sahi: A JavaScript-based record playback tool for browser-based testing.

Of course, not all tools used to test website performance are special-purpose tools. For example, ordinary performance tools can be used to drive load to ascertain if servers can reasonably handle expected load.

Many special-purpose security tools have been designed to search for penetration and other security issues endemic to websites. Some of these can be used

ISTQB Glossary

hyperlink test tool: A tool used to check that no broken hyperlinks are present on a website.

by a developer while writing code; the tool provides real-time feedback to the programmer by automatically detecting risky code. Other tools can provide a complete security assessment of the website before going live.

A variety of test tools are now available for testing client-side capabilities. For example, JavaScript and Ajax are often used on the client to offload some processing from the server; these can be tested in different browsers to ensure optimum user satisfaction.

An important point to remember about many of these web testing tools is that they perform some testing tasks really well but other testing tasks are either not supported or difficult to do. This is fairly common with open-source tools. The designers of the tool are often interested in solving a particular problem and they design the tool accordingly. While this is certainly not true of every open-source tool, an organization that decides to use open-source tools should expect to mix several tools together to create a total solution.

Web tools are used by both test analysts and technical test analysts. They can use these tools at any point in the life cycle once the website to be analyzed exists.

6.3.4 Model-Based Testing Tools

Learning objectives

TTA-6.3.4 (K2) Explain how tools support the concept of model-based testing.

The phrase *model-based testing* can, as a practical matter, mean a number of different things. At a high level, you can say that model-based testing occurs when a model (e.g., a state transition diagram, a business process flow, or the web of screen linkages in a user interface) describes the behavior or structure of a system and that model is used to generate functional or nonfunctional test cases. This process can be supported by tools:

- To generate the tests, either at a logical level or concrete level of detail
- To execute the tests against the system

- To do both of these things, either sequentially (generate then execute) or in parallel (just-in-time generation for execution)

Model-based testing is interesting in that it is a practical, real-world testing approach that grew out of academic research, especially on the subject of state-based systems.

When done with tool support, a nice benefit of the technique is that the tool can generate and execute a huge number of tests, as we'll illustrate with an example. If time is limited, some of the tools have the ability to prune the test set, though the tester will need to be able to indicate to the tool the criteria for selection and rejection of tests. Because of the larger number of tests, and the automatic (in some cases randomized) generation of input values, these model-based test tools will often find bugs that manual testers would miss.

Let's look at an example of a model-based testing tool developed to do functional and nonfunctional testing on a simple mobile application.⁴ The application is called ePRO-LOG, an electronic diary for patients to use for data collection in clinical trials and disease management programs. Depending on the treatment, these diaries include over 100 forms that must be traversed to capture information about a particular treatment, such as injection-site inflammation, allergic reactions, and so forth. Not only were there many forms, but all the forms might have to be supported in a dozen or more languages used in several locales. This led to a large and complex collection of configurations. Each configuration must be adequately tested according to FDA regulations. These regulations call for 100 percent verification of requirements with traceability of tests and their results back to the requirements. The company was also concerned about reliability, translation completeness, functionality of UI, and input error checking.

In addition to addressing these testing issues, the company wanted the following benefits:

- Capture accurate diagrams of the screen flows that had been tested.
- Reduce tedious, error-prone manual screen verification testing.
- Reduce duration and effort associated with the testing.
- Automatically generate tests for any screen flow or translation.

4. This example is a condensed version of the article "Quality Goes Bananas," which Rex cowrote with Daniel Derr and Michael Tyszkiewicz of Arrowhead Electronic Healthcare. Rex thanks Daniel and Mike for their assistance with that article and for the educational experience of building the monkey test tool. The full article can be found at www.rbcus-us.com.

As a first step, we evaluated commercial automated test tools as a potential solution but were unable to find any that would satisfy our requirements. Therefore, a custom tool was built using open-source components. We started with the basic idea of a dumb monkey tool, an unscripted automated test tool that gives input at random. The dumb monkey was implemented in Perl under Cygwin running on a Windows PC. This tool went beyond the typical dumb monkey, though, as it had the ability to check against a model of the system's behavior, encapsulated in machine-readable requirements specifications. Since we were building a tool we called the monkey, as you can imagine elements of humor entered the project, as illustrated by the terminology shown in Table 6–6.

Table 6–6 Terms used for the model-based testing tool

Terms	Meaning
Chef	Testability features added to the application to make the Monkey Chow.
Monkey Chow	Human-readable description of the screen produced by the application in real time.
Eat Monkey	Reads in the Monkey Chow and creates a data structure suitable for the Think Monkey.
Think Monkey	Takes in the data structure from the Eat Monkey and decides what action to take next.
Watch Monkey	Captures screen shots of ePRO-LOG as the monkey operates.
Push Monkey	Interacts with the PDA user interface. The Push Monkey creates custom Windows messages and sends them to the ePRO-LOG application.
Monkey Droppings	Screen shots and human-readable log files produced by the monkey to keep track of where it's been, what it's done, and what it's seen.
Chunky Monkey	The Chunky Monkey encapsulates the Eat Monkey, Think Monkey, Watch Monkey, and Push Monkey and produces the Monkey Droppings.
Presentation Monkey	Transforms Monkey Droppings into graphical flow charts.
dot file	A human-readable data file used by the GraphViz dot application to generate abstract graphs.

The monkey started off working much the same as any dumb monkey tool, sending random inputs to the user interface, forcing a random walk through the diary implemented in the ePRO-LOG application. Because the inputs are random, the tests are diary independent. Three simple test oracles are implemented

at this point without any underlying model: check each form for broken links, missing images, and input validation errors. The tests, once started, run for days and test a huge number of input combinations, thus doing long-term reliability and robustness testing as well. At this point, we are still in the realm of typical dumb monkey test tools.

However, the tool captured all the tests performed and results obtained in human-readable documentation, thus supporting FDA audits if necessary. Since the proper translation of each form had to be checked, the capture of this information simplified that process as well. This is not typical of a dumb monkey. A graphical view of how the elements of the tool all fit together is shown in Figure 6–5.

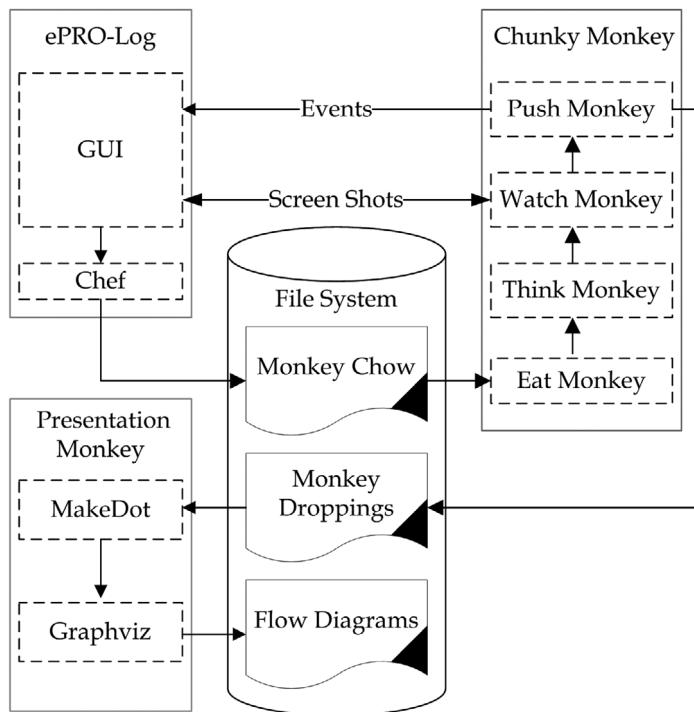


Figure 6–5 ePRO-LOG and the test tool's subsystem interactions

The aspect that transformed the tool from a dumb monkey to a model-based testing tool, however, was the ability to read machine-readable requirements specifications to determine the correct appearance of each screen. These requirements were machine readable to support another element of FDA audit-

ing of the software development (rather than testing) process, so there was no extra work to create them. The tool had a parser that allowed it to read those specifications as it walked through the application. (The product of this parsing process is referred to as Monkey Chow, since it was a main input to the tool.) If the tool found that the wrong screen was displayed, or that the right screen was displayed but some of the information on the screen was wrong, it would log that as a failure. All of the logging was done in a way that it would support FDA audits, which was a tremendous benefit given the effort required to do that manually.

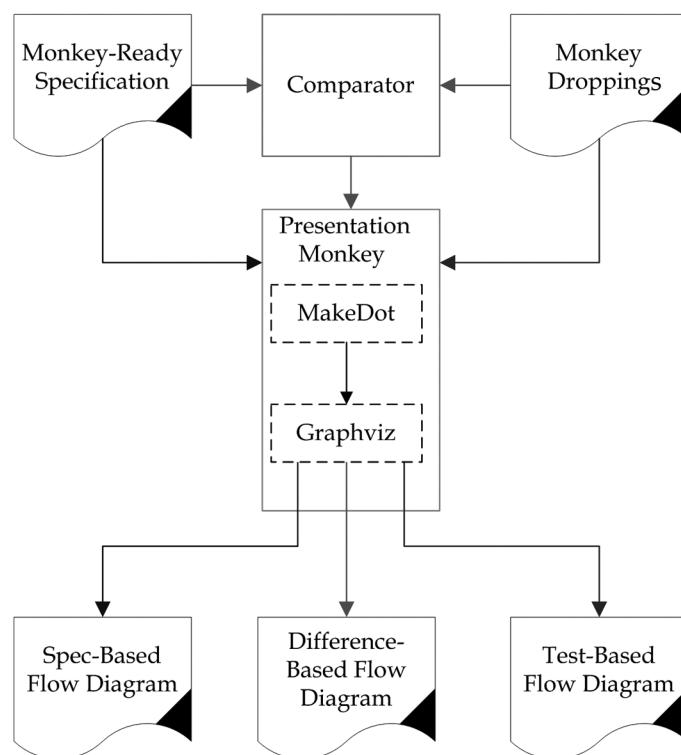


Figure 6–6 Using the requirements as a model

Figure 6–6 shows the model-based element of the tool. The Presentation Monkey produces a screen flow diagram from the Monkey Droppings file as shown previously in Figure 6–5. This diagram shows what screens were observed during monkey testing. It also produces a screen flow diagram using the requirements specifications instead of the Monkey Droppings file. This diagram shows

the expected functional flow of ePRO-LOG. This is fed to a comparator tool that can log any anomalies.

An example is shown in Figure 6–7. The requirements specifications said that the screen flow should go from FormT4 to FormT5 before FormSave, but instead the application went directly from FormT4 to FormSave. In addition, the requirements said the screen flow should go from FormI2 directly to FormSave, but instead the application went from FormI2 to FormI3 before going to FormSave.

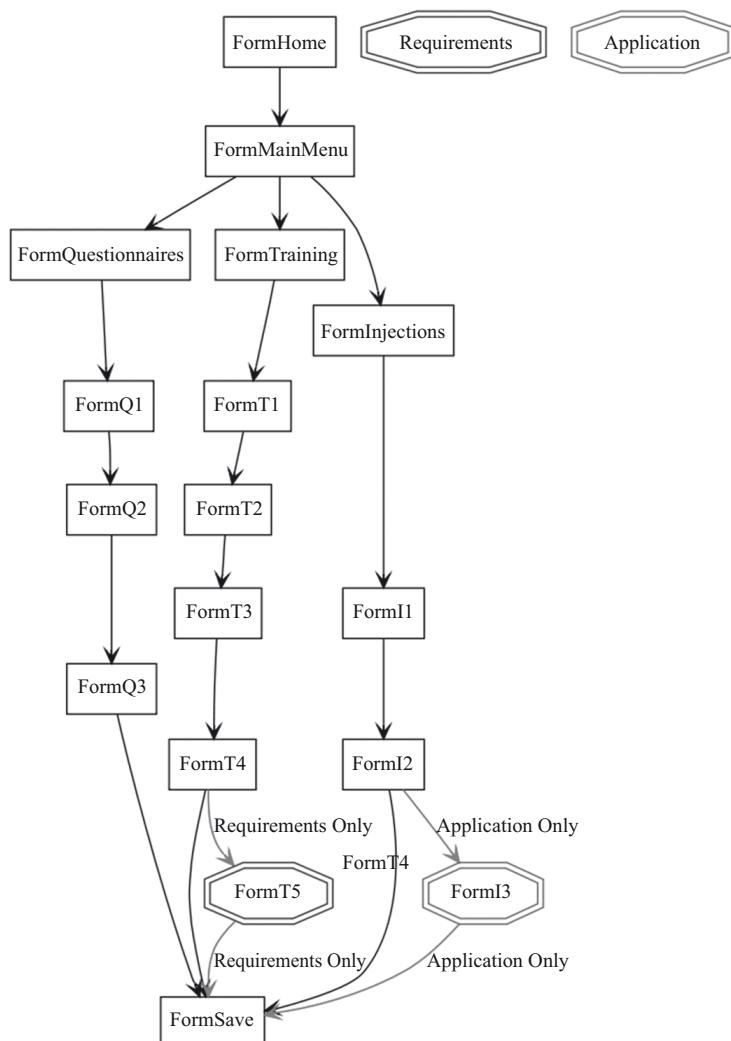


Figure 6–7 An example of the monkey finding two anomalies

6.3.5 Tools to Support Component Testing and Build Process

Learning objectives

TTA-6.3.5 (K2) Outline the purpose of tools used to support component testing and the build process.

While often considered the domain of developers or release engineering teams, in some cases we have seen clients assign build and component test automation responsibilities to technical test analysts. Even if another team owns these tools, you should know how they work because you'll find ways to integrate automated tests into the automated build and component test framework. If you are working in an organization following an Agile life cycle, you will almost certainly encounter these tools. Even if your organization follows a sequential life cycle, you can take advantage of these tools, though you're more likely to have to take a lead role in establishing the tools in that case.

These tools are essential to support a process referred to as continuous integration. Some of the tools are language specific, but a commonality has grown up around these tools such that the same process can be set up and automated for most modern programming languages. Continuous integration is a prevalent practice commonly associated with Agile life cycles. However, Rex has been involved in projects using continuous integration frameworks since 1992, at which time anyone referring to a colleague as an Agile programmer would probably have been inspired by that programmer's ability to juggle.

Continuous integration involves programmers checking in code to the configuration management system once that code has achieved certain entry criteria. At the least, those entry criteria should include passing a set of automated unit tests. The best practice is to require static analysis of the code, followed by a code review meeting that examines the code, the unit tests, the unit test results, the code coverage achieved by the unit tests, and the static analysis results.

The code is checked in, merged with any other changing code as necessary, and integrated with the other code in the relevant code branch. At this point, an automated process occurs periodically—at least once a day but often more frequently—to compile, build, test, and perhaps even deploy a new release. This tight loop of check-in/build/test allows the quick detection and repair of bugs, especially the particularly momentum-killing build-breaking bugs. The

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.

static analyzer: A tool that carries out static analysis.

specifics of the continuous integration tools and process can vary but will include some or all of the following:

- Using a version control tool, often integrated with a continuous integration tool, to drive the entire process automatically, checking for new or changed code periodically and, if appropriate, triggering the build-and-test cycle
- Performing a static code analysis (that might be skipped or restricted in terms of scope if the individual units are already subjected to such an analysis prior to check-in)
- Building the release via whatever compile and link process is relevant for the particular language or languages being used
- Running the automated unit tests, not only for the new and changed code but for all code, measuring the code coverage achieved by these unit tests, and reporting the test results and coverage metrics
- If the static analysis, build, and unit tests are successful, installing the release into a test environment, which may be a dedicated environment for this purpose or the general environment used for system testing
- Running a set of automated integration tests, which should be (but regrettably seldom are) focused on testing interfaces between the units and reporting the test results
- Running higher-level automated tests, such as system tests and feature acceptance tests, which can in some cases include performance and reliability tests, though these test sets may take longer to run and analyze and thus might be run separately from the continuous integration process

The reporting of test results should also be entirely automated, either via updating an intranet site or project wiki or sending email to the project team. In the case of an Agile project, the task board may be updated to display status as well.

If bugs are found during the build-and-test cycle, developers can use debugging tools to chase down the problems. As a technical test analyst, you might be involved in helping to do this task, especially if you have strong programming

skills. However, you'll want to make sure the test manager is aware that you are doing so, because that may distract you from tasks more properly within your scope.

When these processes work well, they can almost entirely eliminate the broken-build or untestable-build problems that can plague many test teams. The automated tests are also effective at catching some percentage of regression bugs as well. However, if only automated unit tests are included in the test set, the regression detection abilities will be limited. Including high-level tests in the process will greatly increase regression risk mitigation.

In Figure 6–8, you can see an example of a continuous integration process that Rex and his colleagues created for a client. The four steps of the process are shown in the figure. UCI stands for “unit, component, and integration,” which were the three levels of tests that the programmers were responsible for defining and automating. It proved rather difficult with this team—as it has with most programmer teams Rex has worked with—to get them to create good integration tests. As a technical test analyst, you should help support this process because when specific integration tests aren't done, system test becomes a de facto big bang integration test level, with all the risk associated with that approach to integration testing. This project happened in early 2001, so you can see that continuous integration is a well-established, time-proven technique.⁵

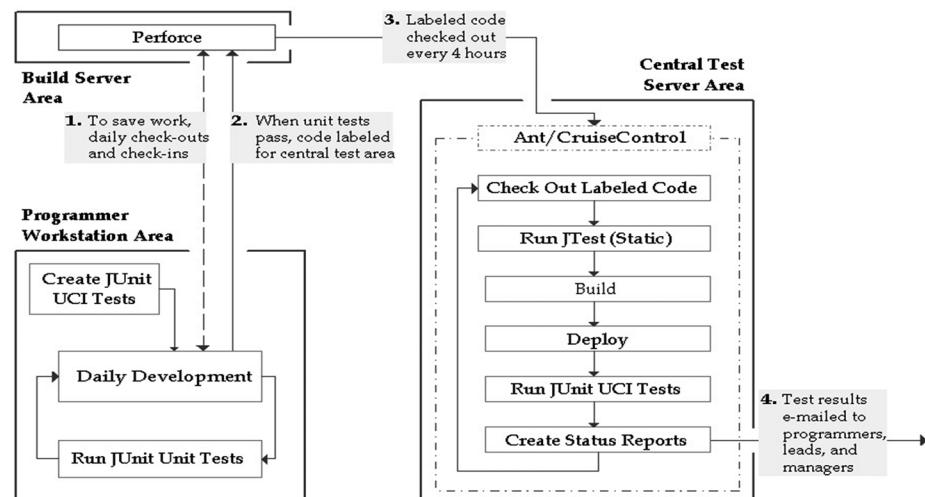


Figure 6–8 A continuous integration process with associated tools

5. You can read more about this system in the article “Mission Made Possible,” written by Greg Kubaczkowski and Rex Black, posted on the RBCS website at www.rbcus.com.

6.4 Sample Exam Questions

1. A test team uses a test management tool to capture its test cases, test results, and bug reports. The business analysts use a separate tool to capture business requirements and system requirements. Which of the following tool integration considerations most applies in this situation?
 - A. Traceability
 - B. Defect classification
 - C. Return on investment
 - D. Probe effect
2. Consider the following costs:
 - I. Designing and documenting the test automation architecture
 - II. Ongoing license fees
 - III. Extending coverage for new features
 - IV. Hardware and software to run the automation tool
 - V. Maintenance costs of scripts
 - VI. Integrating automation processes into the projectWhich of the above should be considered as initial costs and which should be seen as recurring costs?
 - A. All are initial costs.
 - B. I, II, IV are initial costs; III, V, and VI are recurring costs.
 - C. I, IV, VI are initial costs; II, III, and V are recurring costs.
 - D. I, III, and VI are initial costs; II, V, and VI are recurring costs.
3. Your organization has just spent over one million dollars on a capture/replay tool set to introduce automation to an ongoing project. You have been named the lead automator for the automation project. Which of the following tasks should be considered the most important for you to deal with right away?
 - A. Begin building scripts to start accumulating automation assets.
 - B. Start training all testers on how to use the automation tool set.
 - C. Research and request a test management tool to manage the automation.
 - D. Select the automation architecture you are going to pursue.

4. Select the reason an organization may choose to build a keyword-driven architecture instead of a data-driven architecture.
 - A. To allow automation of functional tests
 - B. To allow multiple tests to use the same script
 - C. To allow automation of regression tests
 - D. To allow for better use of technically trained testers
5. Which of the following are possible reasons that automation projects fail to deliver value?
 - I. Testers often do not know how to write code.
 - II. Stakeholders' expectations are often unrealistic.
 - III. Excessive interface changes from build to build.
 - IV. Underestimating the amount of time and resources needed.
 - V. Manual testing processes are too immature.
 - VI. Tool used does not work well in test environment.
 - VII. Failure to design an architecture that anticipates change.
 - A. II, III, IV, V, VI
 - B. I, III, IV, V, VII
 - C. None of these
 - D. All of these
6. Which of the following points of information about your organization would tend to make keyword-driven automation a desirable automation method rather than using a straight data-driven methodology?
 - A. Almost all of the testers on your test team have backgrounds in programming.
 - B. The systems you are testing have radical interface changes at least three times a year.
 - C. Most of your testers came from the user or business community.
 - D. Your organization has a limited budget for purchasing tools.

7. As the lead automator, you have been tasked with creating the design for the keyword-driven architecture. You have decided that, because so many of the testers who will be creating tests are highly experienced using the current interface, you will allow very granular keywords (e.g., ClickButton(btnName), TypeEdit(edtName, Text), etc.). Which of the following may be the worst disadvantage of allowing this level of granularity?
 - A. Testers may have problem using the low-level keywords.
 - B. Keyword tests will become vulnerable to interface changes.
 - C. Keyword tests will likely become too long to understand.
 - D. The framework that supports execution will be too brittle.
8. Which of the following statements captures the difference between fault seeding and fault injection?
 - A. Fault seeding involves corrupting inputs, data, and events, while fault injection involves systematically introducing defects into the code.
 - B. Fault injection involves corrupting inputs, data, and events, while fault seeding involves systematically introducing defects into the code.
 - C. Fault injection and fault seeding are the same; both involve corrupting inputs, data, and events.
 - D. Fault injection and fault seeding are the same; both involve systematically introducing defects into the code.
9. You are in the analysis and design phase of your performance testing project. You have evaluated the production and test environments. You have created the data to be used and built and parameterized the scripts. You have set up all of the monitoring applications and notified the appropriate support personnel so they are ready to troubleshoot problems. Which of the following tasks, had it not been done, would surely invalidate all of your testing?
 - A. Ensure that the test environment is identical to the production environment.
 - B. Model the system to learn how it's actually used.
 - C. Purchase or rent enough virtual user licenses to match peak usage.
 - D. Bring in experienced performance testers to train all of the participants.

10. You are senior technical test analyst for a test organization that is rapidly falling behind the curve; each release, you are less able to perform all of the testing tasks that are needed by your web project. You have very little budget for tools or people, and the time frame for the project is about to be accelerated. The testers in the group tend to have very little in the way of technical skills. Currently, 100 percent of your testing is manual, with about 15 percent of that being regression testing. Which of the following decisions might help you catch up to the curve?
- A. Allow the testers to use open-source tools to pick low-hanging fruit.
 - B. Put a full automation project into place and try to automate all testing.
 - C. Find an inexpensive requirements/test management tool to roll out.
 - D. Build your own automation tool so it does not cost anything.
11. Which of the following is unique about the way model-based testing tools support testing compared to typical test execution tools?
- A. The tools can automatically generate test results logs.
 - B. The tools can compare actual and expected results for anomalies.
 - C. The tools can generate the expected results of the test from a model.
 - D. The tools can generate the model from any type of requirements specification.
12. Which of the following is a way that component testing and build tools directly benefit testers?
- A. The programmer can step through code to find defects.
 - B. The tools automatically update the Agile task board.
 - C. The testers don't have to worry about regression of existing features.
 - D. The incidence of broken or untestable builds is reduced.

7 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 Rex when teaching ISTQB courses

The seventh chapter of this book is concerned with topics that you need to know in order to prepare for the ISTQB Advanced Technical Test Analyst exam. The chapter starts with a discussion of the ISTQB Advanced Technical Test Analyst syllabus 2012 learning objectives, which are the basis of the exams.

Chapter 7 of this book has two sections:

1. Learning Objectives
2. ISTQB Advanced Exams

If you are not interested in taking the ISTQB Advanced Technical Test Analyst exam, this chapter might not be pertinent for you.

Learning Objectives

National boards and exam boards develop the Advanced level exams based on learning objectives. A learning objective states what you should be able to do prior to taking an Advanced level exam. Each Advanced level exam has its own set of learning objectives and its own text. There are no shared learning objectives or text across the three separate Advanced level syllabi, so don't bother to read or study the other two Advanced level syllabi if you're studying for the Advanced Technical Test Analyst exam. We listed the learning objectives for the Advanced Technical Test Analyst 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 answers for the questions. 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 white-box test case, you will have to remember and understand the contents of such a document. 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 answers.

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 Technical Test Analyst syllabus.

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*, and *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 follows:

- non-delivery 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.

This means that you should be able to remember the ISTQB glossary definitions of terms used in the ISTQB Advanced Technical Test Analyst syllabus. Expect this level of learning to be required for questions focused on higher levels of learning like K3 and K4. There are no K1 questions, but your ability to remember the Foundation, Advanced Overview, and Advanced Technical Test Analyst syllabi are all examinable at the K1 level, embedded within K2, K3, and K4 questions.

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 procedures, 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 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.

This means that you should be able to understand the ISTQB glossary terms used in the ISTQB Advanced Technical Test Analyst syllabus. Expect this level of learning to be required for questions focused on higher levels of learning like K3 and K4.

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 Technical Test Analyst 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:

- Design tests to achieve statement, decision, and MC/DC coverage.
- Create a set of keywords for keyword-driven test automation.

This means that you should be able to apply the techniques described in the ISTQB Advanced Technical Test Analyst syllabus to specific exam questions. Expect this level of learning to include lower levels of learning like K1 and K2.

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.
- Evaluate a set of tests to determine what level of white-box coverage (if any) is achieved.

This means that you should be able to analyze the techniques and concepts described in the ISTQB Advanced Technical Test Analyst 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.

Where Did These Levels of Learning Objectives 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.

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 often covered implicitly within the higher-level questions.

In addition, unlike with the Foundation exam, the questions are weighted. K3 and K4 questions will be assigned two and three points, respectively, and in most cases K2 questions will be assigned only one point.

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?

- A. Test items
- B. Probe effect
- C. Purpose
- D. 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 the ability to recall song lyrics from the 1970s qualifies someone 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?

- A. Filing a defect report describing the time and date at which you first noticed the missing test release
- B. Creating a test that describes how to install a test release
- C. Sending an SMS text message to the configuration management team manager

- D. Sending an email to the project manager and the configuration management team manager

The answer is C. A, B, and D are distractors. 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 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 the contents of documents such as the test plan, defect report, and test item transmittal report. In fact, you'll probably get lost in the terminology if you don't know the standard. However, simply knowing the IEEE 829 standard for test documents will not allow you to get the right answer on this question except by chance.

Scenario-Based and Pick-N 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 US 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 US English).

Consider the following possible actions:

- I. Notifying the configuration management team manager
- II. Notifying the project manager
- III. Filing a defect report
- IV. Attempting to repeat the installation
- V. Suspending testing
- VI. 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?

- A. I, II, V
- B. V, I, IV, III, I
- C. VI, II, I, III, IV
- D. II, I, V

The answer is B, while A, C, and D are distractors. A is wrong because there is no defect 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 defect 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 defect 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.

In addition to scenario questions, you'll see another new type of question on the Advanced exams, Pick-N questions. In these questions, you will pick two or three answers out of a list of five or seven options, respectively. These questions are often a harder form of a Roman-type question, in that it is more difficult to use a process of elimination to select the right answer. And, if you only get some of the right answers—that is, one out of two or two out of three—you might not get partial credit.

Let us go back to this question of learning objectives for a moment. We 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. So, you’ll want to read the Advanced Technical Test Analyst syllabus carefully, a number of times.

Not only should you read the Advanced Technical Test Analyst syllabus, but you’ll need to go back and refresh yourself on the Foundation syllabus. Material that is examinable at the Foundation level is also examinable at the Advanced level, especially when material in the Advanced level builds on the Foundation level. It would be smart to take a sample Foundation exam and reread the Foundation syllabus as part of studying for the Advanced Technical Test Analyst exam.

On the Structure of the Exams

So, enough about the questions on the exam; what can you expect from the exam itself? In the Advanced Technical Test Analyst exam, you will get 45 questions. You’ll have two hours (120 minutes) to complete it. (If your native language is not the same as the language of the exam, you’ll be allowed an extra 30 minutes, for a total of 150 minutes.) Most of our customers find that the time limitation is not an issue, unlike with the Foundation exam, where a significant percentage of people need the entire hour.

Now, with the Foundation exam, you could estimate how many questions were going to be asked on each section by using the time allocated in the syllabus for that section. This trick will not work on the Advanced Technical Test Analyst exam. For the Advanced Technical Test Analyst exam, when we wrote the exam guidelines, we used a process of weighting the learning objectives for importance. So, for Chapter 1, here are the number of questions per learning objective:

- TTA-1.3.1 (K2): 1
- TTA-1.x.1 (K2): 1

There will be two questions on Chapter 1. Remember that an exam question might cover multiple learning objectives, so it could be that a single question counts against two (or more) learning objectives for this chapter or across multiple chapters.

For Chapter 2, here are the number of questions per learning objective:

- TTA-2.2.1 (K2): 1
- TTA-2.3.1 (K3): 1
- TTA-2.4.1 (K3): 1
- TTA-2.5.1 (K3): 1
- TTA-2.6.1 (K3): 1
- TTA-2.7.1 (K2): 2
- TTA-2.8.1 (K4): 2

There will be nine questions on Chapter 2.

For Chapter 3, here are the number of questions per learning objective:

- TTA-3.2.1 (K3): 3
- TTA-3.2.2 (K3): 1
- TTA-3.2.3 (K3): 1
- TTA-3.2.4 (K2): 1
- TTA-3.3.1 (K3): 1

For Chapter 3, you will see seven questions, as specified above.

For Chapter 4, here are the number of questions per learning objective:

- TTA-4.2.1 (K4): 2
- TTA-4.3.1 (K3): 1
- TTA-4.4.1 (K3): 1
- TTA-4.5.1 (K3): 1
- TTA-4.x.1 (K2): 1
- TTA-4.x.2 (K3): 2
- TTA-4.x.3 (K2): 2
- TTA-4.x.4 (K3): 2

For Chapter 4, you will see 12 questions, as specified above.

For Chapter 5, here are the number of questions per learning objective:

- TTA-5.1.1 (K2): 1
- TTA-5.2.1 (K4): 2
- TTA-5.2.2 (K4): 2

You'll see five questions on Chapter 5.

For Chapter 6, here are the number of questions per learning objectives:

- TTA-6.6.1 (K2): 1
- TTA-6.2.1 (K2): 1
- TTA-6.2.2 (K2): 1
- TTA-6.2.3 (K2): 1
- TTA-6.2.4 (K3): 1
- TTA-6.3.1 (K2): 1
- TTA-6.3.2 (K2): 1
- TTA-6.3.3 (K2): 1
- TTA-6.3.4 (K2): 1
- TTA-6.3.5 (K2): 1

You'll see 10 questions for Chapter 6.

Based on the suggested point allocation—1 point for a K1 question, 2 points for a K3 question, and 3 points for a K4 questions—you'll see 79 total points. You have to get 52 points (65 percent) to pass.

Okay, we realize that you might be panicking. Don't panic! Remember, the exam is meant to test your achievement of the learning objectives in the Advanced Technical Test Analyst 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 Technical Test Analyst syllabus? If so, then you know the material in the ISTQB Advanced Test Manager syllabus. If not, then review the ISTQB Advanced Technical Test Analyst syllabus and reread those sections of this book that correspond to the parts of the syllabus you find most confusing.

We 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 Technical Test Analyst 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 Technical Test Analyst syllabus to your next testing project.

Appendix

Bibliography

Advanced Syllabus Referenced Standards

The following standards are mentioned in the Advanced Technical Test Analyst syllabus.

ANSI/IEEE Std 1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*.

ISO/IEC 61508, *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*.

ISO/IEC 25000:2005, *Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)*.

ISO/IEC 9126-1:2001, *Software Engineering – Software Product Quality*.

RTCA DO-178B/ED-12B, *Software Considerations in Airborne Systems and Equipment Certification*.

Advanced Syllabus Referenced Books

Len Bass, Paul Clements, Rick Kazman. *Software Architecture in Practice (2nd edition)*. Addison-Wesley, 2003.

Graham Bath, Judy McKay. *The Software Test Engineer's Handbook*. Rocky Nook, 2008.

Boris Beizer. *Software Testing Techniques, Second Edition*. International Thomson Computer Press, 1990.

Boris Beizer. *Black-Box Testing*. John Wiley & Sons, 1995.

Hans Buwalda. *Integrated Test Design and Automation*. Addison-Wesley Longman, 2001.

Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, 2003.

- Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- Paul C. Jorgensen. *Software Testing, a Craftsman's Approach, Third Edition*. CRC Press, 2007.
- Cem Kaner, James Bach, Bret Pettichord. *Lessons Learned in Software Testing*. John Wiley & Sons, 2002.
- Tim Koomen, Leo van der Aalst, Bart Broekman, Michael Vroon. *TMap Next for Result-Driven Testing*. UTN Publishers, 2006.
- Thomas J. McCabe. "A Complexity Measure." *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4. December 1976. PP 308–320.
- Steven Splaine, Stefan P. Jaskiel. *The Web-Testing Handbook*. STQE Publishing, 2001.
- Mark Utting, Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.
- Arthur H. Watson and Thomas J. McCabe, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric." NIST Special Publication 500-235. Prepared under NIST Contract 43NANB517266, September 1996.
- James Whittaker and Herbert Thompson. *How to Break Software Security*. Pearson/Addison-Wesley, 2004.
- Karl Wiegers. *Peer Reviews in Software: A Practical Guide*. Addison-Wesley, 2002.

Advanced Syllabus Other References

<http://www.testingstandards.co.uk>

<http://www.nist.gov> NIST National Institute of Standards and Technology

<http://www.codeproject.com/KB/architecture/SWArchitectureReview.aspx>

<http://portal.acm.org/citation.cfm?id=308798>

http://www.processimpact.com/pr_goodies.shtml

<http://www.ifsq.org>

<http://www.W3C.org>

Other Referenced Books

Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice (3rd Edition)*. Addison-Wesley, 2013.

Graham Bath and Judy McKay. *The Software Test Engineer's Handbook: A Study Guide for the ISTQB Test Analyst and Technical Test Analyst Advanced Level Certificates 2012*. Rocky Nook, Inc., 2014.

Boris Beizer. *Software Testing Techniques, Second Edition*. International Thomson Computer Press, 1990.

Rex Black. *Advanced Software Testing—Vol. 1: Guide to the ISTQB Advanced Certification as an Advanced Test Analyst*. Rocky Nook, Inc., 2009.

Rex Black. *Advanced Software Testing—Vol. 2, Second Edition*. Rocky Nook, Inc., 2014.

Rex Black. *Critical Testing Processes: Plan, Prepare, Perform, Perfect*. Addison-Wesley, 2004.

Rex Black. *Managing the Testing Process: Practical Tools and Techniques for Managing Software and Hardware Testing, Third Edition*. John Wiley & Sons, 2009.

Rex Black. *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*. John Wiley & Sons, 2007.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Malcolm Gladwell. *Outliers: The Story of Success*. Little, Brown, and Company, 2008.

Capers Jones. *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley, 2000.

Paul Jorgensen. *Software Testing: A Craftsman's Approach, Fourth Edition*. CRC Press, 2013.

Tim Koomen, Leo van der Aalst, Bart Broekman, and Michiel Vroon. TMap Next for Result-Driven Testing. UTN Publishers, 2006.

Brian Marick. *The Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing*. Prentice Hall, 1994.

Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, 2009.

- Tim Riley and Adam Goucher (editors). *Beautiful Testing: Leading Professionals Reveal How They Improve Software*. O'Reilly, 2010.
- Steven Soter and Neil deGrasse Tyson (editors). *Cosmic Horizons: Astronomy At The Cutting Edge*. New Press, 2001.
- Y. N. Srikant and Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*, Second Edition. CRC Press, 2008.
- D. H. Stamatis. *Failure Mode and Effect Analysis: FMEA from Theory to Execution, Second Edition*. American Society for Quality. Quality Press, 2003.
- Karl Wiegers. *Peer Reviews in Software: A Practical Guide*. Addison-Wesley, 2001.
- Karl Wiegers and Joy Beatty. *Software Requirements, Third Edition*. Microsoft Press, 2013.
- Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, 1979.

Other References

The following references point to information available on the Internet. Even though these references were checked at the time of publication of this book, the authors cannot be held responsible if the references are not available anymore.

Testing in the API Economy: Top 5 Myths. Wayne Ariola and Cynthia Dunlop, 2014. A white paper available from <http://www.parasoft.com/>.

“Microsoft News—Microsoft Study Reveals that Regular Password Changes are Useless.” Patrick Barnard, 2010. <http://microsoft-news.tmcnet.com/microsoft/articles/81726-microsoft-study-reveals-that-regular-password-changes-useless.htm>.

“Advanced Risk-Based Test Results Reporting: Putting Residual Quality Risk Measurement in Motion.” Rex Black and Nagata Atsushi. <http://www.rbcstechnology.com/images/documents/STQA-Magazine-1210.pdf>.

“Engineering Quality Goes Bananas: How a ‘Dumb Monkey’ Helped One Company Automate Function Testing.” Rex Black, Daniel Derr, and Michael Tyszkiewicz. “Software Test and Performance.” January 2009. <http://www.rbcstechnology.com/images/documents/engineering-quality-goes-bananas.pdf>.

“Mission Made Possible: How one team harnessed tools and procedures to test a complex, distributed system during development.” Rex Black and Greg

Kubaczkowski. <http://www.rbcus-us.com/images/documents/Mission-Made-Possible.pdf>.

“A Case Study in Successful Risk-Based Testing at CA.” Rex Black, Ken Young, and Peter Nash. <http://www.rbcus-us.com/images/documents/A-Case-Study-in-Risk-Based-Testing.pdf>.

“A Survey of Software Inspection Checklists” [Available for Purchase]. Bill Brykczyński. *ACM SIGSOFT Software Engineering Notes*, Volume 24, Issue 1, Jan 1999, Page 82. <http://dl.acm.org/citation.cfm?id=308798>.

Common Attack Pattern Enumeration and Classification. capec.mitre.org.

“A Look into Insidious Threats—The Logical Bomb.” Emmanuel Carabott. September 22, 2010. *GFI Blog*. <http://www.gfi.com/blog/insidious-threats-logical-bomb/>.

“Cyber War 2.0 – Russia v. Georgia.” Ward Carroll. 2008. *DefenseTech* at Military.com. <http://defensetech.org/2008/08/13/cyber-war-2-0-russia-v-georgia/>.

Common Vulnerabilities and Exposures. <http://cve.mitre.org/>.

“Hacker says to show passenger jets at risk of cyber-attack.” Jim Finkle. Reuters, Aug. 4, 2014. <http://www.reuters.com/article/2014/08/04/us-cybersecurity-hackers-airplanes-idUSKBN0G40WQ20140804>.

“Someday Never Comes,” from the album *Mardi Gras*. Written by John Fogerty and performed by Credence Clearwater Revival, 1972.

“catopen() may pose security risk for third party code.” FreeBSD, Inc., 2000. <https://www.freebsd.org/security/advisories/FreeBSD-SA-00:53.catopen.asc>.

API Integrity: An API Economy Must-Have. Gartner Research and Parasoft. <http://alm.parasoft.com/api-testing-gartner>.

The Java Language Specification: Java SE 8 Edition. James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 03/03/2014. <https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>.

A Practical Tutorial on Modified Condition/Decision Coverage. Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. <http://shemesh.larc.nasa.gov/fm/papers/Hayhurst-2001-tm210876-MCDC.pdf>.

“Don’t: the Secret of Self Control.” Jonah Lehrer. *The New Yorker*, May 18, 2009. <http://www.newyorker.com/magazine/2009/05/18/dont-2?currentPage=1>.

DoD Software Fault Patterns. Dr. Nikolai Mansouroff. <https://buildsecurityin.us-cert.gov/sites/default/files/Mansouroff-SoftwareFaultPatterns.pdf>.

“A Complexity Measure.” Thomas J. McCabe. *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976. <http://www.literateprogramming.com/mccabe.pdf>.

Performance Testing Guidance for Web Applications. J.D. Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea for Microsoft Corporation, September 2007. <http://msdn.microsoft.com/en-us/library/bb924375.aspx>.

“The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information.” George A. Miller. *The Psychological Review*, 1956, vol. 63, pp. 81-97. <http://www.musanim.com/miller1956/>.

Software Assurance Standard: NASA Technical Standard. National Aeronautics and Space Administration, NASA-STD-8739.8 w/Change 1, July 28, 2004. <http://www.hq.nasa.gov/office/codeq/doctree/87398.pdf>.

CWE—Common Weakness Enumeration. National Vulnerability Database, NIST. <http://nvd.nist.gov/cwe.cfm>.

Software Architecture Review Guidelines. Alexander Nowak. Sept. 12, 2007. <http://www.codeproject.com/Articles/20467/Software-Architecture-Review-Guidelines>

Code Review Checklist. OpenLazlo. http://wiki.openlaszlo.org/Code_Review_Checklist.

XSS (Cross Site Scripting) Prevention Cheat Sheet. OWASP.org, 04/12/2014. [https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet).

“Bay Bridge bolt problem arose from quality control lapses, officials say.” Will Reisman. *The Examiner*, March 27, 2013. <http://www.sfexaminer.com/sanfrancisco/bay-bridge-bolt-problem-arose-from-quality-control-lapses-officials-say/Content?oid=2336143>.

“There really are 50 Eskimo words for ‘snow?’” David Robson. *The Washington Post*, January 14, 2013. http://www.washingtonpost.com/national/health-science/there-really-are-50-eskimo-words-for-snow/2013/01/14/e0e3f4e0-59a0-11e2-beee-6e38f5215402_story.html.

Microsoft Windows Security Patches. Dan B. Rolsma. <http://www.sans.org/reading-room/whitepapers/windows/microsoft-windows-security-patches-273>.

Fallacies of Distributed Computing Explained: (The more things change the more they stay the same). Arnon Rotem-Gal-Oz. <http://www.rgoarchitects.com/Files/fallacies.pdf>.

Software Verification Tools Assessment Study, DOT/FAA/AR-06/54. Viswa Santhanam, John Joseph Chilenski, Raymond Waldrop, Thomas Leavitt, and Kelly J. Hayhurst. <http://www.tc.faa.gov/its/worldpac/techrpt/ar0654.pdf>.

Software Architecture, Software Engineering Institute. Carnegie Mellon University. <http://www.sei.cmu.edu/architecture/>.

The Golden Age of APIs. SmartBear. <http://www.soapui.org/The-World-Of-API-Testing/the-golden-age-of-apis.html>.

Writing Objectives Using Bloom's Taxonomy. The Center for Teaching and Learning, UNC Charlotte. <http://teaching.uncc.edu/learning-resources/articles-books/best-practice/goals-objectives/writing-objectives>.

NIST Special Publication 500-235, Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. Arthur H. Watson and Thomas J. McCabe. <http://www.itl.nist.gov/lab/specpubs/sp500.htm>.

Goodies for Peer Reviews. Karl Wiegers. http://www.processimpact.com/pr_goodies.shtml.

“Out-of-memory problem caused Mars rover’s glitch.” Todd R. Weiss, *Computerworld*, Feb. 3, 2004. <http://www.computerworld.com/article/2574759/data-storage-solutions/out-of-memory-problem-caused-mars-rover-s-glitch.html>.

“Splint (programming tool).” *Wikipedia*. [http://en.wikipedia.org/wiki/Splint_\(programming_tool\)](http://en.wikipedia.org/wiki/Splint_(programming_tool)).

Referenced Standards

BS 7925/2, *Software Component Testing Standard*.

DO-178C, *Software Considerations in Airborne Systems and Equipment Certification*.

IEEE 1008-1987, *IEEE—Standard for Software Unit Testing*.



HELLOCARMS
The Next Generation of Home Equity Lending

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 and authorized RBCS course attendees.*

This page deliberately blank.

I Table of Contents

HELLOCARMS THE NEXT GENERATION OF HOME EQUITY LENDING	409
SYSTEM REQUIREMENTS DOCUMENT	409
I TABLE OF CONTENTS	411
II VERSIONING	413
III GLOSSARY	415
000 INTRODUCTION	417
001 INFORMAL USE CASE	418
003 SCOPE	419
004 SYSTEM BUSINESS BENEFITS	420
010 FUNCTIONAL SYSTEM REQUIREMENTS	421
020 RELIABILITY SYSTEM REQUIREMENTS	425
030 USABILITY SYSTEM REQUIREMENTS	426
040 EFFICIENCY SYSTEM REQUIREMENTS	427
050 MAINTAINABILITY SYSTEM REQUIREMENTS	429
060 PORTABILITY SYSTEM REQUIREMENTS	430
A ACKNOWLEDGEMENT	431

This page deliberately blank.

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	
0.6	Feb 10, 2010	Jamie Mitchell	Fourth Draft	
0.7	July 20, 2010	Jamie Mitchell	Release for ATTA	

This page deliberately blank.

III Glossary

<i>Term¹</i>	<i>Definition</i>
Home Equity	The difference between a home's fair market value and the unpaid balance of the mortgage and any other debt secured by the home. 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. A homeowner can borrow against home equity using <i>home equity loans</i> , <i>home equity lines of credit</i> , and <i>reverse mortgages</i> (see below).
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 pre-arranged 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.

1. These definitions are adapted from www.dictionary.com.

This page deliberately blank.

000 Introduction

The Home Equity Loan, Line-of-Credit, and Reverse Mortgage System (HELOCARMS), 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 HELOCARMS 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.

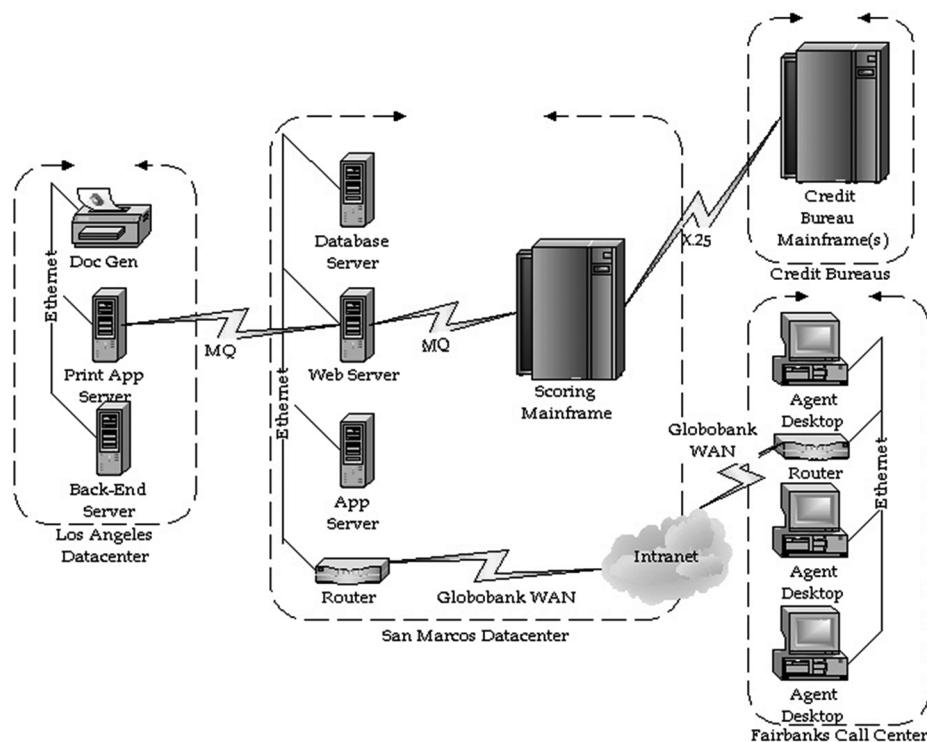


Figure 1 HELLOCARMS System (First Release)

001 Informal Use Case

The following informal use case applies for typical transactions in the HELLOCARMS System:

1. A Globobank Telephone Banker in a Globobank Call Center receives a phone call from a Customer.
2. 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.
3. 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.
4. 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.
5. If the Customer chooses one of these Products, the Telephone Banker will conditionally confirm the Product.
6. 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.
7. The HELLOCARMS system receives an update from the LoDoPS System when the following events occur:
 - a) LoDoPS system sends documents to customer;
 - b) Globobank Loan Servicing Center receives signed documents from customer; and,
 - c) Globobank Loan Servicing Center sends 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 pre-approved 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, e-mail, 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, since 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	
010-010-010	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
010-010-090	Allow user to abort an application. Provide an abort function on all screens.	3
010-010-100	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
010-010-110	Exclude a debt's monthly payment from the debt ratio if the customer requests the debt to be paid off.	3

* Priorities are:
 1 Very high
 2 High
 3 Medium
 4 Low
 5 Very low.

ID	Description	Priority*
010-010-120	Provide a means of requesting an existing application by customer identification number if a customer does not have their loan identifier.	4
010-010-130	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
010-010-140	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
010-010-150	Provide inbound and outbound telemarketing support for all States, Provinces, and Countries in which Globobank operates.	2
010-010-160	Support brokers and other business partners by providing limited partner-specific screens, logos, interfaces, and branding.	2
010-010-170	Support the submission of applications via the Internet, which includes the capability of untrained users to properly enter applications.	3
010-010-180	Provide features and screens that support the operations of the Globobank's retail branches.	4
010-010-190	Support the marketing, sales, and processing of home equity applications.	1
010-010-200	Support the marketing, sales, and processing of home equity line of credit applications.	2
010-010-210	Support the marketing, sales, and processing of home equity reverse mortgage applications.	3
010-010-220	Support the marketing, sales, and processing of applications for combinations of financial products (e.g., home equity and credit cards).	4
010-010-230	Support the marketing, sales, and processing of applications for original mortgages.	5
010-010-240	Support the marketing, sales, and processing of pre-approved applications.	4
010-010-250	Support flexible pricing schemes including introductory pricing, short term pricing, and others.	5
010-020	Accuracy	
010-020-010	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
010-020-020	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

ID	Description	Priority*
010-020-030	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
010-020-040	<p>Add a loan fee based on property type:</p> <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. <p>Do not add a loan fee for the other supported property type, residential single family dwelling.</p>	3
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	<i>Interoperability</i>	
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
010-030-070	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

ID	Description	Priority*
010-030-080	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
010-030-090	Pass application information to the Scoring Mainframe.	1
010-030-100	Receive scoring and decision information back from the Scoring Mainframe.	1
010-030-110	If the Scoring Mainframe is down, queue application information requests.	2
010-030-120	Initiate the origination process by sending the approved loan to LoDoPS.	2
010-030-130	Pass all declined applications to LoDoPS.	2
010-030-140	Receive LoDoPS feedback on the status of applications.	2
010-030-145	Receive changes to loan information made in LoDoPS (e.g., loan amount, rate, etc.).	2
010-030-150	Support computer-telephony integration to provide customized marketing and sales support for inbound telemarketing campaigns and branded business partners.	4
010-040	<i>Security</i>	
010-040-010	Support agreed upon security requirements (encryption, firewalls, etc.)	2
010-040-020	Track "Created By" and "Last Changed By" audit trail information for each application.	1
010-040-030	Allow outsourced telemarketers to see the credit tier but disallow them from seeing the actual credit score of applicants.	2
010-040-040	Support the submission of applications via the Internet, providing security against unintentional and intentional security attacks.	2
010-040-050	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
010-040-060	Support fraud detection for processing of all financial applications.	1
010-050	<i>Compliance (functionality standards/laws/reg)</i>	
	[To be determined in a subsequent revision]	

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	<i>Maturity</i>	
020-010-010	During the SDLC, cyclomatic complexity measurements will be evaluated for all modules to ensure that failures due to complexity are reduced.	1
020-010-020	The system shall average less than five (5) failures per month in production.	2
020-010-030	The system shall average less than one (1) failure per month in production.	4
020-020	<i>Fault-tolerance</i>	
020-020-010	The HELLOCARMS system will contain functionality to help prevent incorrect data to be inputted to the system. When an application is ready to be submitted, it will be evaluated statically to make sure it meets minimum correctness standards before being officially submitted to the system.	2
020-030	<i>Recoverability</i>	
020-030-010	In case of a disconnection of the Telephone Banker's workstation from HELLOCARMS while dealing with a customer, the system shall restore the work to the same state when reconnected.	1
020-040	<i>Compliance (reliability standards/laws/reg)</i>	
	[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	<i>Understandability</i>	
030-010-010	Support the submission of applications via the Internet, including the capability for untrained users to properly enter applications.	2
030-010-020	All screens, instructions, help and error messages shall be understandable at an 8th grade level.	2
030-010-020	All functionality shall be evident to the casual user without having to search for it in compliance with ISO-9126.	3
030-020	<i>Learnability</i>	
030-020-010	All pages shall be self contained with all control information built into the page such that the user does not have to leave the screen to get help.	2
030-020-020	HELLOCARMS will include a self-contained training wizard for all users. This wizard will lead a new user through all of the screens using canned data. The training will be sufficient that an average user will become proficient in the use of HELLOCARMS within 8 hours of training.	3
030-030	<i>Operability</i>	
030-030-010	Input fields, where possible, will immediately check for valid input upon exiting the control. If an input cannot be validated singularly, it will be validated before leaving the screen.	2
030-030-020	All screens will comply with US Federal GSA Section 508 for accessibility	2
030-030-030	Provide for complete customization of the user interface and all user user-supplied documents for business partners, including private branding of the sales and marketing information and all closing documents.	3
030-030-040	All common scenarios shall have a common flow through the interface. In non-exceptional flows, control shall pass through each screen in the same direction reading normally occurs (i.e. left to right, up to down for English.)	3
030-040	<i>Attractiveness</i>	
030-040-010	The interface shall be attractive to the user taking into account colors and graphical design of each screen.	3
030-050	Compliance (usability standards)	
030-050-010	Comply with local handicap-access laws in countries outside the USA.	4

040 Efficiency System Requirements

The capability of the system to provide appropriate performance, relative to the amount of resources used under stated conditions. Assumptions are made that occasionally, performance will be worse than specified; 98% compliance over a 24 hour period will be deemed to be in compliance.

ID	Description	Priority
040-010	<i>Time behavior</i>	
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
040-010-040	Time overhead on Scoring Mainframe shall average less than .1 seconds. This includes any processing needed to transfer a request to and from the Credit Bureau Mainframe(s), but does not include Credit Bureau Mainframe processing time.	4
040-010-050	Credit-worthiness of a customer shall be determined within 10 seconds of request. 98% or higher of all Credit Bureau Mainframe requests shall be completed within 2.5 seconds of the request arriving at the Credit Bureau.	2
040-010-060	Archiving the application and all associated information shall not impact the Telephone Banker's workstation for more than .01 seconds.	2
040-010-070	Escalation to a Senior Telephone Banker shall require no more than 1 second delay.	3
040-010-080	Once a Senior Banker has made a determination, the information shall be transmitted to the Telephone Banker within two (2) seconds.	3
040-010-090	Once a Customer chooses a product from the list of tentative options she was offered, the Telephone Banker shall input the choice and get conditional confirmation of acceptance within 60 seconds.	2
040-010-100	If abort function is triggered by the Telephone Banker, the system shall clear and reload the workstation within 2 seconds in preparation for the next call.	4
040-010-110	Handle up to 2,000 applications per hour.	2
040-010-120	Handle up to 4,000 applications per hour.	3
040-010-130	Support a peak of 4,000 simultaneous (concurrent) application submissions.	4

ID	Description	Priority
040-010-140	Support a total volume of 1.2 million approved applications for the initial year of operation.	2
040-010-150	Support a total volume of 7.2 million applications during the initial year of operation.	2
040-010-160	Support a total volume of 2.4 million conditionally-approved applications for the initial year of operation.	2
	[More to be determined in a subsequent revision]	
<i>040-020</i>	<i>Resource utilization</i>	
040-020-010	Load Database Server to no more than 20% CPU and 25% Resource utilization average rate with peak utilization never more than 80% when handling 4,000 applications per hour.	3
040-020-020	Load Web Server to no more than 30% CPU and 30% Resource utilization average rate with peak utilization never more than 80% when handling 4,000 applications per hour.	3
040-020-030	Load App Server to no more than 30% CPU and 30% Resource utilization average rate with peak utilization never more than 80% when handling 4,000 simultaneous (concurrent) application submissions.	4
<i>040-030</i>	<i>Compliance (performance standards)</i>	
	[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	<i>Analyzability</i>	
050-010-010	Standards and guidelines will be developed and used for all code and other generated materials used in this project to enhance maintainability.	1
050-010-020	Diagnostics shall be built into the HELLOCARMS system to automatically try to determine the proximate cause of internally generated failures.	2
050-010-030	HELOCARMS will generate and save log information when errors are generated. These logs shall be versioned to ensure that subsequent errors do not erase potentially useful data. Logs shall contain the diagnostic information generated in requirement 050-010-020.	2
050-020	<i>Changeability</i>	
050-020-010	Modification complexity (see ISO 9126-2) for all changes to HELLOCARMS shall be measured and tracked throughout the SMLC.	
040-030	<i>Compliance (performance standards)</i>	
	[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
<i>060-010</i>	<i>Adaptability</i>	
060-010-010	HELOCARMS shall be configured to work on Internet Explorer browsers in the current version and at least one level backwards.	1
060-010-020	HELOCARMS shall be configured to work with Internet Explorer and Firefox in the current version and at least one level backwards.	2
060-010-030	HELOCARMS shall be configured to work with all popular browsers that represent five percent (5%) or more of the currently deployed browsers in any countries where Globobank does business.	3
<i>060-020</i>	<i>Installability</i>	
060-020-010	An installation package will be developed to allow brokers and other business partners to easily install the custom screens, logos, interfaces and branding changes as defined in requirement 010-010-160.	2
<i>060-030</i>	<i>Co-existence</i>	
060-030-010	Should not interact in any non-specified way with any other applications in the Globobank call centers or data centers.	1
	[More to be determined in a subsequent revision]	
<i>060-040</i>	<i>Replaceability</i>	
	Not applicable	
<i>060-050</i>	<i>Compliance</i>	
	[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.

This page deliberately blank.

Answers to Sample Questions

Chapter 1

- 1 C
2 C

Chapter 2

- 1 C
2 C
3 A
4 B
5 A
6 D
7 B
8 B
9 D
10 B
11 B and C

Chapter 3

- 1 C
2 D
3 C
4 B
5 A
6 C
7 A
8 C
9 B
10 D
11 B
12 A
13 C

Chapter 4

- 1 B and D
2 D
3 B
4 B
5 A

Chapter 5

- 1 B
2 D
3 A

Chapter 6

- 1 A
2 C
3 D
4 A
5 D
6 C
7 B
8 B
9 B
10 A
11 C
12 D

Index

A

access auditability 192
access controllability 193
activity recording 250
adaptability 264
adaptability metrics
 external 267
 internal 266
adaptability of data structures 266, 267
analyzability 245, 248
analyzability metrics
 external 250
 internal 250
ANSI 87B 30
anti-pattern 293
API 85
API testing 84
application programming interface (API)
 84
atomic condition 45
audit trail capability 250
author-bias problem 16
automation 319
automation architecture 335
automator 330
autonomy of testability 260
availability 214
availability of built-in test function 260
available coexistence 276, 277

B

background testing 224
basis path testing 77
basis paths 77
black-box techniques 94
Boris Beizer 71
Boris Bezier's standard 37
branch coverage 25
branch testing 35
breadth-first approach 13
breakdown avoidance 209
BS 7925/2 standard 94
buffer overflow 183

C

call graphs 149
call-graph-based integration testing 150
capture/replay architecture 337
change cycle efficiency 255
change impact
 257
change implementation elapse time 256
change recordability 255
change success ratio 258
changeability 245, 252
changeability metrics
 external 255
 internal 255
checklist in reviews 292
Checkstyle 147
code parsing tools 144

- code review checklist 304, 306
code review checklist exercise 306, 308
code review exercise 308, 310
code reviews 301
 checklist 301
code-based test techniques 21
coexistence 264, 275
coexistence metrics
 external 277
 internal 276
cohesion 252
common weakness enumeration (CWE)
 207
completeness of built-in test function 259
compliance 216, 230, 242, 261, 278
compliance metric
 external 242, 261
 internal 242, 261
compliance metrics 278
 external 216
 internal 216
condition coverage 25, 43
condition testing 44
conditional call 154
continued use of data 270, 271, 277
control flow analysis 116, 117
control flow anomalies 116
control flow example 65
control flow graph 25, 72, 119
control flow table 76
control flow testing 24, 25
coupling 252
 types 253
cross-site scripting 187
cyclomatic complexity 77, 117
cyclomatic complexity example 82
cyclomatic complexity metric 79
- D**
- data corruption prevention 193
data encryption 193
data flow analysis 128
data flow anomalies 127
data flow combination 130
data flow errors 128
data flow testing 128
data flow testing strategy 140
data transfer interception 185
data-driven architecture 346
data-driven automation 335
data-driven testing
 336
debugging tool 381
decision condition coverage 47
decision condition testing 48
decision coverage 25, 33
decision point 27
decision testing 35
define-use pair example 133
define-use pair notation 130
define-use pattern list 134
definition-use pair 128
denial of service attacks 184
depth-first approach 13
design predicate approach 153
design predicate exercise 161
design review checklist 299
Deutsch's design review checklist 299
diagnostic function support 251
DO-178C 49
dynamic analysis 163, 164
dynamic analysis exercise 168
dynamic analysis tools 163
dynamic testing 13

E

ease of installation 275
ease of set-up retry 274, 275
efficiency 220, 221
efficiency testing 220
 types of 222
efficiency testing exercise 242
encryption 185
endurance testing 223
error tolerance 207
essential complexity 123
estimated latent fault density 203
evolution of the Web 87
exam 387
exit criteria 2

F

failure analysis capability 251
failure analysis efficiency 251
failure avoidance 208, 210
failure density against test cases 204
failure resolution 204
fault density 204
fault detection 202
Fault Pattern 207
fault removal 202, 205
fault seeding tool 363
fault tolerance 207
fault tolerance metrics
 external 209
 internal 208
function inclusiveness 270, 271, 277

G

general planning issues 280
granularity 351

H

hardware environmental adaptability
 266, 267
hexadecimal converter 41
high availability 211
hyperlink test tool 374

I

impact of a problem 9
incorrect operation avoidance 208, 210
incremental strategy 149
installability 264, 271
installability metrics
 external 275
 internal 274
installation effort 274
installation flexibility 274
integration testing strategies 149
intelligent front end 352
ISO/IEC standard 61508 16
ISO-9126 testing responsibilities 179
ISTQB Advanced exams 391
iterative call 156
iterative conditional call 156
I/O devices utilization 239
I/O related errors 240
I/O utilization 238
I/O utilization message density 238

J

junction point 27

K

keyword 354
keyword-driven architecture 348
keyword-driven automation 335
keyword-driven tables 350
keyword-driven tables exercise 359
keyword-driven testing 336

L

learning objectives 387
level of risk 7
likelihood of a problem 8
load testing 222
logic bombs 186
loop coverage 25, 37
loop testing 37

M

maintainability compliance 261
maintainability of code 144
maintainability testing 244, 245
maintainability testing exercise 261
Marick's code review checklist 304
masking MC/DC 57
maturity 197, 201
maturity metrics
 external 203
 internal 202
maximum memory utilization 240
maximum transmission utilization 241
McCabe, Thomas 77
McCabe's design predicate approach 153
MC/DC 48
MC/DC coverage 50
mean amount of throughput 237
mean down time 214
mean I/O fulfillment ratio 240
mean occurrence of memory error 240
mean occurrence of transmission error
 241
mean of transmission error per time 241
mean recovery time 214
mean time between failures (MTBF) 205
mean time for turnaround 237
mean time to response 236
measurement tools 367
memory leak 164
memory leak detection 165

memory utilization 239
memory utilization message density 239
Miller, George A. 123
model-based testing 374
modification complexity 256
modification impact localization 257, 258
modified condition/decision coverage
 (MC/DC) 48
multiple condition coverage 25, 58
multiple condition testing 59
mutually exclusive conditional call 155

N

National Vulnerability Database 190
neighborhood integration 152
non-functional testing 177
nonincremental strategy 149

O

object exchange protocol 86
Open Laszlo code review checklist 306
Open Web Application Security Project
 (OWASP) 191
operational acceptance testing 197
operational profile 197
organizational environment adaptability
 266, 268

P

pairwise graph 151
path testing 70
path testing via flow graph 71
Perfmon 367
performance acceptance criteria 227
performance testing 220, 221, 222
 data 365
performance testing exercise 370
performance testing tool 364
piracy 182
portability compliance 278

portability testing 263, 264
portability testing exercise 278
porting user friendliness 267, 268
Pragmatic Risk Analysis and Management (PRAM) 2
PRAM 2
preventive testing 12
process block 26
product risk 2, 4
programming guidelines 147
programming standards 147
project risk 5
public APIs 88

Q

quality attributes 177
quality risk 4
quality risk analysis document 15

R

ratio of memory error/time 241
readiness of diagnostic function 250
record (capture)/playback tool 336
recoverability 211
recoverability metrics
 external 214
 internal 213
recoverability testing 197
reliability 216
reliability growth model 197
reliability testing 196, 197, 224
 example 217
 exercise 219
replaceability 264, 268, 276
replaceability metrics
 external 270
 internal 270
resource utilization 230, 238

resource utilization metrics
 external 239
 internal 238
resource utilization testing 221, 223
response time 235
restartability 215
restorability 213, 215
restoration effectiveness 213
restore effectiveness 215
re-test efficiency 260
reviews 287

risk 2
risk analysis 4
risk assessment 4, 7
risk control 4, 10
risk identification 4
risk impact 2
risk level 2
risk likelihood 2
risk management 4
risk mitigation 4, 10
risk-based testing 1, 2
 primary activities 3
 strategy 11
 techniques 5
robustness 197, 207

S

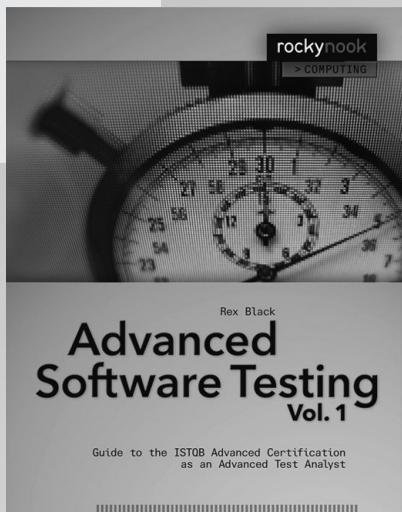
safety integrity level (SIL) 17
scalability testing 223
scripts 366
security metrics
 external 194
 internal 192
security testing 181
security vulnerabilities 189
short-circuiting 55
simple framework architecture 343
software architecture 296
software change control capability 256

- software design 297
software reliability 196
sources of risks 12
spike testing 223
Splint 145
stability 245, 256
stability metrics
 external 258
 internal 257
statement coverage 24, 29
statement testing 30
static analysis 116, 117
static analyzer 381
status monitoring capability 252
stress testing 223
structure-based test techniques 21, 23, 93
structure-based testing example 100
system modeling 225
system software environmental
 adaptability 266, 268
- T**
- technical test analyst
 role 1
test adequacy 202
test automation 323
test automation project 324
test automation projects
 failure 329
test coverage 206
test design 229
test environment 227, 228
test execution tool 336
test management tool 321
test maturity 206
test progress observability 260
test restartability 261
- test tools 319, 362
build process 380
component testing 380
fault injection 362
fault seeding 362
integration 319
monitoring 363
performance 363
testability 245, 258
testability metrics
 external 260
 internal 259
testing tools
 model-based testing 374
throughput 237
throughput time
 235
time behavior 230
time behavior metrics
 external 236
tip-over testing 224
tools for web-based testing 373
transmission capacity utilization 242
transmission utilization 239
turnaround time 235, 237
- U**
- unconditional call 154
uncoupled condition 57
unique cause MC/DC 57
user support functional consistency 271
user waiting time of I/O devices utilizati-
 on 240
- V**
- viruses 186
vulnerabilities 188

W

waiting time 237
white-box test design technique 23
white-box test techniques 21, 94
wild pointer 164

wild pointer detection 167
worms 186
worst case response time 236
worst case throughput ratio 237
worst case turnaround time ratio 237



\$ 49.95
488 pages, Soft Cover
ISBN: 978-1-933952-19-2

Rex Black

Advanced Software Testing—Vol. 1

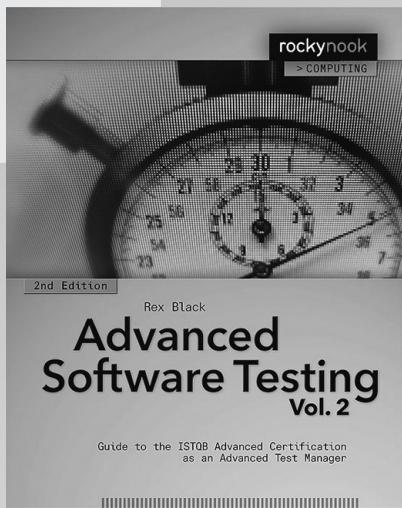
Guide to the ISTQB Advanced Certification as an Advanced Test Analyst

This book presents functional and This book is written for the test analyst who wants to achieve advanced skills in test analysis, design, and execution. With a hands-on, exercise-rich approach, this book teaches you how to define and carry out the tasks required to put a test strategy into action.

It will also help you prepare for the ISTQB Advanced Test Analyst exam. Included are sample exam questions, at the appropriate level of difficulty, for most of the learning objectives covered by the ISTQB Advanced Level syllabus. The ISTQB certification program is the leading software tester certification program in the world. With about 100,000 certificate holders and a global presence in 50 countries, you can be confident in the value and international stature that the Advanced Test Analyst certificate can offer you.

rockynook

Rocky Nook, Inc.
www.rockynook.com



\$ 54.95
536 pages, Soft Cover
ISBN: 978-1-937538-50-7

Rex Black

Advanced Software Testing—Vol. 2, 2nd Edition

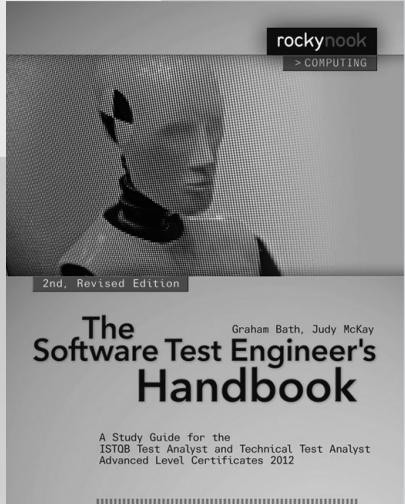
Guide to the ISTQB Advanced Certification as an Advanced Test Manager

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. It will help you prepare for the ISTQB Advanced Test Manager exam.

Included are sample exam questions, at the appropriate level of difficulty, for most of the learning objectives covered by the ISTQB Advanced Level Syllabus.

rockynook

Rocky Nook, Inc.
www.rockynook.com



\$ 49.95
560 pages, Soft Cover
ISBN: 978-1-937538-44-6

Graham Bath / Judy McKay

The Software Test Engineer's Handbook, 2nd Edition

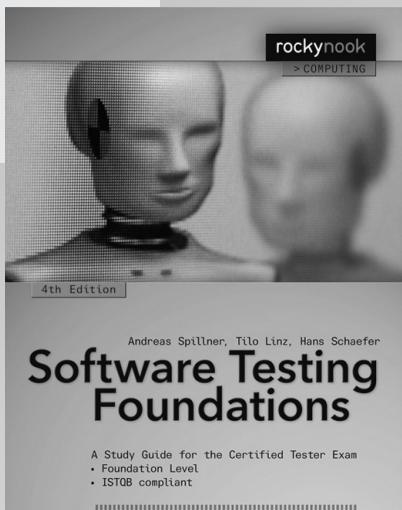
A Study Guide for the ISTQB Test Analyst and Technical Test Analyst Advanced Level Certificates 2012

This book presents functional and technical aspects of testing as a coherent whole, which benefits test analyst/engineers and test managers. It provides a solid preparation base for passing the ISTQB exams for Advanced Test Analyst and Advanced Technical Test Analyst, with enough real-world examples to keep you intellectually invested.

The book includes information that will help you become a highly skilled Advanced Test Analyst and Advanced Technical Test Analyst. You will be able to apply this information in the real world of tight schedules, restricted resources, and projects that do not proceed as planned.

rockynook

Rocky Nook, Inc.
www.rockynook.com



\$ 44.95
304 pages, Soft Cover
ISBN: 978-1-937538-42-2

Andreas Spillner / Tilo Linz /
Hans Schaefer

Software Testing Foundations, 4th Edition

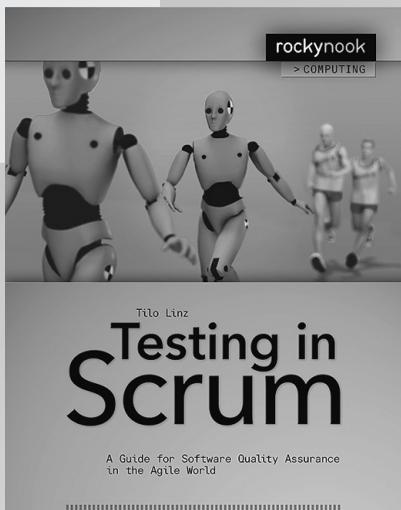
A Study Guide for the Certified Tester Exam

The International Software Testing Qualifications Board (ISTQB) has developed a universally accepted, international qualification scheme aimed at software and system testing professionals, and has created the Syllabi and Tests for the "Certified Tester."

This thoroughly revised and updated fourth edition covers the "Foundations Level" (entry level) and teaches the most important methods of software testing. It is designed for self-study and provides the information necessary to pass the Certified Tester—Foundations Level exam, version 2011, as defined by the ISTQB. Also in this new edition, technical terms have been precisely stated according to the recently revised and updated ISTQB glossary.

rockynook

Rocky Nook, Inc.
www.rockynook.com



\$ 39.95
240 pages, Soft Cover
ISBN: 978-1-937538-39-2

Tilo Linz

Testing in Scrum

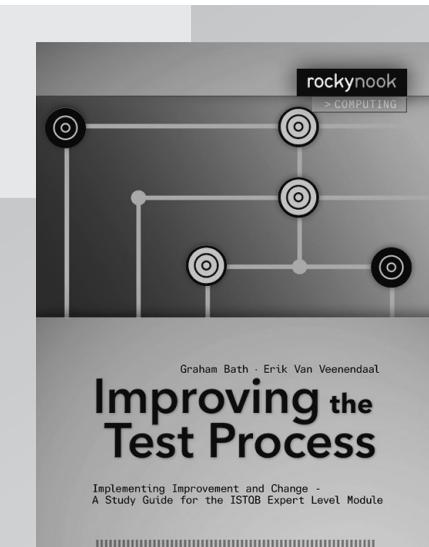
A Guide for Software Quality Assurance in the Agile World

This book discusses agile methodology from the perspective of software testing and software quality assurance management. Software development managers, project managers, and quality assurance managers will obtain tips and tricks on how to organize testing and assure quality so that agile projects maintain their impact.

Professional certified testers and software quality assurance experts will learn how to work successfully within agile software teams and how best to integrate their expertise.

rockynook

Rocky Nook, Inc.
www.rockynook.com



\$ 49.95
432 pages, Soft Cover
ISBN: 978-1-933952-82-6

Graham Bath /
Erik van Veenendaal

Improving the Test Process

Implementing Improvement and Change—
A Study Guide for the
ISTQB Expert Level Module

This book covers the syllabus for the »Improving the Test Process« module of the International Software Testing Qualifications Board (ISTQB) Expert Level exam.

To obtain certification as a professional tester at the Expert Level, candidates may choose to take a course given by an ISTQB accredited training provider and then sit for the exam. Experience shows that many candidates who choose this path still require a reference book that covers the course. There are also many IT professionals who choose self-study as the most appropriate route toward certification.

rockynook

Rocky Nook, Inc.
www.rockynook.com

