rockynook

> COMPUTING

Rex Black · Jamie Mitchell

Advanced Software Testing Vol. 3

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

About the Authors

With over a quarter-century of software and systems engineering experience, Rex Black is President of Rex Black Consulting Services (www.rbcs-us.com), a leader in software, hardware, and systems testing. RBCS delivers consulting, outsourcing, and training services, employing the industry's most experienced and recognized consultants. RBCS worldwide clientele save time and money through improved product development, decreased tech support calls, improved corporate reputation, and more.

Rex is the most prolific author practicing in the field of software testing today. His popular first book, *Managing the Testing Process*, has sold over 40,000 copies around the world, and is now in its third edition. His six other books—*Advanced Software Testing:* Volumes I, II, and III, *Critical Testing Processes*, *Foundations of Software Testing*, and *Pragmatic Software Testing*—have also sold tens of thousands of copies. He has written over thirty articles; presented hundreds of papers, workshops, and seminars; and given about fifty keynote and other speeches at conferences and events around the world. Rex is the immediate past President of the International Software Testing Qualifications Board (ISTQB) and a Director of the American Software Testing Qualifications Board (ASTQB).

Jamie L. Mitchell has over 28 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.

Jamie specializes in increasing the productivity of automation and test teams through innovative ideas and custom tool extensions. In addition, he provides training, mentoring, process auditing, and expert technical support in all aspects of testing and automation.

Jamie holds a Master of Computer Science degree from Lehigh University in Bethlehem, PA, and a Certified Software Test Engineer certification from QAI. He was an instructor and board member of the International Institute of Software Testing (IIST) and a contributing editor, technical editor, and columnist for the Journal of Software Testing Professionals. He has been a frequent speaker on testing and automation at several international conferences, including STAR, QAI, and PSQT.

Rex Black • Jamie L. Mitchell

Advanced Software Testing—Vol. 3

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

rockynook

Rex Black
rex_black@rbcs-us.com

Jamie L. Mitchell
jamie@go-tac.com

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

ISBN: 978-1-933952-39-0

1st Edition © 2011 by Rex Black and Jamie L. Mitchell
16 15 14 13 12 11   1 2 3 4 5

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)
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

Distributed by O'Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472-2811

All product names and services identified throughout this book are trademarks or registered trademarks of their respective companies. They are used throughout this book in editorial fashion only and for the benefit of such companies. No such uses, or the use of any trade name, is intended to convey endorsement or other affiliation with the book. No part of the material protected by this copyright notice may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or bay any information storage and retrieval system, without written permission from the copyright owner.

This book is printed on acid-free paper.

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 and 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 being elected president of both the American Software Testing Qualifications Board and the International Software Testing Qualifications Board twice. I spent two terms in each of these roles, and I continue to serve as a board member, as the ISTQB Governance Officer, and as a member of the ISTQB Advanced Syllabus Working Party. I look back with pride at our accomplishments so far, I look forward with pride to what we'll accomplish together in the future, and I hope this book serves as a suitable expression of the gratitude and professional pride I feel toward what we have done for the field of software testing.

Next, I'd like to thank the people who helped us create the material that grew into this book. Jamie Mitchell co-wrote the materials in this book, our Advanced Technical Test Analyst instructor-lead training course, and our Advanced Technical Test Analyst e-learning course. These materials, along with related materials in the corresponding Advanced Test Analyst book and courses, were reviewed, re-reviewed, and polished with hours of dedicated assistance by José Mata, Judy McKay, and Pat Masters. In addition, James Nazar, Corne Kruger, John Lyerly, Bhavesh Mistry, and Gyorgy Racz provided useful feedback on the first draft of this book. The task of assembling the e-learning

and live courseware from the constituent bits and pieces fell to Dena Pauletti, RBCS' extremely competent and meticulous systems engineer.

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

In the creation of the training courses and the materials that I contributed to this book, I have drawn on all the experiences I have had as an author, practitioner, consultant, and trainer. So, I have benefited from individuals too numerous to list. I thank those of you who have bought one of my previous books, for you contributed to my skills as a writer. I thank those 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 those 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 those 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 you have repaid the debt of gratitude that I owe you.

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

Finally, I thank my family, especially my wife, Laurel, and my daughters, Emma and Charlotte. Tolstoy was wrong: It is not true that all happy families are exactly the same. Our family life is quite hectic, and I know I miss a lot of it thanks to the endless travel and work demands associated with running a global testing services company and writing books. However, I've been able to enjoy seeing my daughters grow up as citizens of the world, with passports given to them before their first birthdays and full of stamps before they started losing

their baby teeth. Laurel, Emma, and Charlotte, I hope the joys of December beach sunburns in the Australian summer sun of Port Douglas, learning to ski in the Alps, hikes in the Canadian Rockies, and other experiences that frequent flier miles and an itinerant father can provide, make up in some way for the limited time we share together. I have won the lottery of life to have such a wonderful family.

Jamie Mitchell's Acknowledgements

What a long, strange trip it's been. The last 25 years has 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 they are too abundant to recall. I would like to give all of you a collective thanks; I would be poorer for not knowing you.

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, Dena, and Michelle: Your patience with me is noted and appreciated. Thanks for being there.

And finally, to my family, who have seen so much less of me over the last 25 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. Whether sitting in my lap while I was trying to learn how to test or sitting at my feet while 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. Thanks.

Contents

Rex Black's Acknowledgements v
Jamie Mitchell's Acknowledgements vii
Introduction xix

1 Test Basics 1
1.1 Introduction ... 1
1.2 Testing in the Software Lifecycle ... 2
1.3 Specific Systems ... 7
1.4 Metrics and Measurement ... 11
1.5 Ethics ... 14
1.6 Sample Exam Questions ... 16

2 Testing Processes 19
2.1 Introduction ... 19
2.2 Test Process Models ... 20
2.3 Test Planning and Control ... 21
2.4 Test Analysis and Design ... 21
2.4.1 Non-functional Test Objectives ... 23
2.4.2 Identifying and Documenting Test Conditions ... 25
2.4.3 Test Oracles ... 29
2.4.4 Standards ... 31
2.4.5 Static Tests ... 34
2.4.6 Metrics ... 35
2.5 Test Implementation and Execution ... 36
2.5.1 Test Procedure Readiness ... 37
2.5.2 Test Environment Readiness ... 39
2.5.3 Blended Test Strategies ... 41

2.5.4 Starting Test Execution 42
2.5.5 Running a Single Test Procedure 44
2.5.6 Logging Test Results 45
2.5.7 Use of Amateur Testers 47
2.5.8 Standards 48
2.5.9 Metrics 53
2.6 Evaluating Exit Criteria and Reporting 53
2.6.1 Test Suite Summary 54
2.6.2 Defect Breakdown 56
2.6.3 Confirmation Test Failure Rate 57
2.6.4 System Test Exit Review 58
2.6.5 Standards 59
2.6.6 Evaluating Exit Criteria and Reporting Exercise 60
2.6.7 System Test Exit Review 60
2.6.8 Evaluating Exit Criteria and Reporting Exercise Debrief 63
2.7 Test Closure Activities 67
2.8 Sample Exam Questions 67

3 Test Management 69

3.1 Introduction 70
3.2 Test Management Documentation 70
3.3 Test Plan Documentation Templates 71
3.4 Test Estimation 72
3.5 Scheduling and Test Planning 73
3.6 Test Progress Monitoring and Control 73
3.7 Business Value of Testing 74
3.8 Distributed, Outsourced, and Insourced Testing 74
3.9 Risk-Based Testing 75
3.9.1 Risk Management 78
3.9.2 Risk Identification 80
3.9.3 Risk Analysis or Risk Assessment 82
3.9.4 Risk Mitigation or Risk Control 84
3.9.5 An Example of Risk Identification and Assessment Results 87

3.9.6 Risk-Based Testing throughout the Lifecycle 89
3.9.7 Risk-Aware Testing Standards 90
3.9.8 Risk-Based Testing Exercise 1 92
3.9.9 Risk-Based Testing Exercise Debrief 1 93
3.9.10 Project Risk By-Products 95
3.9.11 Requirements Defect By-Products 95
3.9.12 Risk-Based Testing Exercise 2 96
3.9.13 Risk-Based Testing Exercise Debrief 2 96
3.9.14 Test Case Sequencing Guidelines 97
3.10 Failure Mode and Effects Analysis 97
3.11 Test Management Issues 98
3.12 Sample Exam Questions 98

4 Test Techniques 101

4.1 Introduction 102
4.2 Specification-Based 104
4.2.1 Equivalence Partitioning 107
4.2.1.1 Avoiding Equivalence Partitioning Errors 110
4.2.1.2 Composing Test Cases with Equivalence Partitioning 111
4.2.1.3 Equivalence Partitioning Exercise 115
4.2.1.4 Equivalence Partitioning Exercise Debrief 116
4.2.2 Boundary Value Analysis 119
4.2.2.1 Examples of Equivalence Partitioning and Boundary Values 120
4.2.2.2 Non-functional Boundaries 123
4.2.2.3 A Closer Look at Functional Boundaries 124
4.2.2.4 Integers 125
4.2.2.5 Floating Point Numbers 128
4.2.2.6 Testing Floating Point Numbers 130
4.2.2.7 How Many Boundaries? 132
4.2.2.8 Boundary Value Exercise 134
4.2.2.9 Boundary Value Exercise Debrief 135

4.2.3 Decision Tables 140
4.2.3.1 Collapsing Columns in the Table 143
4.2.3.2 Combining Decision Table Testing with Other Techniques 145
4.2.3.3 Nonexclusive Rules in Decision Tables 147
4.2.3.4 Decision Table Exercise 148
4.2.3.5 Decision Table Exercise Debrief 149
4.2.4 State-Based Testing and State Transition Diagrams 154
4.2.4.1 Superstates and Substates 161
4.2.4.2 State Transition Tables 162
4.2.4.3 Switch Coverage 166
4.2.4.4 State Testing with Other Techniques 169
4.2.4.5 State Testing Exercise 170
4.2.4.6 State Testing Exercise Debrief 172
4.2.5 Requirements-Based Testing Exercise 175
4.2.6 Requirements-Based Testing Exercise Debrief 175
4.3 Structure-Based 177
4.3.1 Control-Flow Testing 179
4.3.1.1 Building Control-Flow Graphs 180
4.3.1.2 Statement Coverage 183
4.3.1.3 Decision Coverage 188
4.3.1.4 Loop Coverage 191
4.3.1.5 Hexadecimal Converter Exercise 195
4.3.1.6 Hexadecimal Converter Exercise Debrief 197
4.3.1.7 Condition Coverage 197
4.3.1.8 Decision/Condition Coverage 200
4.3.1.9 Modified Condition/Decision Coverage (MC/DC) 201
4.3.1.10 Multiple Condition Coverage 205
4.3.1.11 Control-Flow Exercise 209
4.3.1.12 Control-Flow Exercise Debrief 210
4.3.2 Path Testing 214
4.3.2.1 LCSAJ 215

4.3.2.2 Basis Path/Cyclomatic Complexity Testing 220
4.3.2.3 Cyclomatic Complexity Exercise 225
4.3.2.4 Cyclomatic Complexity Exercise Debrief 225
4.3.3 A Final Word on Structural Testing 227
4.3.4 Structure-Based Testing Exercise 228
4.3.5 Structure-Based Testing Exercise Debrief 229
4.4 Defect- and Experience-Based 236
4.4.1 Defect Taxonomies 237
4.4.2 Error Guessing 242
4.4.3 Checklist Testing 243
4.4.4 Exploratory Testing 245
4.4.4.1 Test Charters 247
4.4.4.2 Exploratory Testing Exercise 249
4.4.4.3 Exploratory Testing Exercise Debrief 249
4.4.5 Software Attacks 252
4.4.5.1 An Example of Effective Attacks 256
4.4.5.2 Other Attacks 257
4.4.5.3 Software Attack Exercise 259
4.4.5.4 Software Attack Exercise Debrief 259
4.4.6 Specification-, Defect-, and Experience-Based Exercise 260
4.4.7 Specification-, Defect-, and Experience-Based Exercise Debrief 260
4.4.8 Common Themes 261
4.5 Static Analysis 264
4.5.1 Complexity Analysis 265
4.5.2 Code Parsing Tools 268
4.5.3 Standards and Guidelines 270
4.5.4 Data-Flow Analysis 273
4.5.5 Set-Use Pairs 275
4.5.6 Set-Use Pair Example 278
4.5.7 Data-Flow Exercise 284
4.5.8 Data-Flow Exercise Debrief 284
4.5.9 Data-Flow Strategies 285

4.5.10 Static Analysis for Integration Testing 288
4.5.11 Call-Graph Based Integration Testing 290
4.5.12 McCabe Design Predicate Approach to Integration Testing 292
4.5.13 Hex Converter Example 296
4.5.14 McCabe Design Predicate Exercise 301
4.5.15 McCabe Design Predicate Exercise Debrief 301
4.6 Dynamic Analysis 302
4.6.1 Memory Leak Detection 305
4.6.2 Wild Pointer Detection 307
4.6.3 API Misuse Detection 308
4.7 Sample Exam Questions 309

5 Tests of Software Characteristics 323

5.1 Introduction 323
5.2 Quality Attributes for Domain Testing 325
5.2.1 Accuracy 326
5.2.2 Suitability 329
5.2.3 Interoperability 330
5.2.4 Usability 331
5.2.5 Usability Test Exercise 335
5.2.6 Usability Test Exercise Debrief 335
5.3 Quality Attributes for Technical Testing 337
5.3.1 Technical Security 338
5.3.2 Security Issues 339
5.3.3 Timely Information 344
5.3.4 Reliability 349
5.3.5 Efficiency 355
5.3.6 Multiple Flavors of Efficiency Testing 357
5.3.7 Modeling the System 361
5.3.8 Efficiency Measurements 366
5.3.9 Examples of Efficiency Bugs 368
5.3.10 Exercise: Security, Reliability, and Efficiency 372

5.3.11 Exercise: Security, Reliability, and Efficiency Debrief 372
5.3.12 Maintainability 375
5.3.13 Subcharacteristics of Maintainability 379
5.3.14 Portability 386
5.3.15 Maintainability and Portability Exercise 393
5.3.16 Maintainability and Portability Exercise Debrief 393
5.4 Sample Exam Questions 396

6 Reviews 399

6.1 Introduction 399
6.2 The Principles of Reviews 403
6.3 Types of Reviews 407
6.4 Introducing Reviews 412
6.5 Success Factors for Reviews 413
6.5.1 Deutsch's Design Review Checklist 417
6.5.2 Marick's Code Review Checklist 419
6.5.3 The OpenLaszlo Code Review Checklist 422
6.6 Code Review Exercise 423
6.7 Code Review Exercise Debrief 424
6.8 Deutsch Checklist Review Exercise 429
6.9 Deutsch Checklist Review Exercise Debrief 430
6.10 Sample Exam Questions 432

7 Incident Management 435

7.1 Introduction 435
7.2 When Can a Defect Be Detected? 436
7.3 Defect Lifecycle 437
7.4 Defect Fields 445
7.5 Metrics and Incident Management 449
7.6 Communicating Incidents 450
7.7 Incident Management Exercise 451
7.8 Incident Management Exercise Debrief 452
7.9 Sample Exam Questions 454

8 Standards and Test Process Improvement 457

9 Test Techniques 459

9.1 Introduction ... 459
9.2 Test Tool Concepts ... 460
9.2.1 The Business Case for Automation ... 461
9.2.2 General Test Automation Strategies ... 466
9.2.3 An Integrated Test System Example ... 471
9.3 Test Tool Categories ... 473
9.3.1 Test Management Tools ... 474
9.3.2 Test Execution Tools ... 475
9.3.3 Debugging, Troubleshooting, Fault Seeding, and Injection Tools ... 479
9.3.4 Static and Dynamic Analysis Tools ... 480
9.3.5 Performance Testing Tools ... 483
9.3.6 Monitoring Tools ... 485
9.3.7 Web Testing Tools ... 486
9.3.8 Simulators and Emulators ... 488
9.4 Keyword-Driven Test Automation ... 489
9.4.1 Capture/Replay Exercise ... 495
9.4.2 Capture/Replay Exercise Debrief ... 495
9.4.3 Evolving from Capture/Replay ... 497
9.4.4 The Simple Framework Architecture ... 499
9.4.5 Data-Driven Architecture ... 502
9.4.6 Keyword-Driven Architecture ... 504
9.4.7 Keyword Exercise ... 511
9.4.8 Keyword Exercise Debrief ... 512
9.5 Performance Testing ... 514
9.5.1 Performance Testing Exercise ... 520
9.5.2 Performance Testing Exercise Debrief ... 521
9.6 Sample Exam Questions ... 523

10 People Skills and Team Composition 527

10.1 Introduction 528
10.2 Individual Skills 528
10.3 Test Team Dynamics 528
10.4 Fitting Testing within an Organization 529
10.5 Motivation 529
10.6 Communication 530
10.7 Sample Exam Questions 532

11 Preparing for the Exam 535

11.1 Learning Objectives 535
11.1.1 Level 1: Remember (K1) 536
11.1.2 Level 2: Understand (K2) 536
11.1.3 Level 3: Apply (K3) 537
11.1.4 Level 4: Analyze (K4) 538
11.1.5 Where Did These Levels of Learning Objectives Come From? 538
11.2 ISTQB Advanced Exams 539
11.2.1 Scenario-Based Questions 541
11.2.2 On the Evolution of the Exams 543

Appendix A – Bibliography 545

11.2.3 Advanced Syllabus Referenced Standards 545
11.2.4 Advanced Syllabus Referenced Books 545
11.2.5 Other Referenced Books 547
11.2.6 Other References 547

Appendix B – HELLOCARMS The Next Generation of Home Equity Lending 549

System Requirements Document 549
I Table of Contents 551
II Versioning 553
III Glossary 555

000 Introduction 557
001 Informal Use Case 558
003 Scope 560
004 System Business Benefits 561
010 Functional System Requirements 562
020 Reliability System Requirements 566
030 Usability System Requirements 567
040 Efficiency System Requirements 568
050 Maintainability System Requirements 570
060 Portability System Requirements 571
A Acknowledgement 573

Appendix C – Answers to Sample Questions 575

Index 577

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 earlier volume for test analysts by including details of test design using structural techniques and details about the use of dynamic analysis to monitor internal status. We assume that you know the basic concepts of test engineering, test design, test tools, testing in the software development lifecycle, and test management. You are ready to mature your level of understanding of these concepts and to apply these advanced concepts to your daily work as a test professional.

This book follows the International Software Testing Qualifications Board's (ISTQB) Advanced Level Syllabus, with a focus on the material and learning objectives for the advanced technical test analyst. As such, this book can help you prepare for the 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-lead 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, automated 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 11 chapters, addressing the following material:

1. Basic aspects of software testing
2. Testing processes
3. Test management
4. Test techniques
5. Testing of software characteristics
6. Reviews
7. Incident (defect) management
8. Standards and test process improvement
9. Test tools and automation
10. People skills (team composition)
11. Preparing for the exam

Since the structure follows the structure of the ISTQB Advanced syllabus, some of the chapters address the material in great detail because they are central to the technical test analyst role. Some of the chapters address the material in less detail because the technical test analyst need only be familiar with it. For example, we cover in detail test techniques—including highly technical techniques like structure-based testing and dynamic analysis and test automation—in this book because these are central to what a technical test analyst does, while we spend less time on test management and no time at all on test process improvement.

If you have already read *Advanced Software Testing: Volume 1*, you will notice that there is overlap in some chapters in the book, especially chapters 1, 2, 6, 7, and 10. (There is also some overlap in chapter 4, in the sections on black-box and experience-based testing.) This overlap is inherent in the structure of the ISTQB Advanced syllabus, where both learning objectives and content are common across the two analysis modules in some areas. We spent some time grappling with how to handle this commonality and decided to make this book completely free-standing. That meant that we had to include common material for those who have not read volume 1. If you have read volume 1, you may choose to skip chapters 1, 2, 6, 7, and 10, though people using this book to prepare for the Technical Test Analysis exam should read those chapters for review.

If you have also read *Advanced Software Testing: Volume 2*, which is for test managers, you'll find parallel chapters that address the material in detail but

with different emphasis. For example, technical test analysts need to know quite a bit about incident management. Technical test analysts spend a lot of time creating incident reports, and you need to know how to do that well. Test managers also need to know a lot about incident management, but they focus on how to keep incidents moving through their reporting and resolution lifecycle and how to gather metrics from such reports.

What should a 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?

- Structure the tasks defined in the test strategy in terms of technical requirements (including the coverage of technically related quality risks)
- Analyze the internal structure of the system in sufficient detail to meet the expected quality level
- Evaluate the system in terms of technical quality attributes such as performance, security, etc.
- Prepare and execute adequate activities, and report on their progress
- Conduct technical testing activities
- Provide the necessary evidence to support evaluations
- Implement the necessary tools and techniques to achieve the defined goals

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

In writing this book, 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 11 first, then read the other 10 chapters in order. If you are using this book to expand your overall understanding of testing to an advanced and highly technical level but do not intend to take the ISTQB Advanced Level Technical Test Analyst exam, then we recommend that you read chapters 1 through 10 only. If you are using this book as a reference, then feel free to read only those chapters that are of specific interest to you.

Each of the first 10 chapters is divided into sections. For the most part, we have followed the organization of the ISTQB Advanced syllabus to the point of section divisions, but subsections and sub-subsection divisions in the syllabus

might not appear. You'll also notice that each section starts with a text box describing the learning objectives for this section. If you are curious about how to interpret those K2, K3, and K4 tags in front of each learning objective, and how learning objectives work within the ISTQB syllabus, read chapter 11.

Software testing is in many ways similar to playing the piano, cooking a meal, or driving a car. How so? In each case, you can read books about these activities, but until you have practiced, you know very little about how to do it. So 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.

ISTQB Copyright

This book is based on the ISTQB Advanced Syllabus version 2007. It also references the ISTQB Foundation Syllabus version 2011. It uses terminology definitions from the ISTQB Glossary version 2.1. These three documents are copyrighted by the ISTQB and used by permission.

1 Test Basics

"Read the directions and directly you will be directed in the right direction."
A doorknob in Lewis Carroll's surreal fantasy, Alice in Wonderland.

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

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

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

1.1 Introduction

Learning objectives
Recall of content only

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

There are four major areas:

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

ISTQB Glossary

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

Many of these concepts are expanded upon in later chapters. This material expands on ideas introduced in the Foundation syllabus.

1.2 Testing in the Software Lifecycle

Learning objectives
Recall of content only

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

Proper alignment between the testing process and other processes in the lifecycle is critical for success. This is especially true at key interfaces and handoffs between testing and lifecycle activities such as these:

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

Let's look at two examples of alignment.

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

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

However, suppose the project follows an incremental lifecycle model, adhering to one of the agile methodologies like Scrum. The test team won't receive a complete set of requirements early in the project, if ever. Instead, the test team will receive requirements at the beginning of sprint, which typically lasts anywhere from two to four weeks.

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

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

ISTQB Glossary

system of systems: Multiple heterogeneous, distributed systems that are embedded in networks at multiple levels and in multiple interconnected domains, addressing large-scale interdisciplinary common problems and purposes, usually without a common management structure.

The Foundation syllabus cited four typical test levels:

- Unit or component
- Integration
- System
- Acceptance

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

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

As discussed in the Foundation syllabus, acceptance test variations include user acceptance tests and regulatory acceptance tests.

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

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

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

- Clearly defined test goals and scope
- Traceability to the test basis (if available)
- Entry and exit criteria, as appropriate both for the level and for the system lifecycle
- Test deliverables, including results reporting

- Test techniques that will be applied, as appropriate for the level, for the team and for the risks inherent in the system
- Measurements and metrics
- Test tools, where applicable and as appropriate for the level
- And, if applicable, compliance with organizational or other standards

When RBCS associates perform assessments of test teams, we often find organizations that use test levels but that perform them in isolation. Such isolation leads to inefficiencies and confusion. While these topics are discussed more in *Advanced Software Testing: Volume 2*, test analysts should keep in mind that using documents like test policies and frequent contact between test-related staff can coordinate the test levels to reduce gaps, overlap, and confusion about results.

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

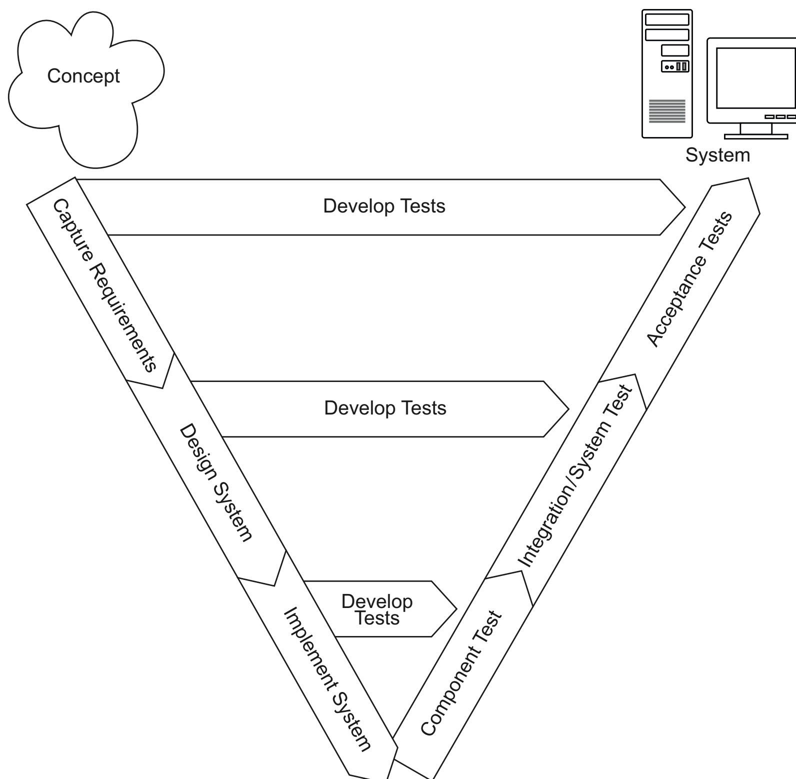


Figure 1–1 *V-model*

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

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

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

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

Evaluation of test exit criteria and reporting of test results occur throughout test execution.

Test closure activities occur after test execution is declared complete.

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

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

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

1.3 Specific Systems

Learning objectives
Recall of content only

In this section, we are going to talk about how testing affects—and is affected by—the need to test two particular types of systems. The first type is systems of systems. The second type is safety-critical systems.

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

Systems of systems projects include the following characteristics and risks:

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

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

Let's focus on the lifecycle implications for a moment.

As mentioned earlier, with systems of systems projects, we are typically going to have multiple levels of integration. First, we will have component integration for each system, and then we'll have system integration as we build the system of systems.

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

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

Because of the size and complexity of the project, it's easy for handoffs and transfers of responsibility to break down. So, we need formal information transfer among project members (especially at milestones), transfers of responsibility within the team, and handoffs. (A handoff, for those of you unfamiliar with the term, is a situation in which some work product is delivered from one group to another and the receiving group must carry out some essential set of activities with that work product.)

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

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

If you think of the fundamental test process in a system of systems project, the progress of levels is not two-dimensional. Instead, imagine a sort of pyramidal structure, as shown in figure 1-2.

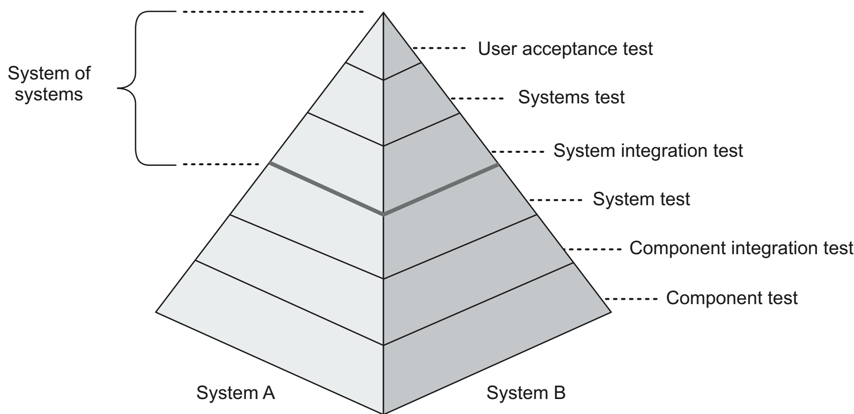


Figure 1–2 *Fundamental test process in a system of systems project*

At the base, we have component testing. A separate component test level exists for each system.

Moving up the pyramid, you have component integration testing. A separate component integration test level exists for each system.

Next, we have system testing. A separate system test level exists for each system.

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

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

Let's move on to safety-critical systems. Simply put, safety-critical systems are those systems upon which lives depend. Failure of such a system—or even temporary performance or reliability degradation or undesirable side effects as support actions are carried out—can injure or kill people or, in the case of military systems, fail to injure or kill people at a critical juncture of a battle.

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

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

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

Audits are common if regulations are imposed. Demonstrating compliance can involve tracing from the regulatory requirement through development to the test results. An outside party typically performs the audits. Therefore, establishing traceability up front and carrying out both from a people and a process point of view.

During the lifecycle—often as early as design—the project team uses safety analysis techniques to identify potential problems. As with quality risk analysis, safety analysis will identify risk items that require testing. Single points of failure are often resolved through system redundancy, and the ability of that redundancy to alleviate the single point of failure must be tested.

In some cases, safety-critical systems are complex systems or even systems of systems. In other cases, non-safety-critical components or systems are inte-

ISTQB Glossary

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

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

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

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

grated into safety-critical systems or systems of systems. For example, networking or communication equipment is not inherently a safety-critical system, but if integrated into an emergency dispatch or military system, it becomes part of a safety-critical system.

Formal quality risk management is essential in these situations. Fortunately, a number of such techniques exist, such as failure mode and effect analysis; failure mode, effect, and criticality analysis; hazard analysis; and software common cause failure analysis. We'll look at a less formal approach to quality risk analysis and management in chapter 3.

1.4 Metrics and Measurement

Learning objectives
Recall of content only

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

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

Not only must we have metrics and measurements, we also need goals. What is a "good" result for a given metric? An acceptable result? An unacceptable result? Without defined goals, successful testing is usually impossible. In fact, when we perform assessments for our clients, we more often than not find ill-defined metrics of test team effectiveness and efficiency with no goals and thus bad and unrealistic expectations (which of course aren't met). We can establish realistic goals for any given metric by establishing a baseline measure for that metric and checking current capability, comparing that baseline against industry averages, and, if appropriate, setting realistic targets for improvement to meet or exceed the industry average.

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

- Planned schedule and coverage
- Requirements and their schedule, resource, and task implications for testing
- Workload and resource usage
- Milestones and scope of testing
- Planned and actual costs
- Risks; both quality and project risks
- Defects, including total found, total fixed, current backlog, average closure periods, and configuration, subsystem, priority, or severity distribution

During test planning, we establish expectations in the form of goals for the various metrics. As part of test control, we can measure actual outcomes and trends against these goals. As part of test reporting, we can consistently explain to management various important aspects of the process, product, and project, using objective, agreed-upon metrics with realistic, achievable goals.

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

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

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

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

Victor Basili's well-known *Goal Question Metric* technique is one way to evolve meaningful metrics. (We prefer to use the word *objective* where Basili uses *goal*.) Using this technique, we proceed from the objectives of the effort—in this case, testing—to the questions we'd have to answer to know if we were achieving those objectives to, ultimately, the specific metrics.

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

Let's move on to tracking.

Since tracking is a recurring activity in a metrics program, the use of automated tool support can reduce the time required to capture, track, analyze, report, and measure the metrics.

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

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

Finally, let's look at reporting.

Most importantly, reporting of metrics and measures should enlighten management and other stakeholders, not confuse or misdirect them. In part, this is achieved through smart definition of metrics and careful tracking, but it is possible to take perfectly clear and meaningful metrics and confuse people with them through bad presentation. Edward Tufte's series of books,

ISTQB Glossary

ethics: No definition provided in the ISTQB Glossary.

starting with *The Graphical Display of Quantitative Information*, is a treasure trove of ideas about how to develop good charts and graphs for reporting purposes.[1]

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

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

1.5 Ethics

Learning objectives
Recall of content only

Many professions have ethical standards. In the context of professionalism, ethics are "rules of conduct recognized in respect to a particular class of human actions or a particular group, culture, etc."[2]

1. The three books of Tufte's that Rex has read and can strongly recommend on this topic are *The Graphical Display of Quantitative Information*, *Visual Explanations*, and *Envisioning Information* (all published by Graphics Press, Cheshire, CT).
2. Definition from dictionary.com.

Since, as a technical test analyst, you'll often have access to confidential and privileged information, ethical guidelines can help you to use that information appropriately. In addition, you should use ethical guidelines to choose the best possible behaviors and outcomes for a given situation, given your constraints. The phrase "best possible" means for everyone, not just you.

Here is an example of ethics in action. One of the authors, Rex Black, is president of three related international software testing consultancies, RBCS, RBCS AU/NZ, and Software TestWorx. He also serves on the ISTQB and ASTQB boards of directors. As such, he might have and does have insight into the direction of the ISTQB program that RBCS' competitors in the software testing consultancy business don't have.

In some cases, such as helping to develop syllabi, Rex has to make those business interests clear to people, but he is allowed to help do so. Rex helped write both the Foundation and Advanced syllabi.

In other cases, such as developing exam questions, Rex agreed, along with his colleagues on the ASTQB, that he should not participate. Direct access to the exam questions would make it all too likely that, consciously or unconsciously, RBCS would warp its training materials to "teach the exam."

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

The ISTQB Advanced syllabus makes it clear that the ISTQB expects certificate holders to adhere to the following code of ethics.

PUBLIC – Certified software testers shall act consistently with the public interest. For example, if you are working on a safety-critical system and are asked to quietly cancel some defect reports, it's an ethical problem if you do so.

CLIENT AND EMPLOYER – Certified software testers shall act in a manner that is in the best interests of their client and employer and consistent with the public interest. For example, if you know that your employer's major project is in trouble and you short-sell the stock and then leak information about the project problems to the Internet, that's a real ethical lapse—and probably a criminal one too.

PRODUCT – Certified software testers shall ensure that the deliverables they provide (on the products and systems they test) meet the highest professional

standards possible. For example, if you are working as a consultant and you leave out important details from a test plan so that the client has to hire you on the next project, that's an ethical lapse.

JUDGMENT – Certified software testers shall maintain integrity and independence in their professional judgment. For example, if a project manager asks you not to report defects in certain areas due to potential business sponsor reactions, that's a blow to your independence and an ethical failure on your part if you comply.

MANAGEMENT – Certified software test managers and leaders shall subscribe to and promote an ethical approach to the management of software testing. For example, favoring one tester over another because you would like to establish a romantic relationship with the favored tester's sister is a serious lapse of managerial ethics.

PROFESSION – Certified software testers shall advance the integrity and reputation of the profession consistent with the public interest. For example, if you have a chance to explain to your child's classmates or your spouse's colleagues what you do, be proud of it and explain the ways software testing benefits society.

COLLEAGUES – Certified software testers shall be fair to and supportive of their colleagues and promote cooperation with software developers. For example, it is unethical to manipulate test results to arrange the firing of a programmer whom you detest.

SELF – Certified software testers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession. For example, attending courses, reading books, and speaking at conferences about what you do help to advance yourself—and the profession. This is called doing well while doing good, and fortunately, it is very ethical!

1.6 Sample Exam Questions

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

1 You are working as a test analyst at a bank. At the bank, technical test analysts work closely with users during user acceptance testing. The bank has bought two financial applications as commercial off-the-shelf (COTS) software from large software vendors. Previous history with these vendors has shown that they deliver quality applications that work on their own, but this is the first time the bank will attempt to integrate applications from these two vendors. Which of the following test levels would you expect to be involved in? [Note: There might be more than one right answer.]

   A Component test

   B Component integration test

   C System integration test

   D Acceptance test

2 Which of the following is necessarily true of safety-critical systems?

   A They are composed of multiple COTS applications.

   B They are complex systems of systems.

   C They are systems upon which lives depend.

   D They are military or intelligence systems.

2 Testing Processes

Do not enter. If the fall does not kill you, the crocodile will.

A sign blocking the entrance to a parapet above a pool in the Sydney Wildlife Centre, Australia, guiding people in a safe viewing process for one of many dangerous-fauna exhibits.

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

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

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

2.1 Introduction

Learning objectives

Recall of content only

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

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

For technical test analysts, we can focus on the middle three activities in the bullet list above.

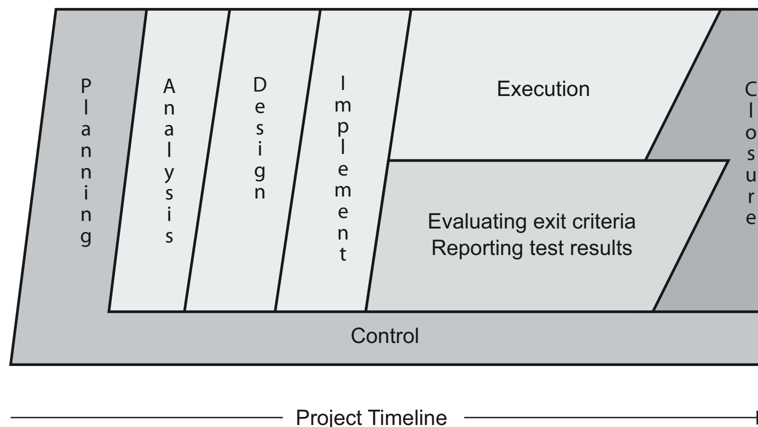


Figure 2–1 *ISTQB fundamental test process*

2.2 Test Process Models

Learning objectives

Recall of content only

The concepts in this section apply primarily for test managers. There are no learning objectives defined for technical test analysts in this section. In the course of studying for the exam, read this section in chapter 2 of the Advanced syllabus for general recall and familiarity only.

ISTQB Glossary

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

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

2.3 Test Planning and Control

Learning objectives

Recall of content only

The concepts in this section apply primarily for test managers. There are no learning objectives defined for technical test analysts in this section. In the course of studying for the exam, read this section in chapter 2 of the Advanced syllabus for general recall and familiarity only.

2.4 Test Analysis and Design

Learning objectives

(K2) Explain the stages in an application's lifecycle where non-functional tests and architecture-based tests may be applied. Explain the causes of non-functional testing taking place only in specific stages of an application's lifecycle.

(K2) Give examples of the criteria that influence the structure and level of test condition development.

(K2) Describe how test analysis and design are static testing techniques that can be used to discover defects.

(K2) Explain by giving examples the concept of test oracles and how a test oracle can be used in test specifications.

ISTQB Glossary

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

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

During the test planning activities in the test process, test leads and test managers work with project stakeholders to identify test objectives. In the IEEE 829 test plan template—which was introduced at the Foundation Level and which we'll review later in this book—the lead or manager can document these in the section "Features to be Tested."

The test objectives are a major deliverable for technical test analysts because without them, we wouldn't know what to test. During test analysis and design activities, we use these test objectives as our guide to carry out two main subactivities:

- Identify and refine the test conditions for each test objective
- Create test cases that exercise the identified test conditions

However, test objectives are not enough by themselves. We not only need to know what to test, but in what order and how much. Because of time constraints, the desire to test the most important areas first, and the need to expend our test effort in the most effective and efficient manner possible, we need to prioritize the test conditions.

When following a risk-based testing strategy—which we'll discuss in detail in chapter 3—the test conditions are quality risk items identified during quality risk analysis. The assignment of priority for each test condition usually involves determining the likelihood and impact associated with each quality risk item; i.e., we assess the level of risk for each risk item. The priority determines the allocation of test effort (throughout the test process) and the order of design, implementation, and execution of the related tests.

ISTQB Glossary

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

Throughout the process, the specific test conditions and their associated priorities can change as the needs—and our understanding of the needs—of the project and project stakeholders evolve.

This prioritization, use of prioritization, and reprioritization occurs regularly in the test process. It starts during risk analysis and test planning, of course. It continues throughout the process, from analysis and design to implementation and execution. It influences evaluation of exit criteria and reporting of test results.

2.4.1 Non-functional Test Objectives

Before we get deeper into this process, let's look at an example of non-functional test objectives.

First, it's important to remember that non-functional test objectives can apply to any test level and exist throughout the lifecycle. Too often major non-functional test objectives are not addressed until the very end of the project, resulting in much wailing and gnashing of teeth when show-stopping failures are found.

Consider a video game as an example. For a video game, the ability to interact with the screen in real time, with no perceptible delays, is a key non-functional test objective. Every subsystem of the game must interact and perform efficiently to achieve this goal.

To be smart about this testing, execution efficiency to enable timely processing should be tested at the unit, integration, system, and acceptance levels. Finding a serious bottleneck during system test would affect the schedule, and that's not good for a consumer product like a game—or any other kind of software or system, for that matter.

ISTQB Glossary

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

Furthermore, why wait until test execution starts at any level, early or late? Instead, start with reviews of requirements specifications, design specifications, and code to assess this function as well.

Many times, non-functional quality characteristics can be quantified; in this case, we might have actual performance requirements which we can test throughout the various test levels. For example, suppose that key events must be processed within 3 milliseconds of input to a specific component to be able to meet performance standards; we can test if the component actually meets that measure. In other cases, the requirements might be implicit rather than explicit: the system must be "fast enough."

Some types of non-functional testing should clearly be performed as early as possible. As an example, for many projects we have worked on, performance testing was only done late in system testing. The thought was that it could not be done earlier because the functional testing had not yet been done, so end-to-end testing would not be possible. Then, when serious bottlenecks resulting in extremely slow performance were discovered in the system, the release schedule was severely impacted.

Other projects targeted performance testing as critical. At the unit and component testing level, measurements were made as to time required for processing through the objects. At integration testing, subsystems were benchmarked to make sure they could perform in an optimal way. As system testing started, performance testing was a crucial piece of the planning and started as soon as some functionality was available, even though the system was not yet feature complete.

All of this takes time and resources, planning and effort. Some performance tools are not really useful too early in the process, but measurements can still be taken using simpler tools.

Other non-functional testing may not make sense until late in the Software Development Life Cycle (SDLC). While error and exception handling can be tested in unit and integration testing, full blown recoverability testing

really makes sense only in the system, acceptance, and system integration phases when the response of the entire system can be measured.

2.4.2 Identifying and Documenting Test Conditions

To identify test conditions, we can perform analysis of the test basis, the test objectives, the quality risks, and so forth using any and all information inputs and sources we have available. For analytical risk-based testing strategies, we'll cover exactly how this works in chapter 3.

If you're not using analytical risk-based testing, then you'll need to select the specific inputs and techniques according to the test strategy or strategies you are following. Those strategies, inputs, and techniques should align with the test plan or plans, of course, as well as with any broader test policies or test handbooks.

Now, in this book, we're concerned primarily with the technical test analyst role. So we address both functional tests (especially from a technical perspective) and non-functional tests. The analysis activities can and should identify functional and non-functional test conditions. We should consider the level and structure of the test conditions for use in addressing functional and non-functional characteristics of the test items.

There are two important choices when identifying and documenting test conditions:

- The structure of the documentation for the test conditions
- The level of detail we need to describe the test conditions in the documentation

There are many common ways to determine the level of detail and structure of the test conditions.

One is to work in parallel with the test basis documents. For example, if you have a marketing requirements document and a system requirements document in your organization, the former is usually high level and the latter is low level. You can use the marketing requirements document to generate the high-level test conditions and then use the system requirements document to elaborate one or more low-level test conditions underneath each high-level test condition.

Another approach is often used with quality risk analysis (sometimes called product risk analysis). In this approach, we can outline the key features and quality characteristics at a high level. We can then identify one or more detailed quality risk items for each feature or characteristic. These quality risk items are thus the test conditions.

Another approach, if you have only detailed requirements, is to go directly to the low-level requirements. In this case, traceability from the detailed test conditions to the requirements (which impose the structure) is needed for management reporting and to document what the test is to establish.

Yet another approach is to identify high-level test conditions only, sometimes without any formal test bases. For example, in exploratory testing some advocate the documentation of test conditions in the form of test charters. At that point, there is little to no additional detail created for the unscripted or barely scripted tests.

Again, it's important to remember that the chosen level of detail and the structure must align with the test strategy or strategies, and those strategies should align with the test plan or plans, of course, as well as with any broader test policies or test handbooks.

Also, remember that it's easy to capture traceability information while you're deriving test conditions from test basis documents like requirements, designs, use cases, user manuals, and so forth. It's much harder to re-create that information later by inspection of test cases.

Let's look at an example of applying a risk-based testing strategy to this step of identifying test conditions.

Suppose you are working on an online banking system project. During a risk analysis session, system response time, a key aspect of system performance, is identified as a high-risk area for the online banking system. Several different failures are possible, each with its own likelihood and impact.

So, discussions with the stakeholders lead us to elaborate the system performance risk area, identifying three specific quality risk items:

- Slow response time during login
- Slow response time during queries
- Slow response time during a transfer transaction

ISTQB Glossary

test design: (1) See *test design specification*. (2) The process of transforming general testing objectives into tangible test conditions and test cases.

test design specification: A document specifying the test conditions (coverage items) for a test item and the detailed test approach and identifying the associated high-level test cases.

high-level test case: A test case without concrete (implementation-level) values for input data and expected results. Logical operators are used; instances of the actual values are not yet defined and/or available.

low-level test case: A test case with concrete (implementation-level) values for input data and expected results. Logical operators from high-level test cases are replaced by actual values that correspond to the objectives of the logical operators.

At this point, the level of detail is specific enough that the risk analysis team can assign specific likelihood and impact ratings for each risk item.

Now that we have test conditions, the next step is usually to elaborate those into test cases. We say “usually” because some test strategies, like the reactive ones discussed in the Foundation syllabus and in this book in chapter 4, don’t always use written test cases. For the moment, let’s assume that we want to specify test cases that are repeatable, verifiable, and traceable back to requirements, quality risk, or whatever else our tests are based on.

If we are going to create test cases, then, for a given test condition—or two or more related test conditions—we can apply various test design techniques to create test cases. These techniques are covered in chapter 4. Keep in mind that you can and should blend techniques in a single test case.

We mentioned traceability to the requirements, quality risks, and other test bases. Some of those other test bases for technical test analysts can include designs (high or low level), architectural documents, class diagrams, object hierarchies, and even the code itself. We can capture traceability directly, by relating the test case to the test basis element or elements that gave rise to the test conditions from which we created the test case. Alternatively, we can relate the test case to the test conditions, which are in turn related to the test basis elements.

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

As with test conditions, we'll need to select a level of detail and structure for our test cases. It's important to remember that the chosen level of detail and the structure must align with the test strategy or strategies. Those strategies should align with the test plan or plans, of course, as well as with any broader test policies or test handbooks.

So, can we say anything else about the test design process? Well, the specific process of test design depends on the technique. However, it typically involves defining the following:

- Preconditions
- Test environment requirements
- Test inputs and other test data requirements
- Expected results
- Postconditions

Defining the expected result of a test can be tricky, especially as expected results are not only screen outputs, but also data and environmental post conditions. Solving this problem requires that we have what's called a test oracle, which we'll look at in a moment.

First, though, notice the mention of test environment requirements in the preceding bullet list. This is an area of fuzziness in the ISTQB fundamental test process. Where is the line between test design and test implementation, exactly?

The Advanced syllabus says, "[D]uring test design the required detailed test infrastructure requirements may be defined, although in practice these may not be finalized until test implementation." Okay, but maybe we're doing some implementation as part of the design? Can't the two overlap? To us, trying to draw sharp distinctions results in many questions along the lines of, How many angels can dance on the head of a pin?

Whatever we call defining test environments and infrastructures—design, implementation, environment setup, or some other name—it is vital to remember that testing involves more than just the test objects and the testware. There is a test environment, and this isn't just hardware. It includes rooms, equipment, personnel, software, tools, peripherals, communications equipment, user authorizations, and all other items required to run the tests.

2.4.3 Test Oracles

Okay, let's look at what test oracles are and what oracle-related problems the technical test analyst faces.

A test oracle is a source we use to determine the expected results of a test. We can compare these expected results with the actual results when we run a test. Sometimes the oracle is the existing system. Sometimes it's a user manual. Sometimes it's an individual's specialized knowledge. Rex usually says that we should never use the code itself as an oracle, even for structural testing, because that's simply testing that the compiler, operating system, and hardware work. Jamie feels that the code can serve as a useful partial oracle, saying it doesn't hurt to consider it, though he agrees with Rex that it should not serve as the sole oracle.

So, what is the oracle problem? Well, if you haven't experienced this firsthand, ask yourself, in general, how we know what "correct results" are for a test? The difficulty of determining the correct result is the oracle problem.

If you've just entered the workforce from the ivory towers of academia, you might have learned about perfect software engineering projects. You may have heard stories about detailed, clear, and consistent test bases like requirements and design specifications that define all expected results. Those stories were myths.

In the real world, on real projects, test basis documents like requirements are vague. Two documents, such as a marketing requirements document and a system requirements document, will often contradict each other. These documents may have gaps, omitting any discussion of important characteristics of the product—especially non-functional characteristics, and especially usability and user interface characteristics.

Sometimes these documents are missing entirely. Sometimes they exist but are so superficial as to be useless. One of our clients showed Rex a hand-

written scrawl on a letter-size piece of paper, complete with crude illustrations, which was all the test team had received by way of requirements on a project that involved 100 or so person-months of effort. We have both worked on projects where even that would be an improvement over what we actually received!

When test basis documents are delivered, they are often delivered late, often too late to wait for them to be done before we begin test design (at least if we want to finish test design before we start test execution). Even with the best intentions on the part of business analysts, sales and marketing staff, and users, test basis documents won't be perfect. Real-world applications are complex and not entirely amenable to complete, unambiguous specification.

So we have to augment the written test basis documents we receive with tester expertise or access to expertise, along with judgment and professional pessimism. Using all available oracles—written and mental, provided and derived—the tester can define expected results before and during test execution.

Since we've been talking a lot about requirements, you might assume that the oracle problem applies only to high-level test levels like system test and acceptance test. Nope. The oracle problem—and its solutions—apply to all test levels. The test bases will vary from one level to another, though. Higher test levels like user acceptance test and system test rely more on requirements specification, use cases, and defined business processes. Lower test levels like component test and integration test rely more on low-level design specification

While this is a hassle, remember that you must solve the oracle problem in your testing. If you run tests with no way to evaluate the results, you are wasting your time. You will provide low, zero, or negative value to the team. Such testing generates false positives and false negatives. It distracts the team with spurious results of all kinds. It creates false confidence in the system.

By the way, as for our sarcastic aside about the "ivory tower of academia" a moment ago, let us mention that, when Rex studied computer science at UCLA quite a few years ago, one of his software engineering professors told him about this problem right from the start. One of Jamie's professors at Lehigh said that that complete requirements were more mythological than unicorns. Neither of us could say we weren't warned!

Let's look at an example of a test oracle, from the real world.

Rex and his associates worked on a project to develop a banking application to replace a legacy system. There were two test oracles. One was the requirements specification, such as it was. The other was the legacy system. They faced two challenges.

For one thing, the requirements were vague. The original concept of the project, from the vendor's side, was "Give the customer whatever the customer wants," which they then realized was a good way to go bankrupt given the indecisive and conflicting ideas about what the system should do among the customer's users. The requirements were the outcome of a belated effort to put more structure around the project.

For another thing, sometimes the new system differed from the legacy system in minor ways. In one infamous situation, there was a single bug report that they opened, then deferred, then reopened, then deferred again, at least four or five times. It described situations where the monthly payment varied by $0.01.

The absence of any reliable, authoritative, consistent set of oracles led to a lot of "bug report ping-pong." They also had bug report prioritization issues as people argued over whether some problems were problems at all. They had high rates of false positives and negatives. The entire team—including the test team—was frustrated. So, you can see that the oracle problem is not some abstract concept; it has real-world consequences.

2.4.4 Standards

At this point, let's review some standards from the Foundation that will be useful in test analysis and design.

First, let's look at two documentation templates you can use to capture information as you analyze and design your tests, assuming you intend to document what you are doing, which is usually true. The first is the IEEE 829 test design specification.

Remember from the Foundation course that a test condition is an item or event of a component or system that could be verified by one or more test cases, e.g., a function, transaction, feature, quality attribute, identified risk, or structural element. The IEEE 829 test design specification describes a condition, feature or small set of interrelated features to be tested and the set of tests that cover them at a very high or logical level. The number of tests

required should be commensurate with the risks we are trying to mitigate (as reflected in the pass/fail criteria.) The design specification template includes the following sections:

- Test design specification identifier (following whatever standard your company uses for document identification)
- Features to be tested (in this test suite)
- Approach refinements (specific techniques, tools, etc.)
- Test identification (tracing to test cases in suites)
- Feature pass/fail criteria (e.g., how we intend to determine whether a feature works, such as via a test oracle, a test basis document, or a legacy system)

The collection of test cases outlined in the test design specification is often called a test suite.

The sequencing of test suites and cases within suites is often driven by risk and business priority. Of course, project constraints, resources, and progress must affect the sequencing of test suites.

Next comes the IEEE 829 test case specification. A test case specification describes the details of a test case. This template includes the following sections:

- Test case specification identifier
- Test items (what is to be delivered and tested)
- Input specifications (user inputs, files, etc.)
- Output specifications (expected results, including screens, files, timing, behaviors of various sorts, etc.)
- Environmental needs (hardware, software, people, props, and so forth)
- Special procedural requirements (operator intervention, permissions, etc.)
- Intercase dependencies (if needed to set up preconditions)

While this template defines a standard for contents, many other attributes of a test case are left as open questions. In practice, test cases vary significantly in effort, duration, and number of test conditions covered.

We'll return to the IEEE 829 standard again in the next section. However, let us also review another related topic from the Foundation syllabus, on the matter of documentation.

In the real world, the extent of test documentation varies considerably. It would be hard to list all the different reasons for this variance, but they include the following:

- Risks to the project created by documenting or not documenting.
- How much value, if any, the test documentation creates—and is meant to create.
- Any standards that are or should be followed, including the possibility of an audit to ensure compliance with those standards.
- The software development lifecycle model used. Advocates of agile approaches try to minimize documentation by ensuring close and frequent team communication.
- The extent to which we must provide traceability from the test basis to the test cases.

The key idea here is to remember to keep an open mind and a clear head when deciding how much to document.

Now, since we focus on both functional and non-functional characteristics as part of this technical test analyst volume, let's review the ISO 9126 standard.

The ISO 9126 quality standard for software defines six software quality characteristics: functionality, reliability, usability, efficiency, maintainability, and portability. Each characteristic has three or more subcharacteristics, as shown in figure 2-2.

Tests that address functionality and its subcharacteristics are functional tests. These were the main topics in the first volume of this series, for test analysts. We will revisit them here, but primarily from a technical perspective. Tests that address the other five characteristics and their subcharacteristics are non-functional tests. These are among the main topics for this book. Finally, keep in mind that, when you are testing hardware/software systems, additional quality characteristics can and will apply.

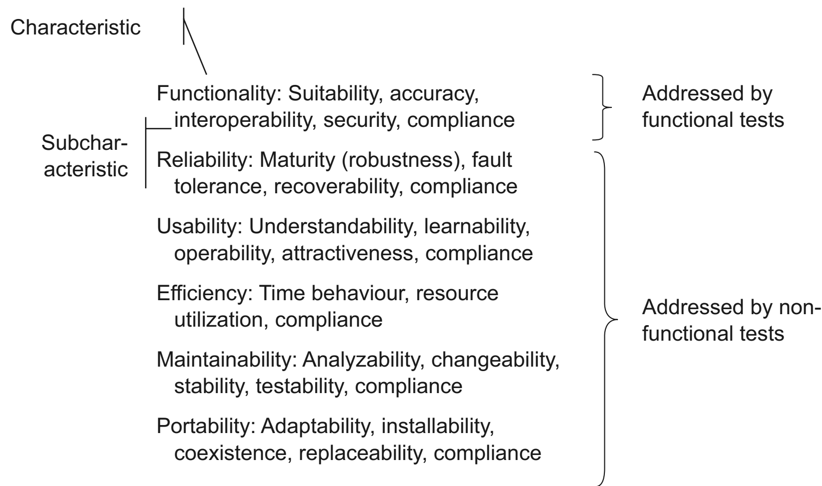


Figure 2–2 *ISO 9126 quality standard*

2.4.5 Static Tests

Now, let's review three important ideas from the Foundation syllabus. One is the value of static testing early in the lifecycle to catch defects when they are cheap and easy to fix. The next is the preventive role testing can play when involved early in the lifecycle. The last is that testing should be involved early in the project. These three ideas are related because technical test analysis and design is a form of static testing; it is synergistic with other forms of static testing, and we can exploit that synergy only if we are involved at the right time.

Notice that, depending on when the analysis and design work is done, you could possibly define test conditions and test cases in parallel with reviews and static analyses of the test basis. In fact, you could prepare for a requirements review meeting by doing test analysis and design on the requirements. Test analysis and design can serve as a structured, failure-focused static test of a requirements specification that generates useful inputs to a requirements review meeting.

Of course, we should also take advantage of the ideas of static testing, and early involvement if we can, to have test and non-test stakeholders participate in reviews of various test work products, including risk analyses, test designs,

test cases, and test plans. We should also use appropriate static analysis techniques on these work products.

Let's look at an example of how test analysis can serve as a static test. Suppose you are following an analytical risk-based testing strategy. If so, then in addition to quality risk items—which are the test conditions—a typical quality risk analysis session can provide other useful deliverables.

We refer to these additional useful deliverables as by-products, along the lines of industrial by-products, in that they are generated by the way as you create the target work product, which in this case is a quality risk analysis document. These by-products are generated when you and the other participants in the quality risk analysis process notice aspects of the project you haven't considered before.

These by-products include the following:

- Project risks—things that could happen and endanger the success of the project
- Identification of defects in the requirements specification, design specification, or other documents used as inputs into the quality risk analysis
- A list of implementation assumptions and simplifications, which can improve the design as well as set up checkpoints you can use to ensure that your risk analysis is aligned with actual implementation later

By directing these by-products to the appropriate members of the project team, you can prevent defects from escaping to later stages of the software lifecycle. That's always a good thing.

2.4.6 Metrics

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

- Percentage of requirements or quality (product) risks covered by test conditions
- Percentage of test conditions covered by test cases
- Number of defects found during test analysis and design

We can track test analysis and design tasks against a work breakdown structure, which is useful in determining whether we are proceeding according to the estimate and schedule.

2.5 Test Implementation and Execution

Learning objectives

(K2) Describe the preconditions for test execution, including testware, test environment, configuration management, and defect management.

Test implementation includes all the remaining tasks necessary to enable test case execution to begin. At this point, remember, we have done our analysis and design work, so what remains?

For one thing, if we intend to use explicitly specified test procedures—rather than relying on the tester's knowledge of the system—we'll need to organize the test cases into test procedures (or, if using automation, test scripts). When we say "organize the test cases," we mean, at the very least, document the steps to carry out the test. How much detail do we put in these procedures? Well, the same considerations that lead to more (or less) detail at the test condition and test case level would apply here. For example, if a regulatory standard like the United States Federal Aviation Administration's DO-178B applies, that's going to require a high level of detail.

Since testing frequently requires test data for both inputs and the test environment itself, we need to make sure that data is available now. In addition, we must set up the test environments. Are both the test data and the test environments in a state such that we can use them for testing now? If not, we must resolve that problem before test execution starts. In some cases test data require the use of data generation tools or production data. Ensuring proper test environment configuration can require the use of configuration management tools.

With the test procedures in hand, we need to put together a test execution schedule. Who is to run the tests? In what order should they run them? What environments are needed for which tests? When should we run the

ISTQB Glossary

test procedure: See *test procedure specification*.

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

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

automated tests? If automated tests run in the same environment as manual tests, how do we schedule the tests to prevent undesirable interactions between the automated and manual tests? We need to answer these questions.

Finally, since we're about to start test execution, we need to check whether all explicit and implicit entry criteria are met. If not, we need to work with project stakeholders to make sure they are met before the scheduled test execution start date.

Now, keep in mind that you should prioritize and schedule the test procedures to ensure that you achieve the objectives in the test strategy in the most efficient way. For example, in risk-based testing, we usually try to run tests in risk priority order. Of course, real-world constraints like availability of test configurations can change that order. Efficiency considerations like the amount of data or environment restoration that must happen after a test is over can change that order too.

Let's look more closely at two key areas, readiness of test procedures and readiness of test environments.

2.5.1 Test Procedure Readiness

Are the test procedures ready to run? Let's examine some of the issues we need to address before we know the answer.

As mentioned earlier, we must have established clear sequencing for the test procedures. This includes identifying who is to run the test procedure, when, in what test environment, with what data.

We have to evaluate constraints that might require tests to run in a particular order. Suppose we have a sequence of test procedures that together make up an end-to-end workflow? There are probably business rules that govern the order in which those test procedures must run.

So, based on all the practical considerations as well as the theoretical ideal of test procedure order—from most important to least important—we need to finalize the order of the test procedures. That includes confirming that order with the test team and other stakeholders. In the process of confirming the order of test procedures, you might find that the order you think you should follow is in fact impossible or perhaps unacceptably less efficient than some other possible sequencing.

We also might have to take steps to enable test automation. Of course, we say “might have to take steps” rather than “must take steps” because not all test efforts involve automation. However, as a technical test analyst, implementing automated testing is a key responsibility, one which we’ll discuss in detail later in this book.

If some tests are automated, we’ll have to determine how those fit into the test sequence. It’s very easy for automated tests, if run in the same environment as manual tests, to damage or corrupt test data, sometimes in a way that causes both the manual and automated tests to generate huge numbers of false positives and false negatives. Guess what? That means you get to run the tests all over again. We don’t want that!

Now, the Advanced syllabus says that we will create the test harness and test scripts during test implementation. Well, that’s theoretically true, but as a practical matter we really need the test harness ready weeks, if not months, before we start to use it to automate test scripts.

We definitely need to know all the test procedure dependencies. If we find that there are reasons—due to these dependencies—we can’t run the test procedures in the sequence we established earlier, we have two choices: One, we can change the sequence to fit the various obstacles we have discovered. Or, two, we can remove the obstacles.

Let’s look more closely at two very common categories of test procedure dependencies—and thus obstacles.

The first is the test environment. You need to know what is required for each test procedure. Now, check to see if that environment will be available during the time you have that test procedure scheduled to run. Notice that “available” means not only is the test environment configured, but also no other test procedure—or any other test activity for that matter—that would interfere with the test procedure under consideration is scheduled to use that test environment during the same period of time.

The interference question is usually where the obstacles emerge. However, for complex and evolving test environments, the mere configuration of the test environment can become a problem. Rex worked on a project a while back that was so complex that he had to construct a special database to track, report, and manage the relationships between test procedures and the test environments they required.

The second category of test procedure dependencies is the test data. You need to know what data each test procedure requires. Now, similar to the process before, check to see if that data will be available during the time you have that test procedure scheduled to run. As before, "available" means not only is the test data created, but also no other test procedure—or any other test activity for that matter—that would interfere with the viability and accessibility of the data is scheduled to use that test data during the same period of time.

With test data, interference is again often a large issue. We had a client who tried to run manual tests during the day and automated tests overnight. This resulted in lots of problems until a process was evolved to properly restore the data at the handover points between manual testing and automated testing (at the end of the day) and between automated testing and manual testing (at the start of the day).

2.5.2 Test Environment Readiness

Are the test environments ready to use? Let's examine some of the issues we need to address before we know the answer.

First, let's make clear the importance of a properly configured test environment. If we run the test procedures perfectly but use an improperly configured test environment, we obtain useless test results. Specifically, we get many false positives. A false positive in software testing is analogous to one in medicine—a test that should have passed instead fails, leading to wasted time analyzing "defects" that turn out to be test environment problems. Often the false positives are so large in number that we also get false negatives. This happens when a test that should have failed instead passes, often in this case because we didn't see it hiding among the false positives. The overall outcomes are low defect detection effectiveness; high field or production failure rates; high defect report rejection rates; a lot of wasted time for testers, managers, and developers; and a severe loss of credibility for the test team. Obviously those are all very bad outcomes.

ISTQB Glossary

false negative: See *false-pass result*.

false-pass result: A test result which fails to identify the presence of a defect that is actually present in the test object.

false positive: See *false-fail result*.

false-fail result: A test result in which a defect is reported although no such defect actually exists in the test object.

So, we have to ensure properly configured test environments. Now, in the ISTQB fundamental test process, implementation is the point where this happens. As with automation, though, we feel this is probably too late, at least if implementation is an activity that starts after analysis and design. If, instead, implementation of the test environment is a subset of the overall implementation activity and can start as soon as the test plan is done, then we are in better shape.

What is a properly configured test environment and what does it do for us?

For one thing, a properly configured test environment enables finding defects under the test conditions we intend to run. For example, if we want to test for performance, it allows us to find unexpected bottlenecks that would slow down the system.

For another thing, a properly configured test environment operates normally when failures are not occurring. In other words, it doesn't generate many false positives.

Additionally, at higher levels of testing such as system test and system integration test, a properly configured test environment replicates the production or end-user environment. Many defects, especially non-functional defects like performance and reliability problems, are hard if not impossible to find in scaled-down environments.

There are some other things we need for a properly configured test environment. We'll need someone to set up and support the environment. For complex environments, this person is usually someone outside the test team. (Jamie and Rex both would prefer the situation in which the person is part of the test team, due to the problems of environment support availability that seem to arise with reliance on external resources, but the preferences of the test manager don't

> **ISTQB Glossary**
>
> **test log:** A chronological record of relevant details about the execution of tests.
>
> test logging: The process of recording information about tests executed into a test log.

always result in the reallocation of the individual.) We also need to make sure someone—perhaps a tester, perhaps someone else—has loaded the testware, test support tools, and associated processes on the test environment.

Test support tools include, at the least, configuration management, incident management, test logging, and test management. Also, you'll need procedures to gather data for exit criteria evaluation and test results reporting. Ideally, your test management system will handle some of that for you.

2.5.3 Blended Test Strategies

It is often a good idea to use a blend of test strategies, leading to a balanced test approach throughout testing, including during test implementation. For example, when RBCS associates run test projects, we typically blend analytical risk-based test strategies with analytical requirements-based test strategies and dynamic test strategies (also referred to as reactive test strategies). We reserve some percentage (often 10 to 20 percent) of the test execution effort for testing that does not follow predetermined scripts.

Analytical strategies follow the ISTQB fundamental test process nicely, with work products produced along the way. However, the risk with blended strategies is that the reactive portion can get out of control. Testing without scripts should not be ad hoc or aimless. Such tests are unpredictable in duration and coverage.

Some techniques like session-based test management, which is covered in the companion volume on test management, can help deal with that inherent control problem in reactive strategies. In addition, we can use experience-based test techniques such as attacks, error guessing, and exploratory testing to structure reactive test strategies. We'll discuss these topics further in chapter 4.

The common trait of a reactive test strategy is that most of the testing involves reacting to the actual system presented to us. This means that test analysis, test design, and test implementation occur primarily during test execution. In other words, reactive test strategies allow—indeed, require—that the results of each test influence the analysis, design, and implementation of the subsequent tests.

As discussed in the Foundation syllabus, these reactive strategies are lightweight in terms of total effort both before and during test execution. Experience-based test techniques are often efficient bug finders, sometimes 5 or 10 times more efficient than scripted techniques. However, being experience based, naturally enough, they require expert testers. As mentioned earlier, reactive test strategies result in test execution periods that are sometimes unpredictable in duration. Their lightweight nature means they don't provide much coverage information and are difficult to repeat for regression testing. Some claim that tools can address this coverage and repeatability problem, but we've never seen that work in actual practice.

That said, when reactive test strategies are blended with analytical test strategies, they tend to balance each other's weak spots. This is analogous to blended scotch whiskey. Blended scotch whiskey consists of malt whiskey—either a single malt or more frequently a combination of various malt whiskeys—blended with grain alcohol (basically, vodka). It's hard to imagine two liquors more different than vodka and single malt scotch, but together they produce a liquor that many people find much easier to drink and enjoy than the more assertive, sometimes almost medicinal single malts.

2.5.4 Starting Test Execution

Before we start test execution, it's a best practice to measure readiness based on some predefined preconditions, often called entry criteria. Entry criteria were discussed in the Foundation syllabus, in the chapter on test management.

Often we will have formal entry criteria for test execution. Consider the following entry criteria, taken from an actual project:

1. Bug tracking and test tracking systems are in place.
2. All components are under formal, automated configuration management and release management control.

3. The operations team has configured the system test server environment, including all target hardware components and subsystems. The test team has been provided with appropriate access to these systems.

These were extracted from the entry criteria section of the test plan for that project. They focus on three typical areas that can adversely affect the value of testing if unready: the test environment (as discussed earlier), configuration management, and defect management.

On the other hand, sometimes projects have less-formal entry criteria for execution. For example, test teams often simply assume that the test cases and test data will be ready, and there is no explicit measurement of them.

Whether the entry criteria are formal or informal, we need to determine if we're ready to start test execution. To do so, we need the delivery of a testable test object or objects and the satisfaction (or waiver) of entry criteria.

Of course, this presumes that the entry criteria alone are enough to ensure that the various necessary implementation tasks discussed earlier in this section are complete. If not, then we have to go back and check those issues of test data, test environments, test dependencies, and so forth.

Now, during test execution, people will run the manual test cases via the test procedures. To execute a test procedure to completion, we'd expect that at least two things had happened. First, we covered all of the test conditions or quality risk items traceable to the test procedure. Second, we carried out all of the steps of the test procedure.

You might ask, "How could I carry out the test procedure without covering all the risks or conditions?" In some cases, the test procedure is written at a high level. In that case, you would need to understand what the test was about and augment the written test procedure with on-the-fly details that ensure that you cover the right areas.

You might also ask, "How could I cover all the risks and conditions without carrying out the entire test procedure?" Some steps of a test procedure serve to enable testing rather than covering conditions or risks. For example, some steps set up data or other preconditions, some steps capture logging information, and some steps restore the system to a known good state at the end.

A third kind of activity can apply during manual test execution. We can incorporate some degree of reactive testing into the procedures. One way to accomplish this is to leave the procedures somewhat vague and to tell the tester to select their favorite way of carrying out a certain task. Another way is to tell the testers, as Rex often does, that a test script is a road map to interesting places and, when they get somewhere interesting, they should stop and look around. This has the effect of giving them permission to transcend, to go beyond, the scripts. We have found it very effective.

Finally, during execution, tools will run any existing automated tests. These tools follow the defined scripts without deviation. That can seem like an unalloyed "good thing" at first. However, if we did not design the scripts properly, that can mean that the script can get out of sync with the system under test and generate a bunch of false positives. We'll talk more about that problem—and how to solve it—in chapter 9.

2.5.5 Running a Single Test Procedure

Let's zoom in on the act of a tester running a single test procedure. After the logistical issues of initial setup are handled, the tester starts running the specific steps of the test. These yield the actual results.

Now we have come to the heart of test execution. We compare actual results with expected results. This is indeed the moment when testing either adds value or removes value from the project. Everything up to this point—all of the work designing and implementing our tests—was about getting us to this point. Everything after this point is about using the value this comparison has delivered. Since running a test procedure is so critical, so central to good testing, attention and focus on your part is essential at this moment.

But what if we notice a mismatch between the expected results and the actual results? The ISTQB glossary refers to each difference between the expected results and the actual results as an anomaly. There can be multiple differences, and thus multiple anomalies, in a mismatch. When we observe an anomaly, we have an incident.

Some incidents are failures. A failure occurs when the system misbehaves due to one or more defects. This is the ideal situation when an incident has occurred. If we are looking at a failure—a symptom of a true defect—we should start to gather data to help the developer resolve the defect. We'll talk more about incident reporting and management in detail in chapter 7.

ISTQB Glossary

anomaly: Any condition that deviates from expectation based on requirements specifications, design documents, user documents, standards, etc. or from someone's perception or experience. Anomalies may be found during, but not limited to, reviewing, testing, analysis, compilation, or use of software products or applicable documentation.

incident: Any event occurring that requires investigation.

failure: Deviation of the component or system from its expected delivery, service, or result.

Some incidents are not failures but rather are false positives. False positives occur when the expected and actual results don't match due to bad test specifications, invalid test data, incorrectly configured test environments, a simple mistake on the part of the person running the test, and so forth.

If we can catch a false positive right away, the moment it happens, the damage is limited. The tester should fix the test, which might involve some configuration management work if the tests are checked into a repository. The tester should then rerun the test. Thus, the damage done was limited to the tester's wasted time along with the possible impact of that lost time on the schedule plus the time needed to fix the test plus the time needed to rerun the test.

All of those activities, all of that lost time, and the impact on the schedule would have happened even if the tester had simply assumed the failure was valid and reported it as such. It just would have happened later, after an additional loss of time on the part of managers, other testers, developers, and so forth.

Here's a cautionary note on these false positives too. Just because a test has never yielded a false positive before, in all the times it's been run before, doesn't mean you're not looking at one this time. Changes in the test basis, the proper expected results, the test object, and so forth can obsolete or invalidate a test specification.

2.5.6 Logging Test Results

Most testers like to run tests—at least the first few times they run them—but sometimes they don't always like to log results. "Paperwork!" they snort. "Bureaucracy and red tape!" they protest.

If you are one of those testers, get over it. We said previously that all the planning, analysis, design, and implementation was about getting to the point of running a test procedure and comparing actual and expected results. We then said that everything after that point is about using the value the comparison delivered. Well, you can't use the value if you don't capture it, and the test logs are about capturing the value.

So remember that, as testers run tests, testers log results. Failure to log results means either doing the test over (most likely) or losing the value of running the tests. When you do the test over, that is pure waste, a loss of your time running the test. Since test execution is usually on the critical path for project completion, that waste puts the planned project end date at risk. People don't like that much.

A side note here, before we move on. We mentioned reactive test strategies and the problems they have with coverage earlier. Note that, with adequate logging, while you can't ascertain reactive test coverage in advance, at least you can capture it afterward. So again, log your results, both for scripted and unscripted tests.

During test execution, there are many moving parts. The test cases might be changing. The test object and each constituent test item are often changing. The test environment might be changing. The test basis might be changing. Logging should identify the versions tested.

The military strategist Clausewitz referred famously to the "fog of war." What he meant was not a literal fog—though black-powder cannons and firearms of his day created plenty of that!—but rather a metaphorical fog whereby no one observing a battle, be they an infantryman or a general, could truly grasp the whole picture.

Clausewitz would recognize his famous fog if he were to come back to life and work as a tester. Test execution periods tend to have a lot of fog. Good test logs are the fog-cutter. Test logs should provide a detailed, rich chronology of test execution.

To do so, test logs need to be test by test and event by event. Each test, uniquely identified, should have status information logged against it as it goes through the test execution period. This information should support not only understanding the overall test status but also the overall test coverage.

ISTQB Glossary

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

You should also log events that occur during test execution and affect the test execution process, whether directly or indirectly. You should document anything that delays, interrupts, or blocks testing.

Test analysts are not always also test managers, but they should work closely with the test managers. Test managers need logging information for test control, test progress reporting, and test process improvement. Test analysts need logging information too, along with the test managers, for measurement of exit criteria, which we'll cover later in this chapter.

Finally, note that the extent, type, and details of test logs will vary based on the test level, the test strategy, the test tools, and various standards and regulations. Automated component testing results in the automated test gathering logging information. Manual acceptance testing usually involves the test manager compiling the test logs or at least collating the information coming from the testers. If we're testing regulated, safety-critical systems like pharmaceutical systems, we might have to log certain information for audit purposes.

2.5.7 Use of Amateur Testers

Amateur testers. This term is rather provocative, so here's what we mean. A person who primarily works as a tester to earn a living is a professional tester. Anyone else engaged in testing is an amateur tester.

Rex is a professional tester now—and has been since 1987. Before that, he was a professional programmer. Rex still writes programs from time to time, but now he's an amateur programmer. He makes many typical amateur-programmer mistakes when he does it. Before Rex was a professional tester, he unit-tested his code as a programmer. He made many typical amateur-tester mistakes when he did that. Since one of the companies he worked for as a programmer relied entirely on programmer unit testing, that sometimes resulted in embarrassing outcomes for their customers—and for Rex.

Jamie is also a professional tester, and he has been one since 1991. Jamie also does some programming from time to time, but he considers himself an amateur programmer. He makes many typical amateur-programmer mistakes when he does it, mistakes that full-time professional programmers often do not make. There are two completely differing mind-sets for testing and programming. Companies that rely entirely on programmer testing sometimes find that it results in embarrassing outcomes for their customers. Professional testers could have solved some of those problems.

There's nothing wrong with involving amateur testers. Sometimes, we want to use amateur testers such as users or customers during test execution. It's important to understand what we're trying to accomplish with this and why it will (or won't) work. For example, often the objective is to build user confidence in the system, but that can backfire! Suppose we involve them too early, when the system is still full of bugs. Oops!

2.5.8 Standards

Let's look at some standards that relate to implementation and execution as well as to other parts of the test process.

Let's start with the IEEE 829 standard. Most of this material about IEEE 829 should be a review of the Foundation syllabus for you, but it might be a while since you've looked at it.

The first IEEE 829 template, which we'd use during test implementation, is the IEEE 829 test procedure specification. A test procedure specification describes how to run one or more test cases. This template includes the following sections:

- Test procedure specification identifier
- Purpose (e.g., which tests are run)
- Special requirements (skills, permissions, environment, etc.)
- Procedure steps (log, set up, start, proceed [the steps themselves], measure results, shut down/suspend, restart [if needed], stop, wrap up/tear down, contingencies)

While the IEEE 829 standard distinguishes between test procedures and test cases, in practice test procedures are often embedded in test cases.

A test procedure is sometimes referred to as a test script. A test script can be manual or automated.

The IEEE 829 standard for test documentation also includes ideas on what to include in a test log. According to the standard, a test log should record the relevant details about test execution. This template includes the following sections:

- Test log identifier.
- Description of the testing, including the items under test (with version numbers), the test environments being used, and the like.
- Activity and event entries. These should be test by test *and* event by event. Events include things like test environments becoming unavailable, people being out sick, and so forth. You should capture information on the test execution process; the results of the tests; environmental changes or issues; bugs, incidents, or anomalies observed; the testers involved; any suspension or blockage of testing; changes to the plan and the impact of change; and so forth.

For those who find the IEEE 829 templates too daunting, simple spreadsheets could also be used to capture execution logging information.

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
- Modified condition decision testing
- Linear Code Sequence and Jump (LCSAJ) testing

Rounding out the list are two sections, “Random Testing” and “Other Testing Techniques.” Random testing was not covered in the Foundation syllabus, but we’ll talk about the use of randomness in relation to reliability testing in chapters 5 and 9. 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 (ATTA) exam?” The answer is in two parts. First, you need to know any test design technique that was on the Foundation syllabus. Such techniques may be covered on the Advanced Level Technical Test Analyst exam. Second, we’ll cover the new test design techniques that might be on the Advanced Level Test Analyst (ATA) exam in detail in chapter 4.

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-178B. (We’ll cover that standard in a moment.) So during test design, we’d employ the test design techniques. During test implementation, we’d use the test measurement techniques to ensure adequate coverage.

In addition to these two major sections, this document also includes two annexes. Annex B brings the dry material in the first two major sections to life by showing an example of applying them to realistic situations. Annex A covers

process considerations, which is perhaps closest to our area of interest here. It discusses the application of the standard to a test project, following a test process given in the document. 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, though, that the ISTQB process includes that as part of a larger activity, test implementation and execution. Note also that the ISTQB process includes test logging as part of test execution, while BS 7925/2 has a separate test recording process.
- Finally, BS 7925/2 has checking for test completion as the final step in its process. That corresponds roughly to the ISTQB's evaluating test criteria and reporting.

Finally, as promised, let's talk about the DO-178B standard. This standard is promulgated by the United States Federal Aviation Administration. As you might guess, it's for avionics 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 2-1.

Table 2–1 *FAA DO/178B mandated coverage*

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

Let us explain table 2-1 a bit more thoroughly:

> Criticality level A, or Catastrophic, applies when a software failure can result in a catastrophic failure of the system. For software with such criticality, the standard requires Modified Condition/Decision, Decision, and Statement coverage.
>
> Criticality level B, or Hazardous and Severe, applies when a software failure can result in a hazardous, severe, or major failure of the system. For software with such criticality, the standard requires Decision and Statement coverage.
>
> Criticality level C, or Major, applies when a software failure can result in a major failure of the system. For software with such criticality, the standard requires only Statement coverage.
>
> Criticality level D, or Minor, applies when a software failure can only result in a minor failure of the system. For software with such criticality, the standard does not require any level of coverage.
>
> Finally, criticality level E, or No Effect, applies when a software failure cannot have an effect on the system. For software with such criticality, the standard does not require any level of coverage.

This makes a certain amount of sense. You should be more concerned about software that affects flight safety, such as rudder and aileron control modules, than 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.

By the way, if you found this a bit confusing, note that two of the white-box coverage metrics we mentioned, statement and decision coverage, were discussed in the Foundation syllabus, in chapter 4. Modified condition/decision coverage was mentioned briefly but not described in any detail in the Foundation; Modified condition/decision coverage will be covered in detail in this book. If you don't remember what statement and decision coverage mean, you should go back and review the material in that chapter on white-box coverage metrics. We'll return to black-box and white-box testing and test coverage in chapter 4 of this book, and we'll assume that you understand the

relevant Foundation-level material. Chapter 4 will be hard to follow if you are not fully conversant with the Foundation-level test design techniques.

2.5.9 Metrics

Finally, what metrics and measurements can we use for the test implementation and execution of the ISTQB fundamental test process? Different people use different metrics, of course.

Typical metrics during test implementation are the percentage of test environments configured, the percentage of test data records loaded, and the percentage of test cases automated.

During test execution, typical metrics look at the percentage of test conditions covered, test cases executed, and so forth.

We can track test implementation and execution tasks against a work-breakdown-structure, which is useful in determining whether we are proceeding according to the estimate and schedule.

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

2.6 Evaluating Exit Criteria and Reporting

Learning objectives

(K3) Determine from a given set of measures if a test completion criterion has been fulfilled.

In one sense, the evaluation of exit criteria and reporting of results is a test management activity. And, in the volume on advanced test management, we examine ways to analyze, graph, present, and report test results as part of the test progress monitoring and control activities.

However, in another sense, there is a key piece of this process that belongs to the technical test analyst. As technical test analysts, in this book we are more interested in two main areas. First, we need to collect the information necessary to support test management reporting of results. Second, we need to measure progress toward completion, and by extension, we need to

detect deviation from the plan. (Of course, if we do detect deviation, it is the test manager's role, as part of the control activities, to get us back on track.)

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

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

We can track evaluation and reporting milestones and tasks against a work-breakdown-structure, which is useful in determining whether we are proceeding according to the estimate and schedule.

We'll now look at examples of test metrics and measures that you can use to evaluate where you stand with respect to exit criteria. Most of these are drawn from actual case studies, projects where RBCS helped a client with their testing.

2.6.1 Test Suite Summary

Figure 2-3 shows a test suite summary worksheet. Such a worksheet summarizes the test-by-test logging information described in a previous section. As a test analyst, you can use this worksheet to track a number of important properties of the test execution so far:

- Test case progress
- Test case pass/fail rates
- Test suite defect priority/severity (weighted failure)
- Earned value

As such, it's a useful chart for the test analyst to understand the status of the entire test effort.

Test Suite Summary
Test Pass Two

	Total	Planned Tests Fullfilled				Weighted	Planned Tests Unfullfilled				Earned Value			
Suite	Cases	Count	Skip	Pass	Fail	Failure	Count	Queued	IP	Block	Plan Hrs	Actual Hrs	%Effort	%Exec
Functionality	14	14	0	4	10	10.6	0	0	0	0	36.0	49.0	163%	100%
Performance	5	4	0	1	3	8.7	1	1	0	0	7.0	11.5	164%	80%
Reliability	2	2	2	0	0	0.0	0	0	0	0	0.0	0.0	0%	0%
Robustness	3	1	0	0	1	0.5	2	2	0	0	12.5	8.0	64%	33%
Installation	4	0	0	0	0	0.0	4	4	0	0	72.0	0.0	0%	0%
Localization	8	1	0	0	1	0.5	7	0	7	0	128.0	22.0	17%	13%
Security	4	2	0	2	0	1.1	2	2	0	0	17.0	8.5	50%	50%
Documentation	3	1	0	0	1	4.0	2	2	0	0	28.0	15.0	54%	33%
Integration	4	4	0	3	1	1.0	0	0	0	0	8.0	12.5	156%	100%
Usability	2	0	0	0	0	0.0	2	0	2	0	16.0	0.0	0%	0%
Exploratory	6	0	0	0		0.0	6	6	0	0	12.0	0.0	0%	0%
Total	55	29	2	10	17	26.4	26	17	9	0	336.5	126.5	38%	51%
Percent	100%	53%	4%	18%	31%	N/A	47%	31%	16%	0%				

Figure 2–3 *Test suite summary worksheet*

Down the left side of figure 2-3, you see two columns, the test suite name and the number of test cases it contains. Again, in some test log somewhere, we have detailed information for each test case.

On the middle-left side, you see four columns under the general heading of "Planned Tests Fulfilled." These are the tests for which no more work remains, at least during this pass of testing.

The weighted failure numbers for each test suite, found in the column about in the middle of the table, give a metric of historical bug finding effectiveness. Each bug is weighted by priority and severity—only the severity one, priority one bugs, count for a full weighted failure point, while lower severity and priority can reduce the weighted failure point count for a bug to as little 0.04. So this is a metric of the technical risk, the likelihood of finding problems, associated with each test suite based on historical bug finding effectiveness.

On the middle-right side, you see four columns under the general heading of "Planned Tests Unfulfilled." These are the tests for which more work remains during this pass of testing. *IP*, by the way, stands for "in progress."

Finally, on the right side you see four columns under the general heading of "Earned Value." Earned value is a simple project management concept. It

says that, in a project, we accomplish tasks by expending resources. So if the percentage of tasks accomplished is about equal to the percentage of resources expended, we're on track. If the percentage of tasks accomplished is greater than the percentage of resources expended, we're on our way to being under budget. If the percentage of tasks accomplished is less than the percentage of resources expended, we're on our way to being over budget.

Similarly, from a schedule point of view, the percentage of tasks accomplished should be approximately equal to the percentage of project time elapsed. As with effort, if the tasks percentage is over the schedule percentage, we're happy. If the tasks percentage is below the schedule percentage, we're sad and worried.

In test execution, we can consider the test case or test procedure to be our basic task. The resources—for manual testing, anyway—are usually the person-hours required to run the tests. That's the way this chart shows earned value, test suite by test suite.

Take a moment to study figure 2-3. Think about how you might be able to use it on your current (or future) projects.

2.6.2 Defect Breakdown

We can analyze defect (or bug) reports in a number of ways. There's just about no end to the analysis that we have done as technical test analysts, test managers, and test consultants. As a test professional, you should think of good, proper analysis of bug reports much the same way as a doctor thinks of good, proper analysis of blood samples.

Just one of these many ways to examine bug reports is by looking at the breakdown of the defects by severity, priority, or some combination of the two. If you use a numerical scale for severity and priority, it's easy to multiply them together to get a weighted metric of overall bug importance, as you just saw.

Figure 2-4 shows an example of such a chart. What can we do with it? Well, for one thing, we can compare this chart with previous projects to get a sense of whether we're in better or worse shape. Remember, though, that the distribution will change over the life of the project. Ideally, the chart will start skewed toward high-priority bugs—at least it will if we're doing proper risk-based testing, which we'll discuss in chapter 3.

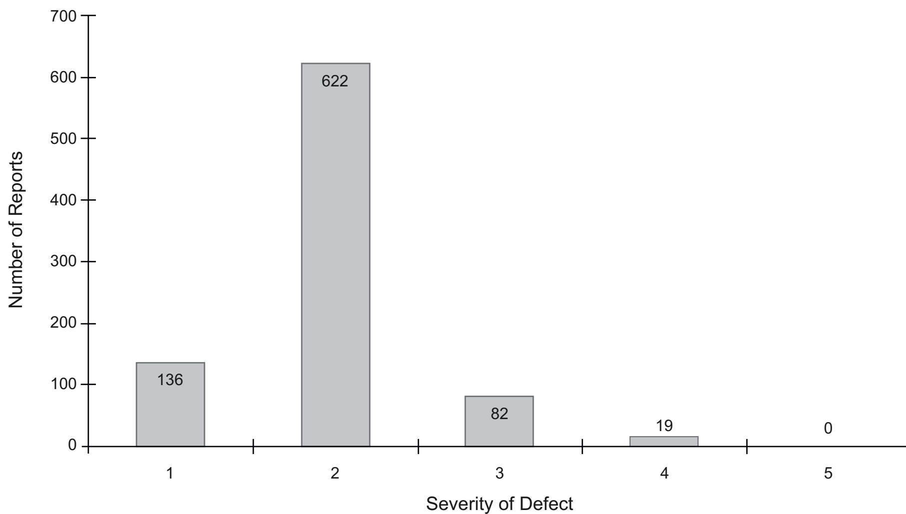


Figure 2–4 *Breakdown of defects by severity*

Figure 2-4 shows lots of severity ones and twos. Usually, severity one is loss of data. Severity two is loss of functionality without a workaround. Either way, bad news.

This chart tells us, as test professionals, to take a random sample of 10 or 20 of these severity one and two reports and see if we have severity inflation going on here. Are our severity classifications accurate? If not, the poor test manager's reporting will be biased and alarmist, which will get her in trouble.

As we mentioned previously, though, if we're doing risk-based testing, this is probably about how this chart should look during the first quarter or so of the project. Find the scary stuff first, we always tell testers. If these severity classifications are right, the test team is doing just that.

2.6.3 Confirmation Test Failure Rate

As technical test analysts, we can count on finding some bugs. Hopefully many of those bugs will be sent back to us, allegedly fixed. At that point, we confirmation-test the fix. How many of those fixes fail the confirmation test? It can feel like it's quite a few, but is it really? Figure 2-5 shows the answer, graphically, for an actual project.

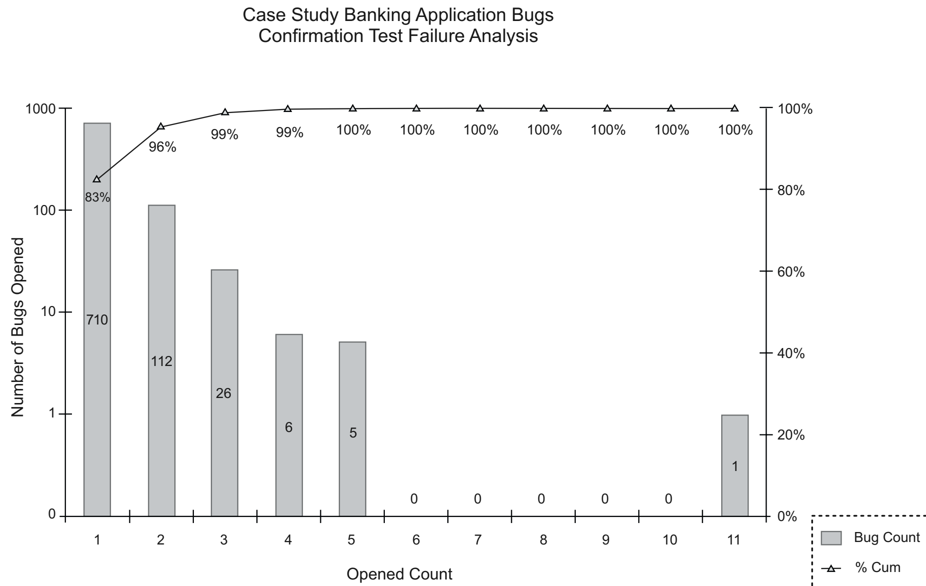


Figure 2–5 *Confirmation test failure analysis*

On this banking project, we can see that quite a few bug fixes failed the confirmation test. We had to reopen fully one in six defect reports at least once. That's a lot of wasted time. It's also a lot of potential schedule delay. Study figure 2-5, thinking about ways you could use this information during test execution.

2.6.4 System Test Exit Review

Finally, let's look at another case study. Figure 2-6 shows an excerpt of the exit criteria for an Internet appliance project that RBCS provided testing for. You'll see that we have graded the criteria as part of a system test exit review.

Each of the three criteria here is graded on a three-point scale:

- Green: Totally fulfilled, with little remaining risk
- Yellow: Not totally fulfilled, but perhaps an acceptable risk
- Red: Not in any sense fulfilled, and poses a substantial risk

Of course, you'd want to provide additional information and data for the yellows and the reds.

System Test Exit Review

Per the Test Plan, System Test was planned to end when following criteria were met:

1. All design, implementation, and feature completion, code completion, and unit test completion commitments made in the System Test Entry meeting were either met or slipped to no later than four (4), three (3), and three (3) weeks, respectively, prior to the proposed System Test Exit date.
   STATUS: RED. Audio and demo functionality have entered System Test in the last three weeks. The modems entered System Test in the last three weeks. On the margins of a violation, off-hook detection was changed significantly.

2. No panic, crash, halt, wedge, unexpected process termination, or other stoppage of processing has occurred on any server software or hardware for the previous three (3) weeks.
   STATUS: YELLOW. The servers have not crashed, but we did not complete all the tip-over and fail-over testing we planned, and so we are not satisfied that the servers are stable under peak load or other inclement conditions.

3. Production Devices have been used for all System Test execution for at least three (3) weeks.
   STATUS: GREEN. Except for the modem situation discussed above, the hardware has been stable.

Figure 2–6 *Case study of system test exit review*

2.6.5 Standards

Finally, let's look at one more IEEE 829 template, one that applies to this part of the test process. The IEEE 829 standard for test documentation includes a template for test summary reports.

A test summary report describes the results of a given level or phase of testing. The IEEE 829 template includes the following sections:

- Test summary report identifier
- Summary (e.g., what was tested, what the conclusions are, etc.)
- Variances (from plan, cases, procedures)
- Comprehensive assessment
- Summary of results (e.g., final metrics, counts)
- Evaluation (of each test item vis-à-vis pass/fail criteria)
- Summary of activities (resource use, efficiency, etc.)
- Approvals

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

ISTQB Glossary

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

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

2.6.6 Evaluating Exit Criteria and Reporting Exercise

Consider the complete set of actual exit criteria from the Internet appliance project, which is shown in the following subsection. Toward the end of the project, the test team rated each criterion on the following scale:

Green: Totally fulfilled, with little remaining risk
Yellow: Not totally fulfilled, but perhaps an acceptable risk
Red: Not in any sense fulfilled, and poses a substantial risk

You can see the ratings we gave each criterion in the STATUS block below the criterion itself.

We used this evaluation of the criteria as an agenda for a System Test Exit Review meeting. Rex led the meeting and walked the team through each criterion. As you can imagine, the RED ones required more explanation than the YELLOW and GREEN ones.

While narrative explanation is provided for each evaluation, perhaps more information and data are needed. So, for each criterion, determine what kind of data and other information you'd want to collect to support the conclusions shown in the status evaluations for each.

2.6.7 System Test Exit Review

Per the Test Plan, System Test was planned to end when the following criteria were met:

1. All design, implementation, and feature completion, code completion, and unit test completion commitments made in the System Test Entry meeting were either met or slipped to no later than four (4), three (3), and three (3) weeks, respectively, prior to the proposed System Test Exit date.
   STATUS: RED. Audio and demo functionality have entered System Test in the last three weeks. The modems entered System Test in the last three weeks. On the margins of a violation, off-hook detection was changed significantly.
2. No panic, crash, halt, wedge, unexpected process termination, or other stoppage of processing has occurred on any server software or hardware for the previous three (3) weeks.
   STATUS: YELLOW. The servers have not crashed, but we did not complete all the tip-over and fail-over testing we planned, and so we are not satisfied that the servers are stable under peak load or other inclement conditions.
3. Production devices have been used for all System Test execution for at least three (3) weeks.
   STATUS: GREEN. Except for the modem situation discussed above, the hardware has been stable.
4. No client systems have become inoperable due to a failed update for at least three (3) weeks.
   STATUS: YELLOW. No system has become permanently inoperable during update, but we have seen systems crash during update and these systems required a reboot to clear the error.
5. Server processes have been running without installation of bug fixes, manual intervention, or tuning of configuration files for two (2) weeks.
   STATUS: RED. Server configurations have been altered by Change Committee–approved changes multiple times over the last two weeks.
6. The Test Team has executed all the planned tests against the release-candidate hardware and software releases of the Device, Server, and Client.
   STATUS: RED. We had planned to test Procurement and Fulfillment, but disengaged from this effort because the systems were not ready. Also, we have just received the release-candidate build; complete testing would take two weeks. In addition, the servers are undergoing Change Committee–approved changes every few days and a new load balancer has been added to the server farm. These server changes have prevented volume, tip-over,

and fail-over testing for the last week and a half. Finally, we have never had a chance to test the server installation and boot processes because we never received documentation on how to perform these tasks.

7. The Test Team has retested all fixes for priority one and two bug reports over the life of the project against the release-candidate hardware and software releases of the Device, Server, and Client.
   STATUS: RED. Testing of the release-candidate software and hardware has been schedule-limited to one week, which does not allow for retesting of all bug fixes.
8. The Development Teams have resolved all "must-fix" bugs. "Must-fix" will be defined by the Project Management Team.
   STATUS: RED. Referring to the attached open/closed charts and the "Bugs Found Since 11/9" report, we continue to find new bugs in the product, though there is good news in that the find rate for priority one bugs has leveled off. Per the closure period charts, it takes on average about two weeks—three weeks for priority one bugs—to close a problem report. In addition, both open/close charts show a significant quality gap between cumulative open and cumulative closed, and it's hard to believe that taken all together, a quantity of bugs that significant doesn't indicate a pervasive fit-and-finish issue with the product. Finally, note that WebGuide and E-commerce problems are design issues—the selected browser is basically incompatible with much of the Internet—which makes these problems much more worrisome.
9. The Test Team has checked that all issues in the bug tracking system are either closed or deferred and, where appropriate, verified by regression and confirmation testing.
   STATUS: RED. A large quality gap exists and has existed for months. Because of the limited test time against the release-candidate build, the risk of regression is significant.
10. The open/close curve indicates that we have achieved product stability and reliability.
    STATUS: RED. The priority-one curve has stabilized, but not the overall bug-find curve. In addition, the run chart of errors requiring a reboot shows that we are still showing about one crash per eight hours of system operation, which is no more stable than a typical Windows 95/Windows 98 laptop. (One of the ad claims is improved stability over a PC.)

11. The Project Management Team agrees that the product, as defined during the final cycle of System Test, will satisfy the customer's reasonable expectations of quality.
    STATUS: YELLOW. We have not really run enough of the test suite at this time to give a good assessment of overall product quality.
12. The Project Management Team holds a System Test Exit Meeting and agrees that we have completed System Test.
    STATUS: In progress.

2.6.8 Evaluating Exit Criteria and Reporting Exercise Debrief

We have added a section called "ADDITIONAL DATA AND INFORMATION" below each criterion. In that section, you'll find Rex's solution to this exercise, based both on what kind of additional data and information he actually had during this meeting and what he would have brought if he knew then what he knows now.

1. All design, implementation, and feature completion, code completion, and unit test completion commitments made in the System Test Entry meeting were either met or slipped to no later than four (4), three (3), and three (3) weeks, respectively, prior to the proposed System Test Exit date.
   STATUS: RED. Audio and demo functionality have entered System Test in the last three weeks. The modems entered System Test in the last three weeks. On the margins of a violation, off-hook detection was changed significantly.
   ADDITIONAL DATA AND INFORMATION: The specific commitments made in the System Test Entry meeting. The delivery dates for the audio functionality, the demo functionality, the modem, and the off-hook detection functionality.
2. No panic, crash, halt, wedge, unexpected process termination, or other stoppage of processing has occurred on any server software or hardware for the previous three (3) weeks.
   STATUS: YELLOW. The servers have not crashed, but we did not complete all the tip-over and fail-over testing we planned, and so we are not satisfied that the servers are stable under peak load or other inclement conditions.
   ADDITIONAL DATA AND INFORMATION: Metrics indicate the percentage completion of tip-over and fail-over tests. Details on which specific

quality risks remain uncovered due to the tip-over and fail-over tests not yet run.

3. Production devices have been used for all System Test execution for at least three (3) weeks.
   STATUS: GREEN. Except for the modem situation discussed above, the hardware has been stable.
   ADDITIONAL DATA AND INFORMATION: None. Good news requires no explanation.
4. No client systems have become inoperable due to a failed update for at least three (3) weeks.
   STATUS: YELLOW. No system has become permanently inoperable during update, but we have seen systems crash during update and these systems required a reboot to clear the error.
   ADDITIONAL DATA AND INFORMATION: Details, from bug reports, on the system crashes described.
5. Server processes have been running without installation of bug fixes, manual intervention, or tuning of configuration files for two (2) weeks.
   STATUS: RED. Server configurations have been altered by Change Committee–approved changes multiple times over the last two weeks.
   ADDITIONAL DATA AND INFORMATION: List of tests that have been run prior to the last change, along with an assessment of the risk posed to each test by the change. (Note: Generating this list and the assessment could be a lot of work unless you have good traceability information.)
6. The Test Team has executed all the planned tests against the release-candidate hardware and software releases of the Device, Server, and Client.
   STATUS: RED. We had planned to test Procurement and Fulfillment, but disengaged from this effort because the systems were not ready. Also, we have just received the release-candidate build; complete testing would take two weeks. In addition, the servers are undergoing Change Committee–approved changes every few days and a new load balancer has been added to the server farm. These server changes have prevented volume, tip-over, and fail-over testing for the last week and a half. Finally, we have never had a chance to test the server installation and boot processes because we never received documentation on how to perform these tasks.
   ADDITIONAL DATA AND INFORMATION: List of Procurement and

Fulfillment tests skipped, along with the risks associated with those tests. List of tests that will be skipped due to time compression of the last pass of testing against the release-candidate, along with the risks associated with those tests. List of changes to the server since the last volume, tip-over, and fail-over tests along with an assessment of reliability risks posed by the change. (Again, this could be a big job.) List of server install and boot process tests skipped, along with the risks associated with those tests.

7. The Test Team has retested all priority one and two bug reports over the life of the project against the release-candidate hardware and software releases of the Device, Server, and Client.
   STATUS: RED. Testing of the release-candidate software and hardware has been schedule-limited to one week, which does not allow for retesting of all bugs.
   ADDITIONAL DATA AND INFORMATION: The list of all the priority one and two bug reports filed during the project, along with an assessment of the risk that those bugs might have re-entered the system in a change-related regression not otherwise caught by testing. (Again, potentially a huge job.)
8. The Development Teams have resolved all “must-fix” bugs. “Must-fix” will be defined by the Project Management Team.
   STATUS: RED. Referring to the attached open/closed charts and the “Bugs Found Since 11/ 9” report, we continue to find new bugs in the product, though there is good news in that the find rate for priority one bugs has leveled off. Per the closure period charts, it takes on average about two weeks—three weeks for priority one bugs—to close a problem report. In addition, both open/close charts show a significant quality gap between cumulative open and cumulative closed, and it’s hard to believe that taken all together, a quantity of bugs that significant doesn’t indicate a pervasive fit-and-finish issue with the product. Finally, note that Web and E-commerce problems are design issues—the selected browser is basically incompatible with much of the Internet—which makes these problems much more worrisome.
   ADDITIONAL DATA AND INFORMATION: Open/closed charts, list of bugs since November 9, closure period charts, and a list of selected impor-

tant sites that won't work with the browser. (Note: The two charts mentioned are covered in the Advanced Test Manager course, if you're curious.)

9. The Test Team has checked that all issues in the bug tracking system are either closed or deferred and, where appropriate, verified by regression and confirmation testing.
   STATUS: RED. A large quality gap exists and has existed for months. Because of the limited test time against the release-candidate build, the risk of regression is significant.
   ADDITIONAL DATA AND INFORMATION: List of bug reports that are neither closed nor deferred, sorted by priority. Risk of tests that will not be run against the release-candidate software, along with the associated risks for each test.
10. The open/close curve indicates that we have achieved product stability and reliability.
    STATUS: RED. The priority-one curve has stabilized, but not the overall bug-find curve. In addition, the run chart of errors requiring a reboot shows that we are still showing about one crash per eight hours of system operation, which is no more stable than a typical Windows 95/Windows 98 laptop. (One of the ad claims is improved stability over a PC.)
    ADDITIONAL DATA AND INFORMATION: Open/closed chart (run for priority one defects only and again for all defects). Run chart of errors requiring a reboot; i.e., a trend chart that shows how many reboot-requiring crashes occurred each day.
11. The Project Management Team agrees that the product, as defined during the final cycle of System Test, will satisfy the customer's reasonable expectations of quality.
    STATUS: YELLOW. We have not really run enough of the test suite at this time to give a good assessment of overall product quality.
    ADDITIONAL DATA AND INFORMATION: List of all the tests not yet run against the release-candidate build, along with their associated risks.
12. The Project Management Team holds a System Test Exit Meeting and agrees that we have completed System Test.
    STATUS: In progress.
    ADDITIONAL DATA AND INFORMATION: None.

2.7 Test Closure Activities

Learning objectives

Recall of content only

The concepts in this section apply primarily for test managers. There are no learning objectives defined for technical test analysts in this section. In the course of studying for the exam, read this section in chapter 2 of the Advanced syllabus for general recall and familiarity only.

2.8 Sample Exam Questions

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

1 Identify all of the following that can be useful as a test oracle the first time a test case is run.

  A Incident report

  B Requirements specification

  C Test summary report

  D Legacy system

2 Assume you are a technical test an=2alyst working on a banking project to upgrade an existing automated teller machine system to allow customers to obtain cash advances from supported credit cards. During test design, you identify a discrepancy between the list of supported credit cards in the requirements specification and the design specification. This is an example of what?

  A Test design as a static test technique

  B A defect in the requirements specification

  C A defect in the design specification

  D Starting test design too early in the project

3 Which of the following is *not always* a precondition for test execution?

A A properly configured test environment

B A thoroughly specified test procedure

C A process for managing identified defects

D A test oracle

4 Assume you are a technical test analyst working on a banking project to upgrade an existing automated teller machine system to allow customers to obtain cash advances from supported credit cards. One of the exit criteria in the test plan requires documentation of successful cash advances of at least 500 euros for all supported credit cards. The correct list of supported credit cards is American Express, Visa, Japan Credit Bank, Eurocard, and MasterCard.

After test execution, a complete list of cash advance test results shows the following:

- American Express allowed advances of up to 1,000 euros.
- Visa allowed advances of up to 500 euros.
- Eurocard allowed advances of up to 1,000 euros.
- MasterCard allowed advances of up to 500 euros.

Which of the following statements is true?

A. The exit criterion fails due to excessive advances for American Express and Eurocard.

B. The exit criterion fails due to a discrepancy between American Express and Eurocard on the one hand and Visa and MasterCard on the other hand.

C. The exit criterion passes because all supported cards allow cash advances of at least the minimum required amount.

D. The exit criterion fails because we cannot document Japan Credit Bank results.

3 Test Management

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

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

Let's look at the sections that pertain to technical test analysis.

ISTQB Glossary

test management: The planning, estimating, monitoring and control of test activities, typically carried out by a test manager.

3.1 Introduction

Learning objectives

Recall of content only

This chapter, as the name indicates, is focused primarily on test management topics. Thus, it is mainly the purview of Advanced Test Manager exam candidates. Since this book is for technical test analysts, most of our coverage in this chapter is to support simple recall.

However, there is one key area that, as a technical test analyst, you need to understand very well: risk-based testing. 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.

This chapter in the Advanced syllabus also covers test documentation templates for test managers. It focuses on the IEEE 829 standard. As a technical test analyst, you'll need to know that standard as well. If you plan to take the Advanced Level Technical Test Analyst exam, remember that all material from the Foundation syllabus, including that related to the use of the IEEE 829 templates and the test management material in chapter 5 of the Foundation syllabus, is examinable.

3.2 Test Management Documentation

Learning objectives

Recall of content only

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

ISTQB Glossary

test policy: A high-level document describing the principles, approach, and major objectives of the organization regarding testing.

test strategy: A high-level description of the test levels to be performed and the testing within those levels for an organization or program (one or more projects).

test level: A group of test activities that are organized and managed together. A test level is linked to the responsibilities in a project. Examples of test levels are component test, integration test, system test, and acceptance test.

level test plan: A test plan that typically addresses one test level. See also *test plan*.

master test plan: A test plan that typically addresses multiple test levels. See also *test plan*.

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

However, the Foundation syllabus covered test management, including the topic of test management documentation. Anything covered in the Foundation syllabus is examinable. In addition, all sections of the Advanced syllabus are examinable in terms of general recall.

So, if you are studying for the exam, you'll want to read this section in chapter 3 of the Advanced syllabus for general recall and familiarity and review the test management documentation material from the Foundation syllabus.

3.3 Test Plan Documentation Templates

Learning objectives

Recall of content only

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

ISTQB Glossary

test estimation: The calculated approximation of a result related to various aspects of testing (e.g., effort spent, completion date, costs involved, number of test cases, etc.) which is usable even if input data may be incomplete, uncertain, or noisy.

test schedule: A list of activities, tasks, or events of the test process, identifying their intended start and finish dates and/or times and interdependencies.

test point analysis (TPA): A formula-based test estimation method based on function point analysis.

Wideband Delphi: An expert-based test estimation technique that aims at making an accurate estimation using the collective wisdom of the team members. [Note: The glossary spells this as Wide Band Delphi, though the spelling given here is more commonly used.]

Earlier, in chapter 2, we reviewed portions of the IEEE 829 test documentation standard. Specifically, we looked at the test design specification template, the test case specification template, the test procedure specification template, the test summary report template, and the test log template. However, the Foundation covered the entire IEEE 829 standard, including the test plan template, test item transmittal report template, and the incident report template.

Anything covered in the Foundation syllabus is examinable. All sections of the Advanced syllabus are examinable in terms of general recall. If you are studying for the exam, you'll want to read this section in chapter 3 of the Advanced syllabus for general recall and familiarity and review the test management documentation material from the Foundation syllabus.

3.4 Test Estimation

Learning objectives

Recall of content only

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

However, chapter 5 of the Foundation syllabus covered test estimation as part of the material on test management. Anything covered in the Foundation syllabus is examinable. All sections of the Advanced syllabus are examinable in terms of general recall. If you are studying for the exam, you'll want to read this section in chapter 3 of the Advanced syllabus for general recall and familiarity and review the test estimation material from the Foundation syllabus.

3.5 Scheduling and Test Planning

Learning objectives

Recall of content only

The concepts in this section apply primarily for test managers. There are no learning objectives defined for technical test analysts in this section. In the course of studying for the exam, read this section in chapter 3 of the Advanced syllabus for general recall and familiarity only.

3.6 Test Progress Monitoring and Control

Learning objectives

Recall of content only

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

However, chapter 5 of the Foundation syllabus covers test progress monitoring and control as part of the material on test management. Anything covered in the Foundation syllabus is examinable. All sections of the Advanced syllabus are examinable in terms of general recall. If you are studying for the exam, you'll want to read this section in chapter 3 of the Advanced syllabus for general recall and familiarity and review the test progress monitoring and control material from the Foundation syllabus.

> **ISTQB Glossary**
>
> **test monitoring:** A test management task that deals with the activities related to periodically checking the status of a test project. Reports are prepared that compare the actuals to that which was planned. See also *test management*.

3.7 Business Value of Testing

Learning objectives

Recall of content only

The concepts in this section apply primarily for test managers. There are no learning objectives defined for technical test analysts in this section. In the course of studying for the exam, read this section in chapter 3 of the Advanced syllabus for general recall and familiarity only.

3.8 Distributed, Outsourced, and Insourced Testing

Learning objectives

Recall of content only

The concepts in this section apply primarily for test managers. There are no learning objectives defined for technical test analysts in this section. In the course of studying for the exam, read this section in chapter 3 of the Advanced syllabus for general recall and familiarity only.

3.9 Risk-Based Testing

Learning objectives

(K2) Outline the activities of a risk-based approach for planning and executing technical testing.

[Note: While the Advanced syllabus does not include K3 or K4 learning objectives for technical test analysts (unlike for test analysts) in this section, we feel that technical test analysts have an equal need for the ability to perform a risk analysis. Therefore, we will include exercises on this topic from a technical perspective.]

Risk is the possibility of a negative or undesirable outcome or event. A specific 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 product or quality risk. When the primary impact of a potential problem is on product quality, such potential problems are called product risks. A synonym for *product risks*, which we use most frequently ourselves, is *quality risks*. An example of a quality risk is a possible reliability defect that could cause a system to crash during normal operation.

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

Of course, you can consider a quality risk as a special type of project risk. While the ISTQB definintion of project risk is given here, Jamie likes 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.

ISTQB Glossary

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

risk type: A set of risks grouped by one or more common factors such as a quality attribute, cause, location, or potential effect of risk. A specific set of product risk types is related to the type of testing that can mitigate (control) that risk type. For example, the risk of user interactions being misunderstood can be mitigated by usability testing.

risk category: see *risk type*. [Note: Included because this is also a common term, and because the process of risk analysis as described in the Advanced syllabus includes risk categorization.]

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.]

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*.

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.

Not all risks are equal in importance. 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

Note the distinction made here in terms of project timeline. Likelihood is assessed based on the likelihood of a problem existing in the software, not on the likelihood of it being encountered by the user. The likelihood of a user encountering the problem influences impact.

Likelihood of a problem arises primarily from technical considerations, such as the programming languages used, the bandwidth of connections, and so forth. The impact of a problem arises from business considerations, such as the finan-

cial loss the business will suffer from a problem, the number of users or customers affected by a problem, and so forth.

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

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

- First, during all test activities, test 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.
- Second, during test planning and test control, test managers provide both mitigation and contingency responses for all significant, identified project risks. The higher the level of risk, the more thoroughly that project risk is managed.
- Third, 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. The three responses to risk we just covered—along with any others that might be needed—should occur throughout the lifecycle. Specifically, we should try to reduce quality risk by running tests and finding defects and reduce project risks through mitigation and, if necessary, contingency actions. Periodically in the project we should reevaluate risk and risk levels based on new information. This might result in our reprioritizing tests and defects, reallocating test effort, and taking other test control activities.

ISTQB Glossary

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

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.

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 that are worrisome, test for bugs that are worrisome, and ignore the areas and bugs that aren't worrisome.

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.

3.9.1 Risk Management

Risk management includes three primary activities:

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

ISTQB Glossary

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

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

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

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

In some sense, these activities are sequential, at least in terms of when they start. They are staged such that risk identification starts first. Risk analysis comes next. Risk control starts once risk analysis has determined the level of risk. However, since risk management should be continuous in a project, the reality is that risk identification, risk analysis, and risk control are all recurring activities.

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

In many cases, though, not all stakeholders can participate or are willing to do so. In such cases, some stakeholders may act as surrogates for other stakeholders. For example, in mass-market software development, the marketing team might ask a small sample of potential customers to help identify potential defects that would affect their use of the software most heavily. In this case, the sample of potential customers serves as a surrogate for the entire eventual customer base. As another example, business analysts on IT projects can sometimes represent the users rather than involving users in potentially distressing risk analysis sessions that include 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 as relates to technically based sources of risk and likelihood. 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.

With that overview of risk management in place, let's look at the three risk management activities more closely.

3.9.2 Risk Identification

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

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

Conceivably, you can use a single integrated process to identify both project and product risks. We usually separate them into two separate processes since they have two separate deliverables. We include the project risk identification process in the test planning process and thus hand the bulk of the responsibility for these kinds of risks to managers, including test managers. In parallel, the quality risk identification process occurs early in the project.

That said, project risks—and not just for testing but also for the project as a whole—are often identified as by-products of quality risk analysis. In addition, if you use a requirements specification, design specification, use cases, or other documentation 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.

Previously, 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 you will miss risk items and even whole risk categories.

ISTQB Glossary

Failure Mode and Effect Analysis (FMEA): A systematic approach to risk identification and analysis of possible modes of failure and attempting to prevent their occurrence. See also *Failure Mode, Effect and Criticality Analysis (FMECA)*.

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

How far should you take this process? Well, it depends on the technique. In the relatively informal Pragmatic Risk Analysis and Management technique that we frequently use, risk identification stops at the risk items. You have to be specific enough about the risk items to allow for analysis and assessment of each risk item 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 were to become an actual negative outcome. These effects include effects on the system—or the system of systems if applicable—as well as effects on the potential users, customers, stakeholders, and even society in general. Failure Mode and Effect Analysis is an example of such a formal risk management technique, and it is commonly used on safety-critical and embedded systems.

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 it ourselves, but Rex has talked to clients who have used it for safety-critical medical systems.[1]

1. For more information on risk identification and analysis techniques, you can see either of Rex Black's two books, *Managing the Testing Process, 3e* (for more both formal and informal techniques) or *Pragmatic Software Testing* (for informal techniques). 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, 3e* for a discussion of how the technique applies to software testing.

3.9.3 Risk Analysis or Risk Assessment

The next step in the risk management process is referred to in the Advanced syllabus as risk analysis. We prefer to call it risk assessment because analysis would seem to us to include both identification and assessment of risk. For example, the process of identifying risk items often includes analysis of work products such as requirements and metrics such as defects found in past projects. Regardless of what we call it, risk analysis or risk assessment involves the study of the identified risks. We typically want to categorize each risk item appropriately and assign each risk item an appropriate level of risk.

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

The other part of risk assessment is determining the level of risk. This often involves likelihood and impact as the two key factors. Likelihood arises from technical considerations, typically, while impact arises from business considerations.

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

- Complexity of technology and teams
- Personnel and training issues
- Intrateam and interteam conflict
- Supplier and vendor contractual problems
- Geographical distribution of the development organization, as with outsourcing
- Legacy or established designs and technologies versus new technologies and designs
- The quality—or lack of quality—in the tools and technology used
- Bad managerial or technical leadership
- Time, resource, and management pressure, especially when financial penalties apply

- Lack of earlier testing and quality assurance tasks in the lifecycle
- High rates of requirements, design, and code changes in the project
- High defect rates
- Complex interfacing and integration issues.

What business factors should we consider when assessing impact? Here's a list to get you started:

- 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

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 multiply the two values together, we can calculate the cost of exposure, which is called—in the insurance business—the expected payout or expected loss.

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

This is not to say—by any means—that a qualitative approach should be seen 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 of such techniques misleads the stakeholders about the extent to which you actually understand and can manage risk. What we've found is that if we accept the limits of our data and apply appropriate qualitative quality risk management approaches, the results are not only perfectly useful, but also indeed essential to a well-managed test process.

In any case, 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 different perceptions and thus possibly different opinions on the level of risk for each risk item. By including all these perceptions, we distill the collective wisdom of the team.

However, we do have a strong possibility of disagreements between stakeholders. 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.

3.9.4 Risk Mitigation or Risk Control

Having identified and assessed risks, we now must control them. As we mentioned earlier, the Advanced syllabus refers to this as risk mitigation, but that's not right. Risk control is a better term. We actually 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
- Contingency, where we have a plan or perhaps multiple plans to reduce the impact of a risk should it occur
- Transference, where we get another party to accept the consequences of a risk should it occur
- Finally, ignoring or accepting the risk and its consequences should it occur

For any given risk item, selecting one or more of these options creates its own set of benefits and opportunities as well as costs and, potentially, additional risks associated.

Analytical risk-based testing is focused on creating risk mitigation opportunities for the test team, including for technical test analysts, especially for qual-

ity risks. Risk-based testing mitigates quality risks via testing throughout the entire lifecycle.

Let us mention that, in some cases, there are standards that can apply. We've already looked at one such standard, the United States Federal Aviation Administration's DO-178B. We'll look at another one of those standards shortly.

It's important too that project risks be controlled. For 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 inputs 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 quality assurance (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, insisting on tougher entry criteria to testing, ensuring requirements for and designing for testability, participating in reviews (including retrospectives for earlier project activities), participating in problem and change management, and monitoring of the project progress and quality.

In preventive testing, we take quality risk control actions throughout the lifecycle. 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 (retesting) and regression testing

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

Of course, 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 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 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

---

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.

see if we have missed anything. We'll talk about about how to accomplish such testing, using blended testing strategies, in chapter 4.

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

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.

3.9.5 An Example of Risk Identification and Assessment Results

In figure 3-1, 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 Failure Mode and Effect Analysis (FMEA) approach.

Failure Mode and Effects Analysis (Quality Risks Analysis) Form--RPN Sorted

Inital FMEA--RPN Sorted

	A	B	C	D	E	F	G	H	I	J	K	L
9	System Function or Feature	Potential Failure Mode(s)-Quality Risk(s)	Potential Effect(s) of Failure	Critical?	Severity	Potential Cause(s) of Failure	Priority	Detection Method(s)	Detection	Risk Pri No	Recommended Action	Who/ When?
10	Shreds Swap Files	Fails to Shred	Security Breach	Y	1	Program Error	1	Test; Debug Trace; Code Review	1	1	Test; Debug Tracing; Code Review	
11	Shreds Swap Files	Shreds Excessively	Data Loss	Y	1	Program Error	1	Test; Debug Trace; Code Review	1	1	Test; Debug Tracing; Code Review	
12	Compression Compatibility	Damages Data	Data Loss	Y	1	Program Error	1	Test	2	2	Test	
13	Compression Compatibility	Hangs System	Data Loss	Y	1	Program Error	1	Test	2	2	Test	
14	Compression Compatibility	Shreds Improperly	Data Loss	Y	1	Program Error	1	Test	2	2	Test	
15	Internet Files Recognition	Recognizes Incorrectly	Data Loss	Y	1	Program Error	1	Test; Debug Trace; Code Review	2	2	Test; Rules Validation	
16	Network Compatibility	Shreds Network	Data Loss	Y	1	Program Error	1	Test	2	2	Test Selected/ Improve Network Coverage	
17	Removes File Name	Damages FS	Data Loss	Y	1	Program Error	1	Test; Debug Trace; Code Review	2	2	Test; Debug Tracing; Code Review	
18	Removes File Name	Fails to Remove	Security Breach	Y	1	Program Error	1	Test; Debug Trace; Code Review	2	2	Test; Debug Tracing; Code Review	

Figure 3–1 *An example of a quality risk analysis document using FMEA*

As you can see, we start—at the left side of the table—with a specific function and then identify failure modes and their possible effects. Criticality is determined based on the effects, as is the severity and priority. Possible causes are listed to enable bug prevention work during requirements, design, and implementation.

Next, we look at detection methods—those methods we expect to be applied for this project. The more likely the failure mode is to escape detection, the worse the detection number. We calculate a risk priority number based on the severity, priority, and detection numbers. Smaller numbers are worse. Severity, priority, and detection each range from 1 to 5, so the risk priority number ranges from 1 to 125.

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

During testing actions associated with a risk item, we'd expect that the number of test cases, the amount of test data, and the degree of test coverage would

all increase as the risk increases. Notice that we can allow any test procedures that cover a risk item to inherit the level of risk from the risk item. That documents the priority of the test procedure, based on the level of risk.

3.9.6 Risk-Based Testing throughout the Lifecycle

We've mentioned that, properly done, risk-based testing manages risk throughout the lifecycle. 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.

3.9.7 Risk-Aware Testing Standards

As you saw in chapter 2, the United States Federal Aviation Administration's DO-178B standard bases the extent of testing—measured in terms of white-box code coverage—on the potential impact of a failure. That makes DO-178B 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 for determing 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, a software test specification, software test results, software safety validation, a verification report, and a 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 techniques are ones we'll cover in this book and in the companion volume for test analysis (volume I) that address various functional and black-box testing design techniques. Many of the techniques are ones that we'll cover in this book, including probabilistic testing, 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, continuing on with the 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. Structural coverage requirements are implied, again based on the safety integrity level. (This is similar to DO-178B.) 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 the controls that movement.

3.9.8 Risk-Based Testing Exercise 1

Read the HELLOCARM system requirements document in Appendix B, a hypothetical project derived from a real project that RBCS helped to test.

If you're working in a classroom, break into groups of three to five. If you are working alone, you'll need to do this yourself. Perform quality risks analysis for this project. Since this is a book focused on non-functional testing, identify and assess risks for efficiency quality characteristics only. Use the templates shown in figure 3-2 and table 3-1 on the following page.

To keep the time spent reasonable, I suggest 30 to 45 minutes identifying quality risks, then 15 to 30 minutes altogether assessing the level of each risk. If you are working in a classroom, once each group has finished its analysis, discuss the results.

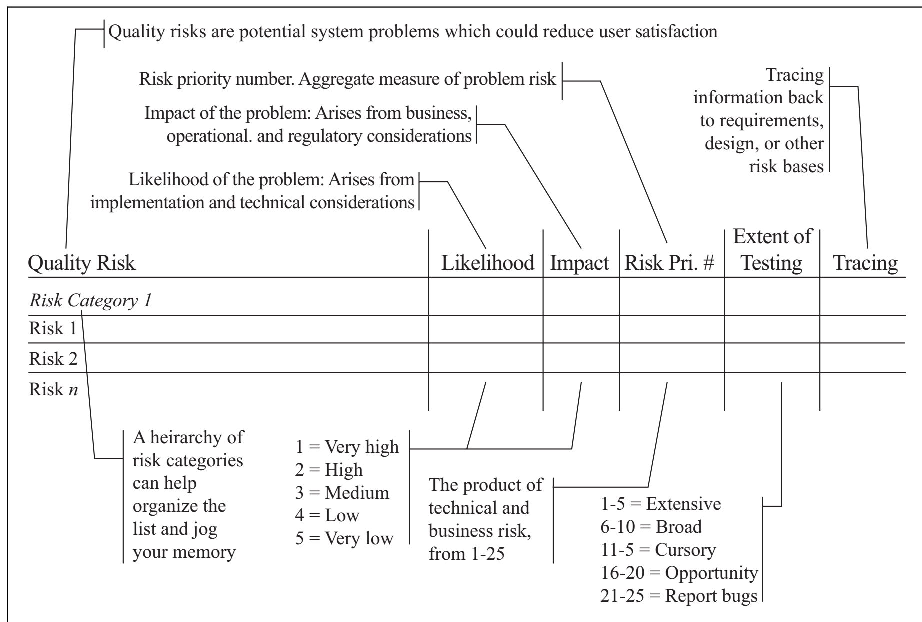


Figure 3–2 *Annotated template for informal quality risk analysis*

Table 3–1 Functional quality risk analysis template showing quality risk categories to address

No.	Quality Risk	Likeli-hood	Im-pact	Risk Pri. #	Extent of Testing	Tracing
1.1.000	Efficiency: Time behavior					
1.1.001	*[Non-functional risks related to time behavior go in this section.]*					
1.1.002						
1.1.003						
1.1.004						
1.1.005						
1.2.000	Efficiency: Resource utilization					
1.2.001	*[Non-functional risks related to accuracy go in this section.]*					
1.2.002						
1.2.003						

3.9.9 Risk-Based Testing Exercise Debrief 1

You can see our solution to the exercise starting in table 3-2. Immediately after that table are two lists of by-products. One is the list of project risks discovered during the analysis. The other is the list of requirements document defects discovered during the analysis.

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

Table 3–2 Functional quality risk analysis for HELLOCARMS

No.	Quality Risk	Likeli-hood	Im-pact	Risk Pri. #	Extent of Testing	Tracing
1.1.000	*Efficiency: Time behavior*					
1.1.001	Screen-to-screen response is continually slow for home equity loans	4	2	8	Broad	040-010-010
1.1.002	Screen-to-screen response is continually slow for home equity lines of credit	4	3	12	Cursory	040-010-010
1.1.003	Screen-to-screen response is continually slow for reverse mortgages	4	4	16	Opportun ity	040-010-010

1.1.004	Approval or decline of loan continually slower than required for all applications	4	2	8	Broad	040-010-020
1.1.005	Approval or decline of high-value loans slower than required	3	2	6	Broad	040-010-020
1.1.006	Loans regularly fail to be processed within one hour	3	3	9	Broad	040-010-030
1.1.007	Time overhead on scoring mainframe does not meet requirements when running at 2,000 applications per hour.	3	4	12	Cursory	040-010-040
1.1.008	Time overhead on scoring mainframe does not meet requirements when running at 4,000 applications per hour	2	4	8	Extensive	040-010-040
1.1.009	Credit-worthiness of customers is not determined in timely manner, especially when working at high rates of throughput	3	2	6	Broad	040-010-050
1.1.010	Fewer than rated Credit Bureau Mainframe requests are completed within the required timeframe at high rates of throughput	2	2	4	Extensive	040-010-050
1.1.011	At high rates of throughput, archiving application is slow, tying up Telephone Banker workstation appreciably	4	2	8	Broad	040-010-060
1.1.012	Escalation to Senior Banker or return to standard Telephone Banker takes longer than rated time when working at high throughput.	4	3	12	Cursory	040-010-070 040-010-080
1.1.013	Conditional confirmation of acceptance takes appreciably more time at high throughput (2,000 per hour)	3	2	6	Broad	040-010-090
1.1.014	Conditional confirmation of acceptance takes appreciably more time at high throughput (4,000 per hour)	2	2	4	Extensive	040-010-090

1.1.015	Abort function does not reset the system to the starting point on the banker's workstation in preparation for the next customer in a timely manner	5	4	20	Report Bugs	040-010-100
1.1.016	Unable to support Internet operations within a timely manner	5	4	20	Opportun ity	010-010-170
1.1.017	Unable to sustain a 2,000 application per hour throughput	*2*	*2*	*4*	Extensive	*040-010-110*
1.1.018	Unable to sustain a 4,000 application per hour throughput	3	3	9	Broad	040-010-120
1.1.019	Unable to support 4,000 concurrent applications	3	4	12	Cursory	040-010-130
1.1.020	Unable to sustain a rate of throughput that would facilitate 1.2 applications the first year	3	2	6	Broad	040-010-140
1.1.021	Turnaround time is prohibitive on transactions (including network transmission time)	4	4	16	Opportun ity	040-010-150
1.2.000	*Efficiency: Resource utilization*					
1.2.001	Database server unable to handle the rated load	4	3	12	Cursory	040-020-010
1.2.002	Web server unable to handle the rated load	3	3	9	Broad	040-020-020
1.2.003	App server unable to handle the rated load	2	3	6	Broad	040-020-030

3.9.10 Project Risk By-Products

In the course of preparing the quality risk analysis document, Jamie observed the following project risk inherent in the requirements.

> *What is Service level agreement (SLA) contract with the Credit Bureau Mainframe? Many efficiency measurements will be dependent on fast turnaround there.*

3.9.11 Requirements Defect By-Products

In the course of preparing the quality risk analysis document, Jamie observed the following defect in the requirements.

1. Assumption appears to be that a certain percentage of applications will be approved or conditionally approved (see 040-010-140, 040-010-150, and 040-010-160). I see no proof that this is a logical necessity.

3.9.12 Risk-Based Testing Exercise 2

Using the quality risks analysis for HELLOCARMS, outline a set of non-functional test suites to run for HELLOCARM system integration and system testing.

Specifically, the system integration test level should have, as an overall objective, looking for defects in and building confidence around the ability of the HELLOCARMS application to work with the other applications in the datacenter efficiently. The system test level should have, as an overall objective, looking for defects in and building confidence around the ability of the HELLOCARMS application to provide the necessary end-to-end capabilities efficiently.

Again, if you are working in a classroom, once each group has finished its work on the test suites and guidelines, discuss the results.

3.9.13 Risk-Based Testing Exercise Debrief 2

Based on the quality risk analysis Jamie performed, he created the following lists of system integration test suites and system test suites.

System Integration Test Suites

- HELLOCARMS/Scoring Mainframe Interfaces
- HELLOCARMS/LoDoPS Interfaces
- HELLOCARMS/GLADS Interfaces
- HELLOCARMS/GloboRainBQW Interfaces

System Test Suites

- Time behavior / Home equity loans
- Time behavior / Home equity lines of credit
- Time behavior / Reverse mortgages
- Resource utilization / Mix of all three

ISTQB Glossary

session-based test management: A method for measuring and managing session-based testing, e.g., exploratory testing.

3.9.14 Test Case Sequencing Guidelines

For system integration testing, we are going to create a suite for each of the different systems that interact with HELLOCARMS. The suite will consist of all the transactions that are possible between the systems.

For system testing, we may be using overkill between the time behavior suites. It is unclear to us at this time in the software development lifecycle how much commonality there may be between the three different products. By the time we get ready to test, we may merge those suites to test all loan products together. For resource utilization, we definitely will mix the different products together to get a realistic test set.

The question of sequencing the tests should be addressed. In performance testing, prioritization must include not only the risk analysis, but also the availability of data, functionality, tools, and resources. Often the resources are problematic because we will be including network, database, and application experts, and perhaps others.

3.10 Failure Mode and Effects Analysis

Learning objectives

Recall of content only

The concepts in this section apply primarily for test managers. There are no learning objectives defined for technical test analysts in this section. In the course of studying for the exam, read this section in chapter 3 of the Advanced syllabus for general recall and familiarity only.

3.11 Test Management Issues

Learning objectives

Recall of content only

The concepts in this section apply primarily for test managers. There are no learning objectives defined for technical test analysts in this section. In the course of studying for the exam, read this section in chapter 3 of the Advanced syllabus for general recall and familiarity only.

3.12 Sample Exam Questions

To end each chapter, you can try one or more sample exam questions to reinforce your knowledge and understanding of the material and to prepare for the ISTQB Advanced Level 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 of 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; 10 percent of these invalid commands 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 follows the IEEE 829 documentation standard, which of the following statements about the IEEE 829 templates could be correct?

   A I belongs in the Test Design Specification, while II belongs in the Test Plan

   B I belongs in the Test Plan, while II belongs in the Test Design Specification

   C I belongs in the Test Case Specification, while II belongs in the Test Plan

   D I belongs in the Test Design Specification, while II belongs in the Test Item Transmittal Report

2. 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.

3. 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 questions 1 and 2 above, and in your answers to

those questions. 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.

4 Test Techniques

"Let's just say I was testing the bounds of reality. I was curious to see what would happen. That's all it was: curiosity."

Jim Morrison of the Doors, an early boundary analyst

The fourth chapter of the Advanced syllabus is concerned with test techniques. To make the material manageable, the syllabus uses a taxonomy—a hierarchical classification scheme—to divide the material into sections and subsections. Conveniently for us, it uses the same taxonomy of test techniques given in the Foundation syllabus, namely specification-based, structure-based, and experience-based, adding additional techniques and further explanation in each category. In addition, the Advanced syllabus also discusses both static and dynamic analysis.[1]

This chapter contains six sections.

1. Introduction
2. Specification-Based
3. Structure-Based
4. Defect- and Experience-Based
5. Static Analysis
6. Dynamic Analysis

We'll look at each section and how it relates to test analysis.

---

1. You might notice a slight change from the organization of the Foundation syllabus here. The Foundation syllabus covers static analysis in the material on static techniques in chapter 3. The Advanced syllabus covers reviews (one form of static techniques) in chapter 6. The Advanced syllabus covers static analysis (another form of static techniques) and dynamic analysis in chapter 4 and, in terms of tools, in chapter 9.

4.1 Introduction

Learning objectives

Recall of content only

In this chapter and the next two chapters, we cover a number of test design techniques. Let's start by putting some structure around these techniques, and then we'll tell you which ones are in scope for the test analyst and which are in scope for the technical test analyst.

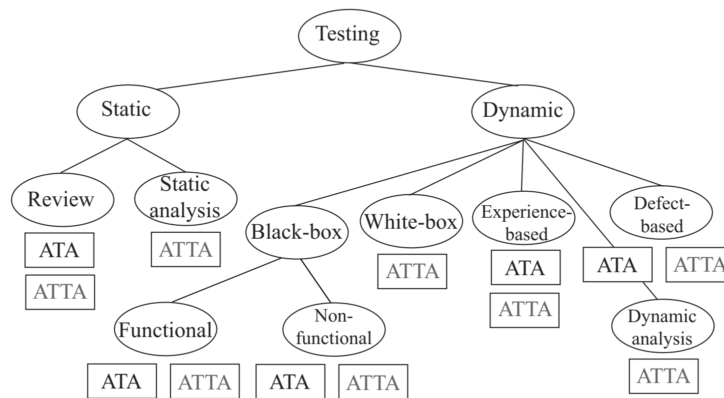


Figure 4–1 *A taxonomy for Advanced syllabus test techniques*

Figure 4-1 shows the highest-level breakdown of testing techniques as the distinction between static and dynamic tests. Static tests, you'll remember from the Foundation syllabus, are those that do not involve running (or executing) the test object itself, while dynamic tests are those that do involve running the test object.

Static tests are broken down into reviews and static analysis. A review is any method where the human being is the primary defect finder and scrutinizer of the item under test, while static analysis relies on a tool as the primary defect finder and scrutinizer.

Dynamic tests are broken down into five main types in the Advanced level:

- Black-box (also referred to as specification-based or behavioral), where we test based on the way the system is supposed to work. Black-box tests can

further be broken down in two main subtypes, functional and non-functional, following the ISO 9126 standard. The easy way to distinguish between functional and non-functional is that functional is testing *what* the system does and non-functional is testing *how* or *how well* the system does what it does.

- White-box (also referred to as structural), where we test based on the way the system is built.
- Experience-based, where we test based on our skills and intuition, along with our experience with similar applications or technologies.
- Defect-based, where we use our understanding of the type of defect targeted by a test as the basis for test design, with tests derived systematically from what is known about the defect.
- Dynamic analysis, where we analyze an application while it is running, usually via some kind of instrumentation in the code.

There's always a temptation, when presented with a taxonomy—a hierarchical classification scheme—like this one, to try to put things into neat, exclusive categories. If you're familiar with biology, you'll remember that every living being, from herpes virus to trees to chimpanzees, has a unique Latin name based on where in the big taxonomy of life it fits. However, when dealing with testing, remember that these techniques are complementary. You should use whichever and as many as are appropriate for any given test activity, whatever level of testing you are doing.

Now, in figure 4-1, below each bubble that shows the lowest-level breakdown of test techniques, you see one or two boxes. Boxes with ATA in them represents test techniques covered by the book *Advanced Software Testing Vol. 1*. Boxes with ATTA in them represent test techniques covered by this book. The fact that most techniques are covered by both books does not, however, mean that the coverage is the same.

You should also keep in mind that this assignment of types of techniques into particular roles is based on common usage, which might not correspond to your own organization. You might have the title of test analyst and be responsible for dynamic analysis. You might have the title of technical test analyst and not be responsible for static analysis at all.

Okay, so that gives you an idea of where we're headed, basically for the rest of this chapter. Figure 4-1 gives an overview of the heart of this book, probably 65 percent of the material covered. In this chapter, we cover dynamic testing, except for non-functional tests, which are covered in the next chapter. A subsequent chapter covers static testing.

You might want to make a copy of figure 4-1 and have it available as you go through this chapter and the next two, as a way of orienting yourself to where we are in the book.

4.2 Specification-Based

Learning objectives

(K2) List examples of typical defects to be identified by each specific specification-based technique.

(K3) Write test cases from a given software model in real life using the following test design techniques (the tests shall achieve a given model coverage):

- Equivalence partitioning
- Boundary value analysis
- Decision tables
- State transition testing

(K4) Analyze a system, or its requirement specification, in order to determine which specification-based techniques to apply for specific objectives, and outline a test specification based on IEEE 829, focusing on component and non-functional test cases and test procedures.

Let's start with a broad overview of specification-based tests before we dive into the details of each technique.

In specification-based testing, we are going to derive and select tests by analyzing the test basis. Remember that the test basis is—or, more accurately, the test bases are—the documents that tell us, directly or indirectly, how the component or system under test should and shouldn't behave, what it is required to do, and how it is required to do it. These are the documents we can base the tests on. Hence the name, test basis. Note also that, while the ISTQB glossary

ISTQB Glossary

black-box testing: Testing, either functional or non-functional, without reference to the internal structure of the component or system.

specification: A document that specifies, ideally in a complete, precise and verifiable manner, the requirements, design, behavior, or other characteristics of a component or system and, often, the procedures for determining whether these provisions have been satisfied.

specification-based technique (or black-box test design technique): Procedure to derive and/or select test cases based on an analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.

test basis: All documents from which the requirements of a component or system can be inferred. The documentation on which the test cases are based. If a document can be amended only by way of formal amendment procedure, then the test basis is called a frozen test basis.

defines the test basis to consist of documents, it is not uncommon to base tests on conversations with users, business analysts, architects, and other stakeholders or on reviews of competitors' systems.

It's probably worth a quick compare-and-contrast with the term *test oracle*, which is similar and related but not the same as the test basis. The test oracle is anything we can use to determine expected results that we can compare with the actual results of the component or system under test. Anything that can serve as a test basis can also be a test oracle, of course. However, an oracle can also be an existing system, either a legacy system being replaced or a competitor's system. An oracle can also be someone's specialized knowledge. An oracle should not be the code because otherwise we are only testing whether the compiler works.

For structural tests—which are covered in the next section—the internal structure of the system is the test basis but not the test oracle. However, for specification-based tests, we do not consider the internal structure at all—at least theoretically.

Beyond being focused on behavior rather than structure, what's common in specification-based test techniques? Well, for one thing, there is some model, whether formal or informal. The model can be a graph or a table. For each

ISTQB Glossary

test oracle: A source to determine expected results to compare with the actual result of the software under test. An oracle may be the existing system (for a benchmark), other software, a user manual, or an individual s specialized knowledge, but it should not be the code.

model, there is some systematic way to derive or create tests using it. And, typically, each technique has a set of coverage criteria that tell you, in effect, when the model has run out of interesting and useful test ideas. It's important to remember that fulfilling coverage criteria for a particular test design technique does not mean that your tests are in any way complete or exhaustive. Instead, it means that the model has run out of useful tests to suggest based on that technique.

There is also typically some family of defects that the technique is particularly good at finding. Boris Beizer, in his books on test design, referred to this as the "bug hypothesis". He meant that, if you hypothesize that a particular kind of bug is likely to exist, you could then select the technique based on that hypothesis. This provides an interesting linkage with the concept of defect-based testing, which we'll cover in a later section of this chapter.[2]

Often, specification-based tests are requirements based. Requirements specifications often describe behavior, especially functional behavior. (The tendency to underspecify non-functional requirements creates a set of quality-related problems that we'll not address at this point.) So, we can use the description of the system behavior in the requirements to create the models. We can then derive tests from the models.

2. Boris Beizer's books on this topic would include *Black-Box Testing* (perhaps most pertinent to test analysts), *Software System Testing and Quality Assurance* (good for test analysts, technical test analysts, and test managers), and *Software Test Techniques* (perhaps most pertinent to technical test analysts).

ISTQB Glossary

requirements-based testing: An approach to testing in which test cases are designed based on test objectives and test conditions derived from requirements, e.g., tests that exercise specific functions or probe non-functional attributes such as reliability or usability.

4.2.1 Equivalence Partitioning

We start with the most basic of specification-based test design techniques, equivalence partitioning. Conceptually, equivalence partitioning involves testing various groups that are expected to be handled the same way by the system and to exhibit similar behavior.

The underlying model is a graphical or mathematical one that identifies equivalent classes—which are also called equivalence partitions—of inputs, outputs, internal values, time relationships, calculations, or just about anything else of interest. These classes or partitions are called *equivalent* because they are likely to be handled the same way by the system. Some of the classes can be called *valid equivalence classes* because they describe valid situations that the system should handle normally. Other classes can be called *invalid equivalence classes* because they describe invalid situations that the system should reject or at least escalate to the user for correction or exception handling.

Once we've identified the equivalence classes, we can derive tests from them. Usually, we are working with more than one set of equivalence classes at one time; for example, each input field on a screen has its own set of valid and invalid equivalence classes. So, we can create one set of valid tests by selecting one valid member from each equivalence partition. We continue this process until each valid class for each equivalence partition is represented in at least one valid test.

Next, we can create negative tests (using invalid data). In most applications, for each equivalence partition, we select one invalid class for one equivalence partition and a valid class for every other equivalence partition. This rule—don't combine multiple invalid equivalent classes in a single test—prevents us from running into a situation where the presence of one invalid value might mask the incorrect handling of another invalid value. We continue this process until each invalid class for each equivalence partition is represented in at least one invalid test.

ISTQB Glossary

equivalence partition: A portion of an input or output domain for which the behavior of a component or system is assumed to be the same, based on the specification.

equivalence partitioning: A black-box test design technique in which test cases are designed to execute representatives from equivalence partitions. In principle, test cases are designed to cover each partition at least once.

However, some applications (particularly web-based applications) are now designed so that, when processing an input screen, they evaluate all of the inputs before responding to the user. With these screens, multiple invalid values are bundled up and returned to the user all at the same time, often by highlighting erroneous values or the controls that contain incorrect values. Careful selection of data is still required, even though multiple invalid values may be tested concurrently if we can check that the input validation code acts independently on each field.

Notice the coverage criterion implicit in the preceding discussion. Every class member, both valid and invalid, is represented in at least one test case.

What is our bug hypothesis with this technique? For the most part, we are looking for a situation where some equivalence class is handled improperly. That could mean the value is accepted when it should have been rejected or vice versa, or that a value is properly accepted or rejected but handled in a way appropriate to another equivalence class, not the class to which it actually belongs.

You might find this concept a bit confusing verbally, so let's try some figures. Figure 4-2 shows a way that we can visualize equivalence partitioning.

As you can see in the top half of the figure 4-2, we start with some set of interest. This set of interest can be an input field, an output field, a test precondition or postcondition, a configuration, or just about anything we're interested in testing. The key is that we can apply the operation of equivalence partitioning to split the set into two or more disjoint subsets, where all the members of each subset share some trait in common that was not shared with the members of the other subset.

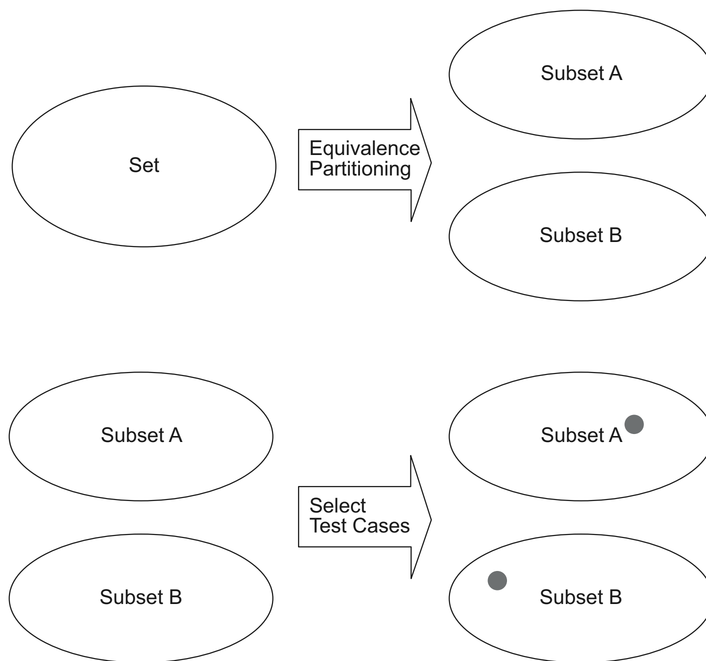


Figure 4–2 *Visualizing equivalence partitioning*

For example, if you have a simple drawing program that can fill figures with red, green, or blue, you can split the set of fill colors into three disjoint sets: red, green, and blue.

In the bottom half of the figure 4-2, we see the selection of test case values from the subsets. The dots in the subsets represent the value chosen from each subset to be represented in the test case. This involves selecting at least one member from each subset. In pure equivalence partitioning, the logic behind the specific selection is outside the scope of the technique. In other words, you can select any member of the subset you please. If you're thinking, "Some members are better than others," that's fine; hold that thought for a few minutes and we'll come back to it.

Now, at this point we'll generate the rest of the test case. If the set that we partitioned was an input field, we might refer to the requirements specification to understand how each subset is supposed to be handled. If the set that we partitioned was an output field, we might refer to the requirements to derive inputs

that should cause that output to occur. We might use other test techniques to design the rest of the test cases.

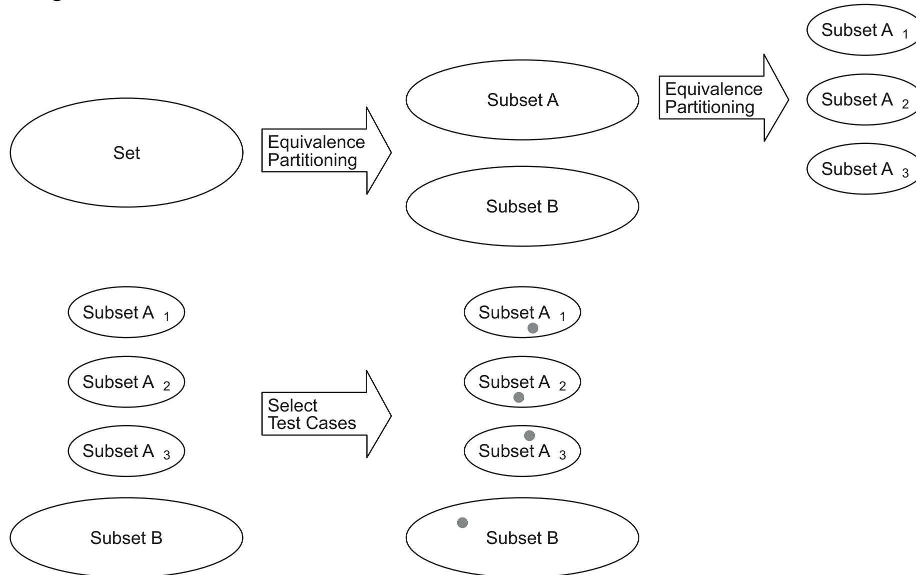


Figure 4–3 *Subpartitioning an equivalence class*

Figure 4-3 shows that equivalence partitioning can be applied iteratively. In this figure, we apply a second round of equivalence partitioning to one of the subsets to generate three smaller subsets. Only at that point do we select four members—one from subset B and one each from subset A1, A2, and A3—for test cases. Note that we don't have to select a member from subset A since each of the members from subsets A1, A2, and A3 are also members of subset A.

4.2.1.1 Avoiding Equivalence Partitioning Errors

While equivalence partitioning is fairly straightforward, people do make some common errors when applying it. Let's look at these errors so you can avoid them.

First, as shown in the top half of figure 4-4, the subsets must be *disjoint*. That is, no two of the subsets can have one or more member in common. The whole point of equivalence partitioning is to test whether a system handles different situations differently (and properly, of course). If it's ambiguous as to which handling is proper, then how do we define a test case around this? Try it out and see what happens? Not much of a test!

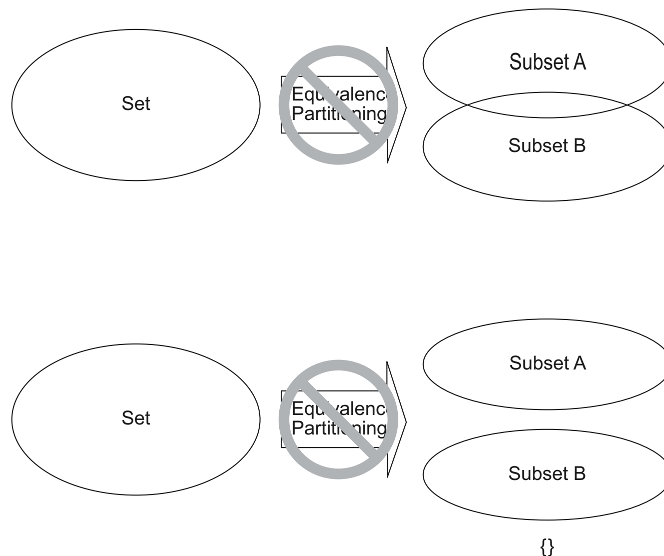


Figure 4–4 *Common equivalence partitioning errors*

Second, as shown in the bottom half of figure 4-4, none of the subsets may be *empty*. That is, if the equivalence partitioning operation produces a subset with no members, that's hardly very useful for testing. We can't select a member of that subset because it has no members.

Third, while not shown graphically—in part because we couldn't figure out a clear way to draw the picture—note that the equivalence partitioning process does not subtract, it divides. What we mean by this is that, in terms of mathematical set theory, the union of the subsets produced by equivalence partitioning must be the same as the original set that was equivalence partitioned. In other words, equivalence partitioning does not generate "spare" subsets that are somehow disposed of in the process—at least, not if we do it properly.

Notice that this is important because, if this is not true, then we stand the chance of failing to test some important subset of inputs, outputs, configurations, or some other factor of interest that somehow was dropped in the test design process.

4.2.1.2 Composing Test Cases with Equivalence Partitioning

When we compose test cases in situations where we've performed equivalence partitioning on more than one set, we select from each subset as shown in figure 4-5.

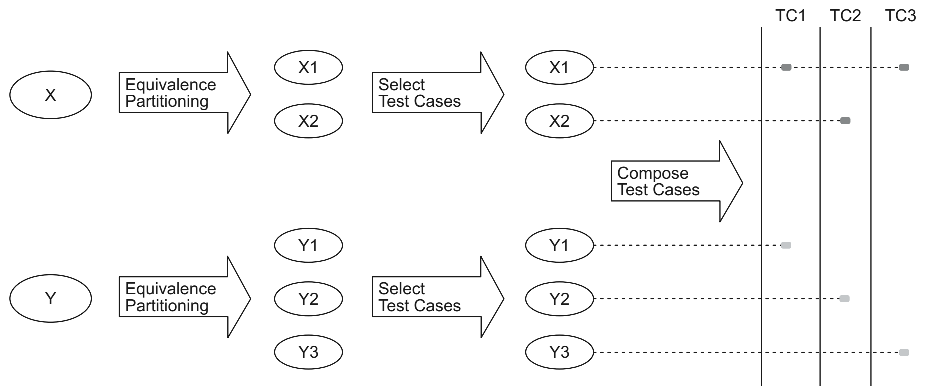


Figure 4–5 *Composing test cases with valid values*

Here, we start with set X and set Y. We partition set X into two subsets, X1 and X2. We partition set Y into three subsets, Y1, Y2, and Y3. We select test case values from each of the five subsets, X1 and X2 on the one hand, Y1, Y2, and Y3 on the other. We then compose three test cases since we can combine the values from the X subsets with values from the Y subsets (assuming the values are independent and all valid).

For example, imagine you are testing a browser-based application. You are interested in two browsers: Internet Explorer and Firefox. You are interested in three connection types: dial-up, DSL, and cable modem. Since the browser and the connection types are independent, we can create three test cases. In each of the test cases, one of the connection types and one of the browser types will be represented. One of the browser types will be represented twice.

When discussing figure 4-5, we made a brief comment that we can combine values across the equivalence partitions when the values are independent and all valid. Of course, that's not always the case.

In some cases, values are not independent, in that the selection of one value from one subset constrains the choice in another subset. For example, imagine if we are trying to test combinations of applications and operating systems. We can't test an application running on an operating system if there is not a version of that application available for that operating system.

In some cases, values are not valid. For example, in figure 4-6, imagine that we are testing a project management application, something like Microsoft Project. Suppose that set X is the type of event we're dealing with, which can be

either a task (X1) or a milestone (X2). Suppose that set Y is the start date of the event, which can be in the past (Y1), today (Y2), or in the future (Y3). Suppose that set Z is the end date of the event, which can be either on or after the start date (Z1) or before the start date (Z2). Of course, Z2 is invalid, since no event can have a negative duration.

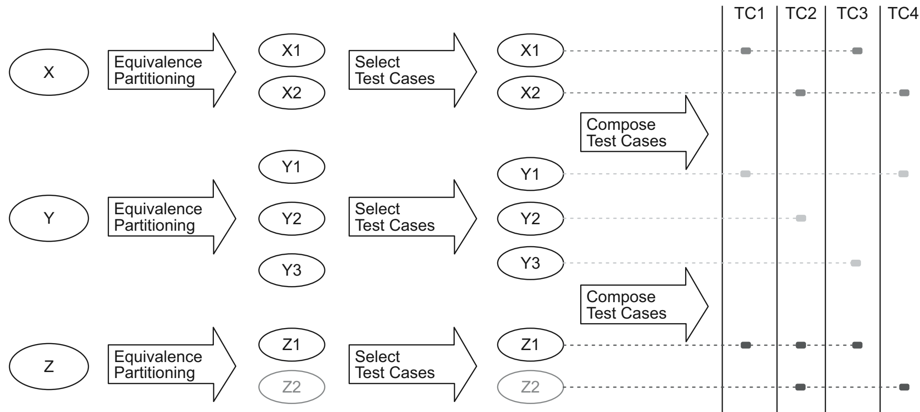


Figure 4–6 *Composing tests with invalid values*

So, we test combinations of tasks and milestones with past, present, and future start dates and valid end dates in test cases TC1, TC2, and TC3. In TC4, we check that illegal end dates are correctly rejected. We try to enter a task with a start date in the past and an end date prior to the start date. If we wanted to subpartition the invalid situation, we could also test with a start date in the present and one in the future together with an end date before the start date, which would add two more test cases. One would test a start date in the present and an end date prior to the start date. The other would test a start date in the future and an end date prior to the start date.

In this particular example, we had a single subset for just one of the sets that was invalid. The more general case is that many—perhaps all—of the fields will have invalid subsets. Imagine testing an e-commerce application. On the checkout screens of the typical e-commerce application, there are multiple required fields, and usually there is at least one way for such a field to be invalid.

When there are multiple invalid values, there are two possible ways of testing them, depending on how the error handling is designed in the application.

Historically, most error handling is done by parsing through the values and immediately stopping when the first error is found. Some newer applications parse through the entire set of values first, and then compile a list of errors found.

Whichever error handler is used, we can always run test cases with a single invalid value entered. That way, we can check that every invalid value is correctly rejected or otherwise handled by the system. Of course, the number of test cases required by this method will equal the number of invalid partitions.

When we have an error handler that aggregates all of the errors in one pass, we could test it by submitting multiple invalid values and thence reduce the total number of test cases.

In a safety-critical or mission-critical system, you might want to test combinations of invalid values even if the parser stops on the first error. Just remember, anytime you start down the trail of combinatorial testing, you are taking a chance that you'll spend a lot of time testing things that aren't terribly important.

Consider a simple example of equivalence partitioning for system configuration. On the Internet appliance project we've mentioned, there were four possible configurations for the appliances. They could be configured for kids, teens, adults, or seniors. This configuration value was stored in a database on the Internet service provider's server so that, when an Internet appliance is connected to the Internet, this configuration value became a property of its connection.

Based on this configuration value, there were two key areas of difference in the expected behavior. For one thing, for the kids and teens systems, there was a filtering function enabled. This determined the allowed and disallowed websites the system could surf to. The setting was most strict for kids and somewhat less strict for teens. Adults and seniors, of course, were to have no filtering at all and should be able to surf anywhere.

For another thing, each of the four configurations had a default set of e-commerce sites they could visit called the mall. These sites were selected by the marketing team and were meant to be age appropriate.

Of course, these were the expected differences. We were also aware of the fact that there could be weird unexpected differences that could arise because

that's the nature of some types of bugs. For example, performance was supposed to be the same, but it's possible that performance problems with the filtering software could introduce perceptible response-time issues with the kids and teens systems. We had to watch for those kinds of misbehaviors.

So, to test for the unexpected differences, we simply had at least one of each configuration in the test lab at all times and spread the nonconfiguration-specific tests more or less randomly across the different configurations. To test expected differences related to filtering and e-commerce, we made sure these configuration-specific tests were run on the correct configuration. The challenge here was that, while the expected results were concrete for the mall—the marketing people gave us four sets of specific sites, one for each configuration—the expected results for the filtering were not concrete but rather logical. This led to an enormous number of disturbing defect reports during test execution as we found creative ways to sneak around the filters and access age-inappropriate materials on the kids and teens configurations.

4.2.1.3 Equivalence Partitioning Exercise

A screen prototype for one screen of the HELLOCARMS system is shown in figure 4-7. This screen asks for three pieces of information:

- The product being applied for, which is one of the following:
  - Home equity loan
  - Home equity line of credit
  - Reverse mortgage
- Whether someone has an existing Globobank checking account, which is either Yes or No
- Whether someone has an existing Globobank savings account, which is either Yes or No

If the user indicates an existing Globobank account, then the user must enter the corresponding account number. This number is validated against the bank's central database upon entry. If the user indicates no such account, the user must leave the corresponding account number field blank.

If the fields are valid, including the account number fields, then the screen will be accepted. If one or more fields are invalid, an error message is displayed.

Apply for Product? {Select a product}
Home equity loan
Home equity line of credit
Reverse mortgage
Existing Checking? {Select Yes or No} {If Yes input account number}
Yes
No
Existing Savings? {Select Yes or No} {If Yes input account number}
Yes
No

Figure 4–7 *HELLOCARMS system product screen prototype*

The exercise consists of two parts:

1. Show the equivalence partitions for each of the three pieces of information, indicating valid and invalid members.
2. Create test cases to cover these partitions, keeping in mind the rules about combinations of valid and invalid members.

The answers to the two parts are shown on the next pages. You should review the answer to the first part (and, if necessary, revise your answer to the second part) before reviewing the answer to the second part.

4.2.1.4 Equivalence Partitioning Exercise Debrief

First, let's take a look at the equivalence partitions.

For the application-product field, the equivalence partitions are as follows:

Table 4–1

#	Partition
1	Home equity loan
2	Home equity line of credit
3	Reverse mortgage

Note that the screen prototype shows this information as selected from a pull-down list, so there is no possibility of entering an invalid product here. Some people do include an invalid product test, but we have not.

For each of two existing-account entries, the situation is best modeled as a single input field, which consists of two subfields. The first subfield is the Yes/No field. This subfield determines the rule for checking the second subfield, which is the account number. If the first subfield is Yes, the second subfield must be a valid account number. If the first subfield is No, the second subfield must be blank.

So, the existing checking account information partitions are as follows:

Table 4–2

#	Partition
1	Yes-Valid
2	Yes-Invalid
3	No-Blank
4	No-Nonblank

And, the existing savings account information partitions are as follows:

Table 4–3

#	Partition
1	Yes-Valid
2	Yes-Invalid
3	No-Blank
4	No-Nonblank

Note that, for both of these, partitions 2 and 4 are invalid partitions, while partitions 1 and 3 are valid partitions.

Just as a side note, we could simplify the interface to eliminate some error handling code, testing, and possible end-user errors. This simplification could be done by designing the interface such that selecting No for either existing checking or savings accounts automatically deleted any value in the respective account number text field and then hid that field. In both table 4-2 and table 4-3, this would have the effect of eliminating partition 4. The user would not be allowed to make the mistakes, therefore the code would not have to handle the exceptions, and therefore the tester would not need to test

them. A win-win-win situation. Often, clever design of the interface can increase the testability of an application in this way.

Now, let's create tests from these equivalence partitions. As we do so, we're going to capture traceability information from the test case number back to the partitions. Once we have a trace from each partition to a test case, we're done—provided that we're careful to follow the rules about combining valid and invalid partitions!

Table 4–4

Inputs	1	2	3	4	5	6	7
Product	HEL	LOC	RM	HEL	LOC	RM	HEL
Existing Checking?	Yes	No	No	Yes	No	No	No
Checking Account	Valid	Blank	Blank	Invalid	Nonblank	Blank	Blank
Existing Savings?	No	Yes	No	No	No	Yes	No
Savings Account	Blank	Valid	Blank	Blank	Blank	Invalid	Nonblank
Outputs							
Accept?	Yes	Yes	Yes	No	No	No	No
Error?	No	No	No	Yes	Yes	Yes	Yes

Table 4–5

#	Partition	Test Case
1	Home equity loan (HEL)	1
2	Home equity line of credit (LOC)	2
3	Reverse mortgage (RM)	3

Table 4–6

#	Partition	Test Case
1	Yes-Valid	1
2	Yes-Invalid	4
3	No-Blank	2
4	No-Nonblank	5

Table 4–7

#	Partition	Test Case
1	Yes-Valid	2
2	Yes-Invalid	6
3	No-Blank	1
4	No-Nonblank	7

> **ISTQB Glossary**
>
> boundary value: An input value or output value which is on the edge of an equivalence partition or at the smallest incremental distance on either side of an edge, such as, for example, the minimum or maximum value of a range.
>
> **boundary value analysis:** A black-box test design technique in which test cases are designed based on boundary values.

You should notice that these test cases do not cover all interesting possible combinations of factors here. For example, we don't test to make sure a person with both a valid savings and valid checking account work properly. That could be an interesting test because the accounts might have been established at different times and might have information that now conflicts in some way; e.g., in some countries it is still relatively common for a woman to take her husband's last name upon marriage. We also don't test the combination of invalid accounts or the combination of account numbers that are valid alone but not valid together; e.g., the two accounts belong to entirely different people.

4.2.2 Boundary Value Analysis

Let's refine our equivalence partitioning test design technique with the next technique, boundary value analysis. Conceptually, boundary value analysis is predominately about testing the edges of equivalence classes. In other words, instead of selecting one member of the class, we select the largest and smallest members of the class and test them. We will discuss some other options for boundary analysis later.

The underlying model is again either a graphical or mathematical one that identifies two boundary values at the boundary between one equivalence class and another. (In some techniques, the model identifies three boundary values, which we'll discuss later.) Whether such a boundary exists for subsets where we've performed equivalence partitioning is another question that we'll get to in just a moment. Right now, assuming the boundary does exist, notice that the boundary values are special members of the equivalence classes that happen to be right next to each other and right next to the point where the expected behavior of the system changes. If the boundary values are members of a valid

equivalence class, they are valid, of course, but if members of an invalid equivalence class, they are invalid.

Deriving tests with boundary values as the equivalence class members is much the same as for plain equivalence classes. We test valid boundary values together and then combine one invalid boundary value with other valid boundary values (unless the application being tested aggregates errors).

We have to represent each boundary value in a test case, analogous to the equivalence partitioning situation. In other words, the coverage criterion is that every boundary value, both valid and invalid, must be represented in at least one test.

The main difference is that there are at least two boundary values in each equivalence class. Therefore, we'll have more test cases, about twice as many.

More test cases? That's not something we like unless there's a good reason. What is the point of these extra test cases? That is revealed by the bug hypothesis for boundary value analysis. Since we are testing equivalence class members—every boundary value is an equivalence class member—we are testing for situations where some equivalence class is handled improperly. Again, that could mean acceptance of values that should be rejected, or vice versa, or proper acceptance or rejection but improper handling subsequently, as if the value were in another equivalence class, not the class to which it actually belongs. However, by testing boundary values, we also test whether the boundary between equivalence classes is defined in the right place.

So, do all equivalence classes have boundary values? No, definitely not. Boundary value analysis is an extension of equivalence partitioning that applies only when the members of an equivalence class are ordered.

So, what does that mean? Well, an ordered set is one where we can say that one member is greater than or less than some other member if those two members are not the same. We have to be able to say this meaningfully too. Just because some item is right above or below some other item on a pull-down menu does not mean that, within the program, the two items have a greater-than/less-than relationship.

4.2.2.1 Examples of Equivalence Partitioning and Boundary Values

Let's look at two examples where a technical test analyst can (and can't) use boundary value analysis on equivalence classes: enumerations and arrays.

Many languages allow a programmer to define a set of named values that can be used as a substitute for constant values. These are called enumerations.

As an example, assume that a database stored colors as integers to save space. Red equals 0, blue equals 1, green equals 2. When the programmer wanted to set an object to red, he would have to use the value 0.

These arbitrary values are often called magic numbers because they have a (somewhat) hidden meaning, often known only by the programmer. A programming language that allowed enumeration could allow the programmer to create a mapping of the magic numbers to a textual value, allowing the programmer to write code that is self-documenting.

Instead of writing ColorGun = 0;, the programmer can write ColorGun = Red. In figure 4-8, an example enumeration is made in the C language, defining the five different colors.

```
enum Colors {
      Red,
      Blue,
      Green,
      Yellow,
      Cyan
};
```

Figure 4–8

In some languages, the programmer is given special functions to allow easier manipulation of the enumerated values. For example, Pascal has the functions pred and succ to traverse the values of an enumeration. Of course, trying to "pred" the first or "succ" the last is going to fail, sometimes with unknown symptoms. Hence boundaries that can be tested: both valid and invalid.

The compiler usually manages the access to the values in the enumeration. We should be able to assume (we hope) that if we can access the first and last elements of an enumeration, we should be able to access them all. If the language supports functions that walk the enumerated list, then we should test both valid and invalid boundaries using the list.

In some languages (e.g., C), the order of enumeration may be set by the compiler but the rules used are specific to each compiler and therefore not consistent for testing. Knowing the rules for the programming language and compiler being used is essential.

Arrays are data structures allowing a programmer to store homogenous data and access it via index rather than by name. The data must be homogenous, which means of a single type. In most languages, arrays can consist of elements of programmer-defined types. A defined type may contain any number of different internal elements (including a mix of different data types) but each element in the array must be of that type, whether that type is built in or programmer defined.

Some programming languages allow only static arrays[3] which are never allowed to change in number of elements. Other programming languages allow arrays to be dynamically created (memory for the array is allocated off the heap at runtime). These types of arrays can often be reallocated to allow resizing them on the fly. That can make testing them interesting. Even more interesting is testing what happens after an array is resized. Then, we get issues as to whether pointers to the old array are still accessible, whether data get moved correctly (or lost if the array is resized downward).

For languages that do bounds checking of arrays, where an attempt to access an item before or after the array is automatically prevented, testing of boundaries may be somewhat less important. However, other issues may be problematic. For example, if the system throws an exception while resizing an array, what should happen?

Many languages, however, are more permissive when it comes to array checking. They allow the programmer the freedom to make mistakes. And, very often, those mistakes often come at the boundaries.

In some operating and runtime systems, an attempt to access memory beyond the bounds of the array will immediately result in a runtime failure that is immediately visible. But not all. A common problem occurs when the runtime system allows a write action to memory not owned by the same process or thread that owns the array. The data being held in that location beyond the array are overwritten. When the actual owning process subsequently tries to use the overwritten memory, the failure may then become evident (i.e., total failure or freeze up) or the memory's contents might be used in a calculation (resulting in the wrong answer), or it might be stored in a file or database, corrupting the data store silently. None of those are attractive results; thus worthy of testing.

---

3. Those created and sized at compile time

There are also security issues to this array overwrite that we will discuss in a later chapter.

Multidimensional arrays, where an item is accessed through two or more indices, can be arbitrarily complex, making all of the issues mentioned above even worse.

Dynamic analysis tools can be useful when testing arrays; they can catch the failure at the first touch beyond the bounds of the array rather than waiting for the symptom of the failure to show. We will discuss dynamic analysis tools later in this chapter and in chapter 9.

4.2.2.2 Non-functional Boundaries

Boundaries are sometimes simple to find. For example, when dealing with simple data types like integers, where the values are ordered, there is no need to guess. The boundaries are self-evident. Other boundaries may be somewhat murky as you will see when we discuss integer and floating point data types.

And then there are times when a boundary is only implicitly defined and the tester must settle on ways to test it. Consider the following requirement:

The maximum file size is 16 MB.

Exactly how many bytes is 16 MB? How do we create a 16 MB file to test the maximum? How do we test the invalid high boundary? The answer to the first question as to the number of bytes is...it depends. What is the operating system? What is the file system under use: FAT32, NTFS? What are the sector and cluster sizes on the hard disk? Testing the exact valid high boundary on a 16 MB file is very difficult, as is testing the invalid high boundary. When we are testing such a system—assuming it is not a critical or safety-related system—we will often not spend time trying to ascertain the exact size but will test a file as close to 16 MB as we can get for the valid high boundary and perhaps a 17 MB file for the invalid high boundary.

On the other end of the file size, the invalid low boundary is relatively easy to comprehend: there is none. No file can be minus one (-1) byte. But perhaps the invalid low must be compared to the valid low size. Is it zero bytes? Some file systems and operating systems allow zero-sized files, and therefore there is no invalid low boundary size file possible. If the file is a structured file, with defined header information, the application that created it likely will not allow a zero-byte file, and therefore an invalid low boundary is testable.

In our experience, you should take these types of questions to the development team if they were not already defined in the design specification. Be aware, however, that often the developers have not given sufficient thought to some of these questions; their answers might be incorrect. Often, experimenting is required once the system arrives.

As a separate example, assume the following efficiency requirement:

The system shall process at least 150 transactions per second.

An invalid low boundary may not exist. It would, of course, be impossible to trigger a negative number of transactions; however, if the system is designed such that a minimum number of transactions must flow within a given unit of time, there would be both valid and invalid low boundaries. Defining the minimum should be approached as we did the file example.

Is a valid high really a boundary though? Clearly we must test 150 transactions per second, consistent with good performance testing—we will discuss that later in the book. What would be an invalid high? In a case like this, we would argue that there is no specific high invalid boundary to test; however, good performance testing would continue to ramp up the number of transactions until we see an appreciable change in performance.

The point of this discussion is that boundaries are really useful for testing; sometimes you just must search a bit to determine what those boundaries are.

4.2.2.3 A Closer Look at Functional Boundaries

In the Foundation syllabus, we discussed a few boundary examples in a somewhat simplistic way. For an integer where the valid range is 1 to 10 inclusive, the boundaries were discussed as if there were four. These are shown on the number line in figure 4-9 as 0 (invalid low), 1 (valid low), 10 (valid high), and 11 (invalid high).

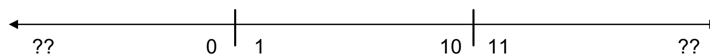


Figure 4–9 *Boundaries for integers*

However, the closer you look, the more complicated it actually gets. Looking at this number line, there appear to be two missing values that we did not discuss: on the far left and far right. In math, we would say that the two missing bound-

aries represent negative infinity on the left, positive infinity on the right. Clearly a computer cannot represent infinity using integers, as that would take an infinite number of bytes. That means there are going to be specific values that are both the largest and the smallest values that the computer can represent. Boundaries! In order to understand how these boundaries are actually represented—and where errors can occur—we need to look at some computer architectural details. Exactly how are numbers stored in memory?

Technical test analysts must understand how different values are represented in memory; we will look at this topic for both integer and floating point numbers.

4.2.2.4 Integers

In most computer architectures, integers are stored differently than floating point numbers (those numbers that have fractional values). The maximum size of an integer—and hence the boundaries—(in most cases) is defined by two things: the number of bits that are allocated for the number and whether it is signed or unsigned.

In these architectures, an integer is encoded using the binary number system, a combination of ones and zeroes. The width or precision of the number is the number of bits in its representation. A number with n bits can encode 2^n different numeric values.

If the number is signed, the left-most bit represents the sign; this has the effect of making approximately half of the possible values positive and the other half negative. For example, assume that 8 bits are used to store the number. The range of the unsigned number is from 0 (0000 0000) to +255 (1111 1111). The range of the signed number is from -128 (1000 0000) to +127 (0111 1111). If those actual bit representations look strange, it is because in most computers, negative numbers are represented as two's complements.

A two's complement number is represented as the value obtained by subtracting the number from a large power of two. To get the two's complement of an 8-bit value, the value (in binary) is subtracted from 2^8. The full math and logic behind using two's complement encoding for negative numbers is beyond the scope of this book. The important concept is that the boundaries can be determined for use in testing.

Bits	Name	Range (assuming two's complement for signed)	Decimal digits
4	nibble, semioctet	*Signed:* From -8 to 7, from $-(2^3)$ to $2^3 - 1$	1
		Unsigned: From 0 to 15 which equals to $2^4 - 1$	2
8	byte, octet	*Signed:* From -128 to 127, from $-(2^7)$ to $2^7 - 1$	3
		Unsigned: From 0 to 255 which equals to $2^8 - 1$	3
16	halfword, word, short, short	*Signed:* From $-32{,}768$ to $32{,}767$, from $-(2^{15})$ to $2^{15} - 1$	5
		Unsigned: From 0 to $65{,}535$ which equals to $2^{16} - 1$	5
32	word, long, doubleword, longword, int	*Signed:* From $-2{,}147{,}483{,}648$ to $2{,}147{,}483{,}647$, from $-(2^{31})$ to $2^{31} - 1$	10
		Unsigned: From 0 to $4{,}294{,}967{,}295$ which equals to $2^{32} - 1$	10
64	doubleword, longword, long long, quad, quadword, int64	*Signed:* From $-9{,}223{,}372{,}036{,}854{,}775{,}808$ to $9{,}223{,}372{,}036{,}854{,}775{,}807$, from $-(2^{63})$ to $2^{63} - 1$	19
		Unsigned: From 0 to $18{,}446{,}744{,}073{,}709{,}551{,}615$ which equals to $2^{64} - 1$	20
128	octaword, double quadword	*Signed:* From $-170{,}141{,}183{,}460{,}469{,}231{,}731{,}687{,}303{,}715{,}884{,}105{,}728$ to $170{,}141{,}183{,}460{,}469{,}231{,}731{,}687{,}303{,}715{,}884{,}105{,}727$, from $-(2^{127})$ to $2^{127} - 1$	39
		Unsigned: From 0 to $340{,}282{,}366{,}920{,}938{,}463{,}463{,}374{,}607{,}431{,}768{,}211{,}455$ which equals to $2^{128} - 1$	39
n	*n*-bit integer (general case)	*Signed:* (-2^{n-1}) to $(2^{n-1} - 1)$	$\lceil (n-1) \log_{10} 2 \rceil$
		Unsigned: 0 to $(2^n - 1)$	$\lceil n \log_{10} 2 \rceil$

Figure 4–10 *Whole number representations*[4]

Figure 4-10 shows several possible number sizes and the values that can be represented.

Many mainframes (and some databases) use a different encoding scheme called binary-coded decimals (BCDs). Each digit in a represented number takes up 4 bits. The possible maximum length of a number is proportional to the number of bytes allowed by the architecture. So, if the computer allowed 1,000 bytes for a single number, that number could be 2,000 digits long. By writing

4. Wikipedia at http://en.wikipedia.org/wiki/Integer_(computer_science)

your own handler routines, you could conceivably have numbers of any magnitude in any programming language. The issue would then become computational speed—how long does it take to perform mathematical calculations.

Note that the binary representation is slightly more efficient than the BCD representation as far as the relative size of the number that can be stored. A 128-bit, unsigned value (16 bytes) can hold up to 39 digits, while you would need a BCD of 20 bytes to hold the same size number. While this size differential is not huge, the speed of processing may become paramount; BCD calculations tend to be slower than calculations for built-in data types because the calculation must be done digit by digit.

Most programming languages have a number of different width whole numbers that can be selected by the programmer. The typical default value integer in 16-bit computers was 16 bits; in most 32-bit computers the default is 32 bits wide. For example, in the Delphi programming language, a *short* is 16 bits, an *int* is 32 bits, and an *int64* is 64 bits. A special library can be used in Delphi to support 128-bit numbers. When the programmer declares a variable, they define the length and whether the value is signed or unsigned. While a programmer could conceivably use the largest possible value for each number to minimize the chance of overflow, that has always been considered a bad programming technique that wastes space and often slows down computations.

In C as it was originally defined for DEC minicomputers, the language only guaranteed that an *int* was at least as long (if not longer) as a *short* and a *long* was at least as long (if not longer) as an *int*. On the PDP-11, a *short* was 16 bits, an *int* 32 bits, and a *long* also 32 bits. Other architectures gave other lengths, and even with modern 64-bit CPUs, you might not always know how many bits you'll get with your *int*. This ambiguity about precision in C has created—and continues to create—a lot of bugs and thus is fertile ground for testing.

By looking at the definitions of the data variables used, a technical tester can discern the length of the value and thence the boundaries. Since there is always a defined *maxint* (the maximum size integer) and *minint* (the maximum magnitude negative number)—no matter which type variable is used—it is relatively straightforward to find and test the boundaries. Assuming an 8-bit, unsigned integer 255 (1111 1111), adding 1 to it would result in an overflow, giving the value 0 (0000 0000). 255 + 1 = 0. Probably not what the programmer intended...

Since any value inputted to the system may be used in an expression where a value is added to it, testing with very large and very small numbers based on the representation used by the programmer should be considered a good test technique. Because division and multiplication happens too, be sure to test zero, which is always an interesting boundary value.

4.2.2.5 Floating Point Numbers

The other types of numbers we need to discuss are floating point (floats). These numbers allow a fractional value; some number of digits before and some number of digits after a decimal point. As before, a technical tester needs to investigate the architecture and selections made by programmers when testing these types of numbers. These numbers are represented by a specific number of significant digits and scaled by using an exponent. This is the computer equivalent of scientific notation. The base is usually 2 but could be 10 or 8. Like scientific notation, floats are used to represent very large or very small numbers. For example, a light year is approximately $9.460536207 \times 10^{15}$ meters. This value is a bit more than even a really large integer can represent. Likewise, an angstrom is approx 1×10^{-15} meters, a very tiny number.

Just to get the terminology correct, a floating point number consists of a signed digit string of a given length in a given base. The signed string (as shown in the preceding paragraph, 9.460536207) is called the significand (or sometimes the mantissa). The length of the significand determines the precision that the float can represent—that is, the number of significant digits. The decimal point, sometimes called the radix, is usually understood to be in a specific place—usually to the right of the first digit of the significand. The base is usually set by the architecture (usually 2 or 10). The exponent (also called the characteristic or scale) modifies the magnitude of the number. In the preceding paragraph, in the first example, the exponent is 15. That means that the number is 9.460536207 times 10 to the 15th power.

Like integers, floats can also be represented by binary coded decimals. These are sometimes called *fixed-point* decimals and are actually undistinguishable from integer BCDs. In other words, the decimal point is not stored in the value; instead, its relative location is understood by the programmer and compiler. For example, assume a 4-byte value: 12 34 56 7C. In fixed-point notation,

this value might represent +1,234.567 if the compiler understands the decimal point to be between the fourth and fifth digits.

In general, non-integer numbers have a far wider range than integers while taking up only a few more bytes. Different architectures have a variety of schemes for storing these values. Like integers, floats come in several different sizes; one common naming terminology is 16 bit (*halfs*), 32 bit (*singles*), 64 bit (*doubles*) and 128 bit (*quadruples*). Over the years, a wide variety of schemes have been used to represent these values; IEEE 754 has standardized most of them. This standard, first adopted in 1985, is followed by almost every computer manufacturer with the exception of IBM mainframes and Cray supercomputers. The larger the representation, the larger (and smaller) the numbers that can be stored.

Type	Sign	Exponent	Significand	Total bits	Exponent bias	Bits precision
Half (IEEE 754-2008)	1	5	10	16	15	11
Single	1	8	23	32	127	24
Double	1	11	52	64	1023	53
Quad	1	15	112	128	16383	113

Figure 4–11 *IEEE 754-2008 float representations*

As shown in figure 4-11, a float contains a sign (positive or negative), an exponent of certain size, and a significand of certain size. The magnitude of the significand determines the number of significant figures the representation can hold (called the precision). Note that the bits precision is actually one more than the significand can hold. For complex reasons having to do with the way floats are used in calculations, the leading bit of the significand is implicit—that is, not stored. That adds the extra bit of precision. Related to that is the bias of the exponent (bias is used in the engineering sense, as in offset). The value of the exponent is offset by a given value to allow it to be represented by an unsigned number even though it actually represents both large (positive) and small (negative) exponents.

A key issue with these values that must be understood by technical testers is that most floats are not exact; instead, they are approximations of the value we may be interested in. In many cases, this is not an issue. Occasionally, however, interesting bugs can lurk in these approximate values.

When values are calculated, the result is often an irrational number. The definition of an irrational number is any real number that cannot be expressed as the fraction a/b where a and b are integers. Since an irrational number, by definition, has an infinite number of decimal digits (i.e., is a non-terminating number), there is no way to represent it exactly by any size floating point number. That means anytime an operation occurs with irrational values, the result will be irrational (and hence approximate). There are many irrational numbers that are very important to mathematic concepts, including pi and the square root of 2.

It is not that a floating point value cannot be exact. Whole numbers may be represented exactly. The representation for 734 would be 7.34×10^2. However, when calculations are made using floating point values, the exactness of the result might depend on how the values were originally set. It is probably safe to hold, as a rule of thumb, that any floating number is only close, an approximation of the mathematically correct value.

To the tester, close may or may not be problematic. The longer a chain of calculations using floats, the more likely the values will drift from exactness. Likewise, rounding and truncating can cause errors. Consider the following chain of events. Two values are calculated and placed into floating point variables; one comes close to 6, the other close to 2. Very close; for all intents and purposes the values are almost exactly 6 and 2. The second value is divided into the first, resulting in a value close to 3. Perhaps something like 2.999999999999932. Now, suppose the value is placed into an integer rather than a float—the compiler will allow that. Suppose the developer decides that they do not want the fractional value and, rather than rounding, uses the truncation function. Essentially, following along this logic, we get 6 divided by 2 equals 2. Oops. If this sounds far-fetched, it is. But this particular error actually happened to software that we were testing, and we did a lot of head scratching digging it out. Each step the programmer made was sensible, even if the answer was nonsensical. Clearly, boundaries of floating point numbers—and calculations made with them—need to be tested.

4.2.2.6 Testing Floating Point Numbers

Testing of financial operations involves floating point numbers, so all of the issues just discussed apply. Notice that the question of precision has serious

real-world implications. People tend to be very sensitive to rounding errors that involve things like money, investment account balances, and the like. Furthermore, there are regulations and contracts that often apply, along with any specified requirements. So testing financial applications involves a great deal of care when these kinds of issues are involved.

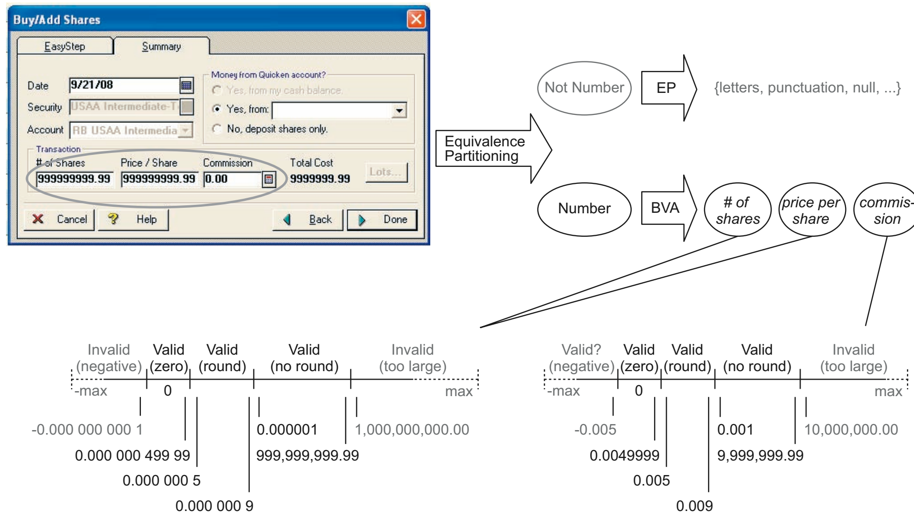


Figure 4–12 *A stock-purchase screen from Quicken*

In figure 4-12, you can see the application of equivalence partitioning and boundary value analysis to the input of a stock purchase transaction in Quicken. In this application, when you enter a stock purchase transaction, after entering the stock, you input the number of shares, the price per share, and the commission, if any. Quicken calculates the amount of the transaction.

First, for each of the three fields—number of shares, price per share, and commission—we can say that any non-numeric input is invalid. So that's common across all three.

For the numeric inputs, the price per share and the number of shares are handled the same. The number must be zero or greater. Look closely at all of the boundaries that are shown in figure 4-12. Some of these boundaries are not necessarily intuitive. When boundary testing with real numbers, there are two complexities we must deal with: precision and round-off.

By precision, in this case we are talking about the number of decimal digits we are willing to test. Many people refer to this as the *epsilon* (or smallest recognizable difference) to which we will test. Suppose we have a change in system behavior exactly at the value 10.00. Values below are valid, above are invalid. What is the closest possible value that you can reasonably say is the invalid boundary? 10.01? Clearly we can get many values between 10.00 and 10.01, including 10.001, 10.0001, 10.00001...

Somewhere in testing we have to decide the boundary beyond which it makes no sense to test. One such line is the epsilon, and it generally represents the number of decimal digits to which we will test. In the preceding example, we might decide that we want to test to an epsilon of 2. That would make our invalid boundary 10.01, even if, in reality, it is not a physical boundary. Testers must understand the epsilon to which they will be testing. This epsilon may be decided by the programmer who wrote the code, the designer who designed it, or even the computer representation used.

Likewise, rounding off is an important facet of real-world boundaries. Again, look at figure 4-12. It is possible to buy a fractional portion of a share of stock. Clearly, if we put 0 (zero) in the number of shares purchased, we will buy no stock. You might think that if we put a positive non-zero value in, we would automatically be purchasing some stock. However, some values will round down to zero, even though the absolute value is greater than zero. And therein lies our boundary. We should find out exactly where the boundary is, that is, where the behavior changes (with due respect to the epsilon).

In this version of Quicken, Rex discovered through trial and error that the smallest amount of a share that can be purchased is 0.000 001 shares. This is a value where behavior changes. However, the testable boundary must allow for rounding. The value 0.000 000 5 will round up to 0.000 001. That makes it the low valid boundary. But we need to go to more decimal places to allow the round-off to occur, down to the epsilon. Hence the low invalid boundary is some value less than that which will round down to zero; in this case we select 0.000 000 499 99.

4.2.2.7 How Many Boundaries?

Let's wind down our discussion of boundary value analysis by mentioning a somewhat obscure point that can nevertheless arise if you do any reading

about testing. This is the question of how many boundary values exist at a boundary.

In the material so far, we've shown just two boundary values per boundary. The boundary lies between the largest member of one equivalence class and the smallest member of the equivalence class above it. In other words, the boundary itself doesn't correspond to any member of any class.

However, some authors, of whom Boris Beizer is probably the most notable, say there are three boundary values. Why would there be three?

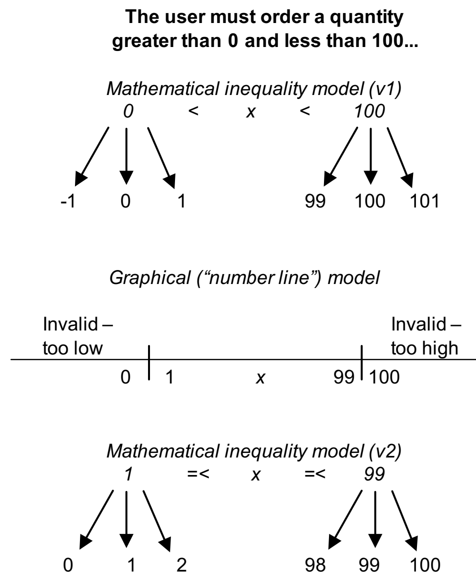


Figure 4–13 *Two ways to look at boundaries*

The difference arises from the use of a mathematical analysis rather than the graphical one that we've used, as shown in figure 4-13. You can see that the two mathematical inequalities shown in figure 4-13 describe the same situation as the graphical model. Applying Beizer's methodology, you select the value itself as the middle boundary value, then you add the smallest possible amount to that value and subtract the smallest possible amount from that value to generate the two other boundary values. Remember, with integers, as in this example, the smallest possible amount is one, while for floating points we have to consider those pesky issues of precision and round-off.

As you can see from this example, the mathematical boundary values will always include the graphical boundary values. To us, the three-value approach is wasted effort. We have enough to do already without creating more test values to cover, especially when they don't address any risks that haven't already been covered by other test values.

4.2.2.8 Boundary Value Exercise

Loan amount	{Enter a loan amount} Whole dollar amounts (no cents). Will be rounded to nearest $ 100.
Property value?	{Enter the property's value} Whole dollar amounts (no cents). Will be rounded to nearest $ 100.

Figure 4–14 *HELLOCARMS system amount screen prototype*

A screen prototype for one screen of the HELLOCARMS system is shown in figure 4-14. This screen asks for two pieces of information:

- Loan amount
- Property value

For both fields, the system allows entry of whole dollar amounts only (no cents), and it rounds to the nearest $100.

Assume the following rules apply to loans:

- The minimum loan amount is $5,000.
- The maximum loan amount is $1,000,000.
- The minimum property value is $25,000.
- The maximum property value is $5,000,000.

Refer also to requirements specification elements 010-010-130 and 010-010-140 in the Functional System Requirements section of the HELLOCARMS system requirements document.

If the fields are valid, then the screen will be accepted. Either the Telephone Banker will continue with the application or the application will be transferred to a Senior Telephone Banker for further handling. If one or both fields are invalid, an error message is displayed.

The exercise consists of two parts:

1. Show the equivalence partitions and boundary values for each of the two fields, indicating valid and invalid members and the boundaries for those partitions.
2. Create test cases to cover these partitions and boundary values, keeping in mind the rules about combinations of valid and invalid members.

The answers to the two parts are shown on the following pages. You should review the answer to the first part (and, if necessary, revise your answer to the second part) before reviewing the answer to the second part.

4.2.2.9 Boundary Value Exercise Debrief

First, let's take a look at the equivalence partitions and boundary values, which are shown in figure 4-15.

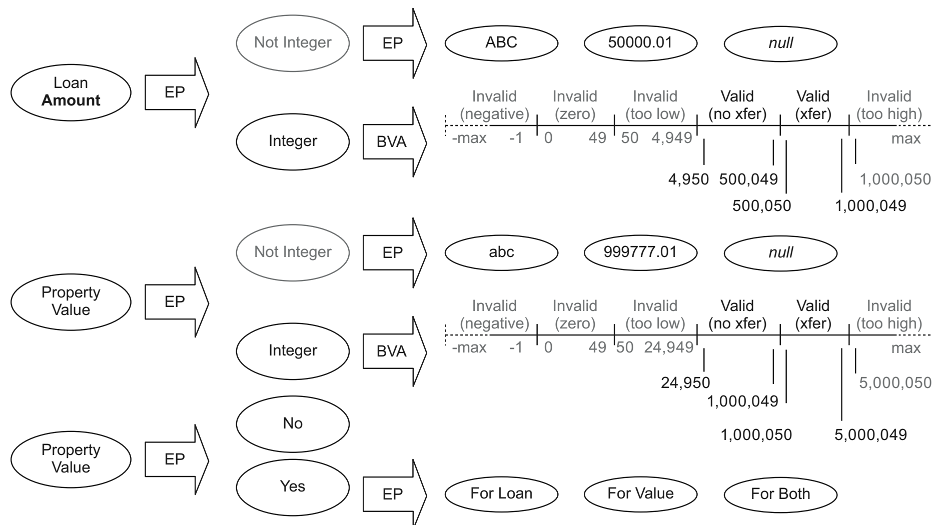


Figure 4–15 Equivalence partitions (EP) and boundary values (BVA) for exercise

So, for the loan amount we can show the boundary values and equivalence partitions as shown in table 4-8.

Table 4–8

#	Partition	Boundary Value
1	Letter: ABC	-
2	Decimal: 50,000.01	-
3	Null	-
4	Invalid (negative)	*-max*
5	Invalid (negative)	-1
6	Invalid (zero)	0
7	Invalid (zero)	49
8	Invalid (too low)	50
9	Invalid (too low)	4,949
10	Valid (no transfer)	4,950
11	Valid (no transfer)	500,049
12	Valid (transfer)	500,050
13	Valid (transfer)	1,000,049
14	Invalid (too high)	1,000,050
15	Invalid (too high)	max

For the property value, we can show the boundary values and equivalence partitions as shown in table 4-9.

Table 4–9

#	Partition	Boundary Value
1	Letter: abc	-
2	Decimal: 999,777.01	-
3	Null	-
4	Invalid (negative)	*-max*
5	Invalid (negative)	-1
6	Invalid (zero)	0
7	Invalid (zero)	49
8	Invalid (too low)	50
9	Invalid (too low)	24,949
10	Valid (no transfer)	24,950
11	Valid (no transfer)	1,000,049
12	Valid (transfer)	1,000,050
13	Valid (transfer)	5,000,049
14	Invalid (too high)	5,000,050
15	Invalid (too high)	max

Make sure you understand why these values are boundaries based on round-off rules given in the requirements. For the transfer decision, we can show the equivalence partitions for the loan amount as shown in table 4-10.

Table 4–10

#	Partition
1	No
2	Yes, for the loan amount
3	Yes, for the property value
4	Yes, for both

Now, let's create tests from these equivalence partitions and boundary values. We'll capture traceability information from the test case number back to the partitions or boundary values, and as before, once we have a trace from each partition to a test case, we're done—as long as we didn't combine invalid values!

Table 4–11

Inputs	*1*	*2*	*3*	*4*	*5*	*6*
Loan amount	4,950	500,050	500,049	1,000,049	ABC	50,000.01
Property value	24,950	1,000,049	1,000,050	5,000,049	100,000	200,000
Outputs						
Accept?	Y	Y	Y	Y	N	N
Transfer?	N	Y (loan)	Y (prop)	Y (both)	-	-

Inputs	*7*	*8*	*9*	*10*	*11*	*12*
Loan amount	*null*	100,000	200,000	300,000	*-max*	-1
Property value	300,000	abc	999,777.01	*null*	400,000	500,000
Outputs						
Accept?	N	N	N	N	N	N
Transfer?	-	-	-	-	-	-

Inputs	*13*	*14*	*15*	*16*	*17*	*18*
Loan amount	0	49	50	4,949	1,000,050	*max*
Property value	600,000	700,000	800,000	900,000	1,000,000	1,100,000
Outputs						
Accept?	N	N	N	N	N	N
Transfer?	-	-	-	-	-	-

Inputs	*19*	*20*	*21*	*22*	*23*	*24*
Loan amount	400,000	500,000	600,000	700,000	800,000	900,000
Property value	*-max*	-1	0	49	50	24,949
Outputs						
Accept?	N	N	N	N	N	N
Transfer?	-	-	-	-	-	-

Inputs	*25*	*26*	*27*	*28*	*29*	*30*
Loan amount	1,000,000	555,555				
Property value	5,000,050	*max*				
Outputs						
Accept?	N	N				
Transfer?	-	-				

Table 4–12

#	*Partition*	*Boundary Value*	*Test Case*
1	Letter: ABC	-	5
2	Decimal: 50,000.01	-	6
3	Null	-	7
4	Invalid (negative)	*-max*	11
5	Invalid (negative)	-1	12
6	Invalid (zero)	0	13
7	Invalid (zero)	49	14
8	Invalid (too low)	50	15
9	Invalid (too low)	4,949	16
10	Valid (no transfer)	4,950	1
11	Valid (no transfer)	500,049	3
12	Valid (transfer)	500,050	2
13	Valid (transfer)	1,000,049	4
14	Invalid (too high)	1,000,050	17
15	Invalid (too high)	max	18

Table 4–13

#	Partition	Boundary Value	Test Case
1	Letter: abc	-	8
2	Decimal: 999,777.01	-	9
3	Null	-	10
4	Invalid (negative)	*-max*	19
5	Invalid (negative)	-1	20
6	Invalid (zero)	0	21
7	Invalid (zero)	49	22
8	Invalid (too low)	50	23
9	Invalid (too low)	24,949	24
10	Valid (no transfer)	24,950	1
11	Valid (no transfer)	1,000,049	2
12	Valid (transfer)	1,000,050	3
13	Valid (transfer)	5,000,049	4
14	Invalid (too high)	5,000,050	25
15	Invalid (too high)	max	26

Table 4–14

#	Partition	Test Case
1	No	1
2	Yes, for the loan amount	2
3	Yes, for the property value	3
4	Yes, for both	4

Notice that's there's another interesting combination related to the transfer decision that we covered in our tests. This was when the values were rejected as inputs, in which case we should not even be able to leave the screen, not to mention transfer the application. We did test with both loan amounts and property values that would have triggered a transfer had the other value been valid. We could have shown that as a third set of equivalence classes for the transfer decision.

ISTQB Glossary

decision table: A table showing combinations of inputs and/or stimuli (causes) with their associated outputs and/or actions (effects), which can be used to design test cases.

decision table testing: A black-box test design technique in which test cases are designed to execute the combinations of inputs and/or stimuli (causes) shown in a decision table.

4.2.3 Decision Tables

Equivalence partitioning and boundary value analysis are very useful techniques. They are especially useful, as you saw in the earlier parts of this section, when testing input field validation at the user interface. However, lots of testing that we do as technical test analysts involves testing the business logic that sits underneath the user interface. We can use boundary values and equivalence partitioning on business logic too, but two additional techniques, decision tables and state-based testing, will often prove handier and more powerful.

Let's start with decision tables.

Conceptually, decision tables express the rules that govern handling of transactional situations. By their simple, concise structure, they make it easy for us to design tests for those rules, usually at least one test per rule.

When we said "transactional situations," what we meant was those situations where the conditions—inputs, preconditions, etc.—that exist at a given moment in time for a single transaction are sufficient by themselves to determine the actions the system should take. If the conditions on their own are not sufficient, but we must also refer to what conditions have existed in the past, then we'll want to use state-based testing, which we'll cover later in this chapter.

The underlying model is a table. The model connects combinations of conditions with the action or actions that should occur when each particular combination of conditions arises.

To create test cases from a decision table, we are going to design test inputs that fulfill the conditions given. The test outputs will correspond to the action or actions given for that combination of conditions. During test execution, we check that the actual actions taken correspond to the expected actions.

We create enough test cases that every combination of conditions is covered by at least one test case. Oftentimes, that coverage criterion is relaxed to ensure that we cover those combinations of conditions that can determine the action or actions. If that's a little confusing—which it might be, depending on how you prepared for the Foundation exam, because this isn't always explained properly in books and classes—the distinction we're drawing will become clear to you when we talk about collapsed decision tables.

With a decision table, the coverage criterion boils down to an easy-to-remember rule of at least one test per column in the table.

So, what is our bug hypothesis with decision tables? What kind of bugs are we looking for? There are several. First, under some combination of conditions, the wrong action might occur. In other words, there may be some action that the system is not to take under this combination of conditions, yet it does. Second, under some combination of conditions, the system might not take the right action. In other words, there is some action that the system is to take under this combination of conditions, yet it does not.

Beyond those two possibilities, there is also a very real positive value of using decision tables during the analysis portion of our testing. A decision table forces testers to consider many possibilities that we may not have considered otherwise. By considering all of the permutations of conditions possible, even if we don't test them all, we may discover some unknown unknowns. That is, we are likely to find some scenarios that the analysts and developers had not considered. By considering these scenarios, we can not only capture incipient defects but avoid later disagreements about correct behavior.

Table 4–15

Conditions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Real account?	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Active account?	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
Within limit?	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Location okay?	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Actions																
Approve	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Call cardholder?	N	Y	Y	Y	N	Y	Y	Y	N	N	N	N	N	N	N	N
Call vendor?	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Consider the decision table shown in table 4-15. This table shows the decisions being made at an e-commerce website about whether to accept a credit card purchase of merchandise. Once the information goes to the credit card processing company for validation, how can we test their decisions? We could handle that with equivalence partitioning, but there is actually a whole set of conditions that determine this processing:

- Does the named person hold the credit card entered, and is the other information correct?
- Is it still active or has it been cancelled?
- Is the person within or over their limit?
- Is the transaction coming from a normal or a suspicious location?

The decision table in table 4-15 shows how these four conditions interact to determine which of the following three actions will occur:

- Should we approve the transaction?
- Should we call the cardholder (e.g., to warn them about a purchase from a strange place)?
- Should we call the vendor (e.g., to ask them to seize the cancelled card)?

Take a minute to study the table to see how this works. The conditions are listed at the top left of the table, and the actions at the bottom left. Each column to the right of this leftmost column contains a business rule. Each rule says, in essence, "Under this particular combination of conditions (shown at the top of the rule), carry out this particular combination of actions (shown at the bottom of the rule)."

Notice that the number of columns—i.e., the number of business rules—is equal to 2 (two) raised to the power of the number of conditions. In other words, 2 raised to the 4th power (based on four conditions), which results in 16 columns. When the conditions are strictly Boolean—true or false—and we're dealing with a full decision table (not a collapsed one), that will always be the case. If one or more of the conditions are not rendered as Boolean, then the number of columns will not follow this rule.

Did you notice how we populated the conditions? The topmost condition changes most slowly. Half of the columns are Yes, then half No. The condition under the topmost changes more quickly, but it changes more slowly than all the

other conditions below it. The pattern is a quarter Yes, then a quarter No, then a quarter Yes, then a quarter No. Finally, for the bottommost condition, the alternation is Yes, No, Yes, No, Yes, etc. This pattern makes it easy to ensure that you don't miss anything. If you start with the topmost condition, set the left half of the rule columns to Yes and the right half of the rule columns to No, then following the pattern we showed, if you get to the bottom and the Yes, No, Yes, No, Yes, etc. pattern doesn't hold, you did something wrong. The other algorithm that you can use, again assuming that all conditions are Boolean, is to count the columns in binary. The first column is YYYY, second YYYN, third YYNY, fourth YYNN, etc. If you are not familiar with binary arithmetic, forget we mentioned it.

Deriving test cases from this example is relatively easy: each column (business rule) of the table generates a test case. When time comes to run the tests, we'll create the conditions that are each test's inputs. We'll replace the yes/no conditions with actual input values for credit card number, security code, expiration date, and cardholder name, either during test design or perhaps even at test execution time. We'll verify the actions that are the test's expected results.

In some cases, we might generate more than one test case per column. We'll cover this possibility later, as we enlist our previous test techniques, equivalence partitioning and boundary value analysis, to extend decision table testing.

Notice that, in this case, some of the test cases don't make much sense. For example, how can the account not be real but yet active? How can the account not be real but within the given limit?

This kind of situation is a hint that maybe we don't need all the columns in our decision table.

4.2.3.1 Collapsing Columns in the Table

We can sometimes collapse the decision table, combining columns, to achieve a more concise—and in some cases more sensible—decision table. In any situation where the value of one or more particular conditions can't affect the actions for two or more combinations of conditions, we can collapse the decision table.

This involves combining two or more columns where, as we said, one or more of the conditions don't affect the actions. As a hint, combinable columns are often *but not always* next to each other. You can at least start by looking at columns next to each other.

To combine two or more columns, look for two or more columns that result in the same combination of actions. Note that the actions in the two columns must be the same for all of the actions in the table, not just some of them. In these columns, some of the conditions will be the same, and some will be different. The ones that are different obviously don't affect the outcome. We can replace the conditions that are different in those columns with a dash. The dash usually means we don't care, it doesn't matter, or it can't happen, given the other conditions.

Now, repeat this process until the only columns that share the same combination of actions for all the actions in the table are ones where you'd be combining a dash with a Yes or No value and thus wiping out an important distinction for cause of action. What we mean by this will be clear in the example on the next page, if it's not clear already.

Another word of caution at this point: Be careful when dealing with a table where more than one rule can apply at one single point in time. These tables have nonexclusive rules. We'll discuss that further later in this section.

Table 4–16

Conditions	1	2	3	5	6	7	9
Real account?	Y	Y	Y	Y	Y	Y	N
Active account?	Y	Y	Y	N	N	N	-
Within limit?	Y	Y	N	Y	Y	N	-
Location okay?	Y	N	-	Y	N	-	-
Actions							
Approve	Y	N	N	N	N	N	N
Call card holder?	N	Y	Y	N	Y	Y	N
Call vendor?	N	N	N	Y	Y	Y	Y

Table 4-16 shows the same decision table as before, but collapsed to eliminate extraneous columns. Most notably, you can see that what were columns 9 through 16 in the original decision table collapsed into a single column. Notice that all of the columns 9 through 16 are essentially the same test: Is this for a real account? If it is not real (note that for all columns, 9 through 16, the answer is no), then it certainly cannot be active, within limit, or have an okay location. Since those conditions are impossible, we would essentially be running the same test eight times.

We've kept the original column numbers for ease of comparison. Again, take a minute to study the table to see how we did this. Look carefully at columns 1, 2, and 3. Notice that we can't collapse 2 and 3 because that would result in "dash" for both "within limit" and "location okay." If you study this table or the full one, you can see that one of these conditions must *not* be true for the cardholder to receive a call. The collapse of rule 4 into rule 3 says that, if the card is over limit, the cardholder will be called, regardless of location.

The same logic applies to the collapse of rule 8 into rule 7.

Notice that the format is unchanged. The conditions are listed at the top left of the table, and the actions at the bottom left. Each column to the right of this leftmost column contains a business rule. Each rule says, "Under this particular combination of conditions (shown at the top of the rule, some of which might not be applicable), carry out this particular combination of actions (shown at the bottom of the rule, all of which are fully specified)."

Notice that the number of columns is no longer equal to 2 raised to the power of the number of conditions. This makes sense, since otherwise no collapsing would have occurred. If you are concerned that you might miss something important, you can always start with the full decision table. In a full table, because of the way you generate it, it is guaranteed to have all the combinations of conditions. You can mathematically check if it does. Then, carefully collapse the table to reduce the number of test cases you create.

Also, notice that, when you collapse the table, that pleasant pattern of Yes and No columns present in the full table goes away. This is yet another reason to be very careful when collapsing the columns, because you can't count on the pattern or the mathematical formula to check your work.

4.2.3.2 Combining Decision Table Testing with Other Techniques

Okay, let's address an issue we brought up earlier, the possibility of multiple test cases per column in the decision table via the combination of equivalence partitioning and the decision table technique. In figure 4-16, we refer to our example decision table of table 4-16, specifically column 9.

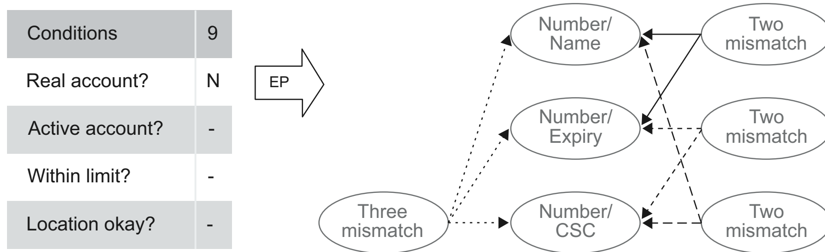


Figure 4–16 *Equivalence partitions and decision tables*

We can apply equivalence partitioning to the question, How many interesting—from a test point of view—ways are there to have an account not be real? As you can see from figure 4-16, this could happen seven potentially interesting ways:

- Card number and cardholder mismatch
- Card number and expiry mismatch
- Card number and CSC (card security code) mismatch
- Two of the above mismatches (three possibilities)
- All three mismatches

So, there could be seven tests for that column.

How about boundary value analysis? Yes, that too can be applied to decision tables to find new and interesting tests. For example, How many interesting test values relate to the credit limit?

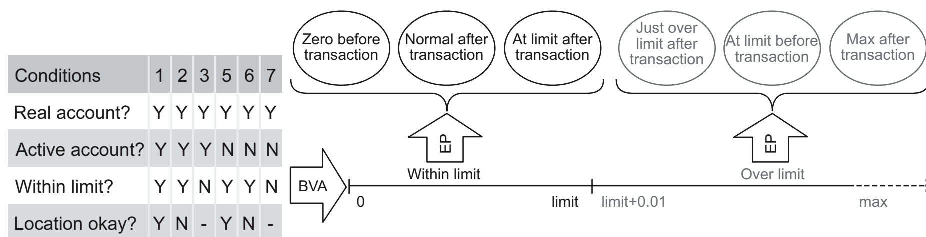


Figure 4–17 *Boundary values and decision tables*

As you can see from figure 4-17, equivalence partitioning and boundary value analysis show us six interesting possibilities:

1. The account starts at zero balance.
2. The account would be at a normal balance after transaction.

3. The account would be exactly at the limit after the transaction.
4. The account would be over the limit after the transaction.
5. The account was at exactly the limit before the transaction (which would ensure going over if the transaction concluded).
6. The account would be at the maximum overdraft value after the transaction (which might not be possible).

Combining this with the decision table, we can see that would again end up with more "over limit" tests than we have columns—one more, to be exact—so we'd increase the number of tests just slightly. In other words, there would be four within-limit tests and three over-limit tests. That's true unless you wanted to make sure that each within-limit equivalence class was represented in an approved transaction, in which case column 1 would go from one test to three.

4.2.3.3 Nonexclusive Rules in Decision Tables

Let's finish our discussion about decision tables by looking at the issue of non-exclusive rules we mentioned earlier.

Table 4–17

Conditions	1	2	3
Foreign exchange?	Y	-	-
Balance forward?	-	Y	-
Late payment?	-	-	Y
Actions			
Exchange fee?	Y	-	-
Charge interest?	-	Y	-
Charge late fee?	-	-	Y

Sometimes more than one rule can apply to a transaction. In table 4-17, you see a table that shows the calculation of credit card fees. There are three conditions, and notice that zero, one, two, or all three of those conditions could be met in a given month. How does this situation affect testing?

It complicates the testing a bit, but we can use a methodical approach and risk-based testing to avoid the major pitfalls.

To start with, test the decision table like a normal one, one rule at a time, making sure that no conditions not related to the rule you are testing are met. This allows you to test rules in isolation—just as you are forced to do in situations where the rules are exclusive.

Next, consider testing combinations of rules. Notice we said, "Consider," not "test all possible combinations of rules." You'll want to avoid combinatorial explosions, which is what happens when testers start to test combinations of factors without consideration of the value of those tests. Now, in this case, there are only 8 possible combinations—three factors, two options for each factor, 2 times 2 times 2 is 8. However, if you have six factors with five options each, you would now have 15,625 combinations.

One way to avoid combinatorial explosions is to identify the possible combinations and then use risk to weight those combinations. Try to get to the important combinations and don't worry about the rest.

Another way to avoid combinatorial explosions is to use techniques like classification trees and pairwise testing, which are covered in Rex's book *Advanced Software Testing Vol. 1*.

4.2.3.4 Decision Table Exercise

During development, the HELLOCARMS project team added a feature to HELLOCARMS. This feature allows the system to sell a life insurance policy to cover the amount of a home equity loan so that, should the borrower die, the policy will pay off the loan. The premium is calculated annually, at the beginning of each annual policy period and based on the loan balance at that time. The base annual premium will be $1 for $10,000 in loan balance. The insurance policy is not available for lines of credit or for reverse mortgages.

The system will increase the base premium by a certain percentage based on some basic physical and health questions that the Telephone Banker will ask during the interview.

A Yes answer to any of the following questions will trigger a 50 percent increase to the base premium:

1. Have you smoked cigarettes in the past 12 months?
2. Have you ever been diagnosed with cancer, diabetes, high cholesterol, high blood pressure, a heart disorder, or stroke?
3. Within the last 5 years, have you been hospitalized for more than 72 hours except for childbirth or broken bones?
4. Within the last 5 years, have you been completely disabled from work for a week or longer due to a single illness or injury?

The Telephone Banker will also ask about age, weight, and height. (Applicants cannot be under 18.) The weight and height are combined to calculate the body mass index (BMI). Based on that information, the Telephone Banker will apply the rules in table 4-18 to decide whether to increase the rate or even decline to issue the policy based on possible weight-related illnesses in the person's future.

Table 4–18

	Body Mass Index (BMI)			
Age	<17	34-36	37-39	>39
18-39	Decline	75%	100%	Decline
40-59	Decline	50%	75%	Decline
>59	Decline	25%	50%	Decline

The increases are cumulative. For example, if the person has normal weight, smokes cigarettes, and has high blood pressure, the annual rate is increased from $1 per $10,000 to $2.25[5] per $10,000. If the person is a 45-year-old male diabetic with a body mass index of 39, the annual rate is increased from $1 per $10,000 to $2.625[6] per $10,000.

The exercise consists of three steps:

1. Create a decision table that shows the effect of the four health questions and the body mass index.
2. Show the boundary values for body mass index and age.
3. Create test cases to cover the decision table and the boundary values, keeping in mind the rules about testing nonexclusive rules.

The answers to the three parts are shown on the next pages. You should review the answer to the each part (and, if necessary, revise your answer to the next parts) before reviewing the answer to the next part.

4.2.3.5 Decision Table Exercise Debrief

First, we created the decision table from the four health questions and the BMI/age table. The answer is shown in table 4-19. Note that the increases are shown in percentages.

5. Two risk factors; the calculation is as follows: ($1 x 50%) = $1.50 + ($1.50 x 50%) = $2.25
6. One risk factor plus BMI increase of 75%: ($1 x 50%) = $1.50 + ($1.50 x 75%) = $2.625

Table 4–19

Conditions	1	2	3	4	5	6	7	8	9	10	11	12
Smoked?	Y	–	–	–	–	–	–	–	–	–	–	–
Diagnosed?	–	Y	–	–	–	–	–	–	–	–	–	–
Hospitalized?	–	–	Y	–	–	–	–	–	–	–	–	–
Disabled?	–	–	–	Y	–	–	–	–	–	–	–	–
BMI	–	–	–	–	34–36	34–36	34–36	37–39	37–39	37–39	<17	>39
Age	–	–	–	–	18–39	40–59	>59	18–39	40–59	>59	–	–
Actions												
Increase	50	50	50	50	75	50	25	100	75	50	–	–
Decline	–	–	–	–	–	–	–	–	–	–	Y	Y

It's important to notice that rules 1 through 4 are nonexclusive, though rules 5 through 12 are exclusive.

In addition, there is an implicit rule that the age must be greater than 17 or the applicant will be denied not only insurance but the loan itself. We could have put that here in the decision table, but our focus is primarily on testing business functionality, not input validation. We'll cover those tests with boundary values.

Now, let's look at the boundary values for body mass index and age, shown in figure 4-18.

Figure 4–18 *BMI and age boundary values*

Three important testing notes relate to the body mass index. First, the body mass index is not entered directly but rather by entering height and weight. Depending on the range and precision of these two fields, there could be dozens of ways to enter a given body mass index. Second, the maximum body mass index is achieved by entering the smallest possible height and the largest possible weight. Third, we'd need to separately understand the boundary values for these two fields and make sure those were tested properly.

An important testing note relates to the age. You can see that we omitted equivalence classes related to invalid ages, such as negative ages and non-integer input for ages. Again, our idea is that we'd need to separately test the input field validation. Here, our focus is on testing business logic.

Finally, for both fields, we omit any attempt to figure out the maxima. Either someone will give us a requirements specification that tells us that during test design or we'll try to ascertain it empirically during test execution.

So, for the BMI, we can show the boundary values and equivalence partitions as shown in table 4-20.

Table 4–20

#	Partition	Boundary Value
1	Too thin	0
2	Too thin	16
3	No increase	17
4	No increase	33
5	Smaller increase	34
6	Smaller increase	36
7	Bigger increase	37
8	Bigger increase	39
9	Too heavy	40
10	Too heavy	max

For the age, we can show the boundary values and equivalence partitions as shown in table 4-21.

Table 4–21

#	Partition	Boundary Value
1	Too young	0
2	Too young	17
3	Young	18
4	Young	39
5	Mid-aged	40
6	Mid-aged	59
7	Senior	60
8	Senior	max

Finally, table 4-22 shows the test cases. They are much like the decision table, but note that we have shown the rate (in dollars per $10,000 of loan balance) rather than the percentage increase.

Table 4–22

	Test case											
Conditions	1	2	3	4	5	6	7	8	9	10	11	12
Smoked?	Y	N	N	N	N	N	N	N	N	N	N	N
Diagnosed?	N	Y	N	N	N	N	N	N	N	N	N	N
Hospitalized?	N	N	Y	N	N	N	N	N	N	N	N	N
Disabled?	N	N	N	Y	N	N	N	N	N	N	N	N
BMI	N	N	N	N	34	36	35	34	36	35	37	39
Age	N	N	N	N	18	39	40	59	60	max	20	30
Actions												
Rate	1.5	1.5	1.5	1.5	1.75	1.75	1.5	1.5	1.25	1.25	2	2
Decline	N	N	N	N	N	N	N	N	N	N	N	N

	Test case											
Conditions	13	14	15	16	17	18	19	20	21	22	23	24
Smoked?	N	N	N	N	N	N	N	N	N	N	N	N
Diagnosed?	N	N	N	N	N	N	N	N	N	N	N	N
Hospitalized?	N	N	N	N	N	N	N	N	N	N	N	N
Disabled?	N	N	N	N	N	N	N	N	N	N	N	N
BMI	38	37	39	38	16	40	0	max	17	33	20	30
Age	45	55	65	75	35	50	25	70	37	47	0	17
Actions												
Rate	1.75	1.75	1.50	1.50	N/A	N/A	N/A	N/A	1	1	N/A	N/A
Decline	N	N	N	N	Y	Y	Y	Y	N	N	Y	Y

	Test case				
Conditions	25	26	27	28	29
Smoked?	Y	N	N	N	Y
Diagnosed?	N	Y	N	N	Y
Hospitalized?	N	N	Y	N	Y
Disabled?	N	N	N	Y	Y
BMI	35	36	34	38	37
Age	20	50	70	30	35
Actions					
Rate	2.625	2.25	1.875	3	10.125
Decline	N	N	N	N	N

Notice our approach to testing the nonexclusive rules. First, we tested every rule, exclusive and nonexclusive, in isolation. Then, we tested the remaining untested boundary values. Next, we tested combinations of only one nonexclusive rule with one exclusive rule, making sure each nonexclusive rule had been tested once in combination (but not all the exclusive rules were tested in combination). Finally, we tested a combination of all four nonexclusive rules with one exclusive rule. We did not use combinations with the "decline" rules since presumably there's no way to check if the increase was correctly calculated.

You might also have noticed that we managed to sneak in covering the minimum and maximum increases. However, we probably didn't cover every possible increase. Since we didn't test every possible pair, triple, and quadruple combination of rules, we certainly didn't test every way an increase could be calculated by that table. That topic is covered in *Advanced Software Testing Vol.1*.

For the decision table and the boundary values, we've captured test coverage in the following tables to make sure we missed nothing. Table 4-23, table 4-24, and table 4-25 show decision table coverage using three coverage metrics.

Table 4–23

Conditions	1	2	3	4	5	6	7	8	9	10	11	12
Smoked?	Y	-	-	-	-	-	-	-	-	-	-	-
Diagnosed?	-	Y	-	-	-	-	-	-	-	-	-	-
Hospitalized?	-	-	Y	-	-	-	-	-	-	-	-	-
Disabled?	-	-	-	Y	-	-	-	-	-	-	-	-
BMI	-	-	-	-	34-36	34-36	34-36	37-39	37-39	37-39	<17	>39
Age	-	-	-	-	18-39	40-59	>59	18-39	40-59	>59	-	-
Actions												
Increase	50	50	50	50	75	50	25	100	75	50	-	-
Decline	-	-	-	-	-	-	-	-	-	-	Y	Y
Single Rule Coverage												
Test case(s)	1	2	3	4	5, 6	7, 8	9, 10	11, 12	13, 14	15, 16	17,19	18,20
Pairs of Rules Coverage												
Test case(s)	25	26	27	28	25	26	27	28				
Maximum Combination of Rules Coverage												
Test case(s)	29	29	29	29				29				

ISTQB Glossary

state diagram: A diagram that depicts the states that a component or system can assume and shows the events or circumstances that cause and/or result from a change from one state to another.

state transition: A transition between two states of a component or system.

state transition testing: A black-box test design technique in which test cases are designed to execute valid and invalid state transitions.

Table 4–24

#	Partition	Boundary Value	Test Case
1	Too thin	0	19
2	Too thin	16	17
3	No increase	17	21
4	No increase	33	22
5	Smaller increase	34	5
6	Smaller increase	36	6
7	Bigger increase	37	11
8	Bigger increase	39	12
9	Too heavy	40	18
10	Too heavy	*max*	20

Table 4–25

#	Partition	Boundary Value	Test Case
1	Too young	0	23
2	Too young	17	24
3	Young	18	5
4	Young	39	6
5	Mid-aged	40	7
6	Mid-aged	59	8
7	Senior	60	9
8	Senior	*max*	10

4.2.4 State-Based Testing and State Transition Diagrams

We said that after our discussion of equivalence partitioning and boundary value analysis, we would cover two techniques that would prove useful for testing business logic, even more useful than equivalence partitioning and bound-

ary value analysis. We covered decision tables, which work very well in transactional situations, in the last section.

Now we move on to state-based testing. State-based testing is ideal when we have sequences of events that occur and conditions that apply to those events and the proper handling of a particular event/condition depends on the events and conditions that have occurred in the past. In some cases, the sequences of events can be potentially infinite, which of course exceeds our testing capabilities, but we want to have a test design technique that allows us to handle arbitrarily long sequences of events.

The underlying model is a state transition diagram or table. The diagram or table connects beginning states, events, and conditions with resulting states and actions.

To describe this interaction, consider a system. At a given time, some status quo prevails and the system is in a steady state. Then some event occurs, some event that the system must handle. The handling of that event might be influenced by one or more conditions. The event/condition combination triggers a state transition, either from the current state to a new state or from the current state back to itself again. In the course of the transition, the system takes one or more actions.

Given this model, we generate tests that traverse the states and transitions. The inputs trigger events and create conditions, while the expected results of the test are the new states and actions taken by the system.

Differing coverage criteria apply for state-based testing. The weakest criterion requires that the tests visit every state and traverse every transition. This criterion can be applied to state transition diagrams. A higher coverage criterion is at least one test for every row in a state transition table. Achieving "every row" coverage will achieve "every state and transition" coverage, which is why we said it was a higher coverage criterion.

Another potentially higher coverage criterion requires that at least one test cover each transition sequence of *N* or less length. The *N* can be 1, 2, 3, 4, or higher. This is called alternatively *Chow's switch coverage*—after Professor Chow, who developed it—or *N-1 switch coverage*, after the level given to the degree of coverage. If you cover all transitions of length one, then *N*-1 switch coverage means 0 switch coverage. Notice that this is the same as the lowest level of coverage discussed. If you cover all transitions of length one and two,

then N-1 switch coverage means 1 switch coverage. This is a higher level of coverage than the lowest level, of course.

Now, 1 switch coverage is not necessarily a higher level of coverage than "every-row" coverage. This is because the state transition table forces testing of state and event/condition combinations that do not occur in the state-transition diagram. The so-called "switches" in *N*-1 switch coverage are derived from the state transition diagram, not the state transition table.

All this might be a bit confusing if you're fuzzy on the test design material covered at the Foundation level. Don't worry, though; it will be clear to you shortly.

So, what is the bug hypothesis in state-based testing? We're looking for situations where the wrong action or the wrong new state occurs in response to a particular event under a given set of conditions based on the history of event/condition combinations so far.

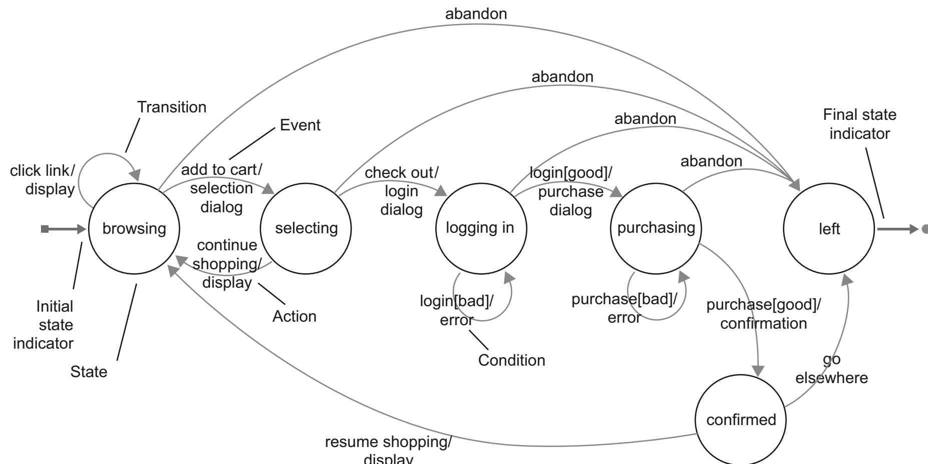


Figure 4–19 *State transition diagram example*

Figure 4-19 shows the state transition diagram for shopping and buying items online from an e-commerce application. It shows the interaction of the system with a customer, from the customer's point of view. Let's walk through it, and we'll point out the key elements of state transition diagrams in general and the features of this one in particular.

First, notice that we have at the leftmost side a small dot-and-arrow element labeled "initial state indicator." This notation shows that, from the customer's point of view, the transaction starts when she starts browsing the website. We can click on links and browse the catalog of items, remaining in a browsing state. Notice the looping arrow above the browsing state. The nodes or bubbles represent states, as shown by the label below the browsing state. The arrows represent transitions, as shown by the label above the looping arrow.

Next, we see that the customer can enter a "selecting" state by adding an item to the shopping cart; "add to cart" is the event, as shown by the label above. The system will display a "selection dialog" where it asks the customer to tell us how many of the item she wants, along with any other information we need to add the item to the cart. Once that's done, the customer can tell the system she wants to continue shopping, in which case the system displays the home screen again and the customer is back in a browsing state. From a notation point of view, notice that the actions taken by the system are shown under the event and after the slash symbol, on the transition arrow, as shown by the label below.

Alternatively, the customer can choose to check out. At this point, she enters a logging-in state. She enters login information. A condition applies to that login information: either it was good or it was bad. If it was bad, the system displays an error and the customer remains in the logging-in state. If it was good, the system displays the first screen in the purchasing dialog. Notice that the "bad" and "good" shown in brackets are, notationally, conditions.

While in the purchasing state, the system will display screens and the customer will enter payment information. Either that information is good or bad—conditions again—which determines whether we can complete and confirm the transaction. Once the transaction is confirmed, the customer can either resume shopping or go somewhere else.

Notice also that the user can always abandon the transaction and go elsewhere.

When we talk about state-based testing during live courses we teach, people often ask, "How do we distinguish a state, an event, or an action?" The main distinctions are as follows:

- A state persists until something happens—something external to the thing itself, usually—to trigger a transition. A state can persist for an indefinite period.
- An event occurs, either instantly or in a limited, finite period. It is the something that happened—the external occurrence—that triggers the transition. Events can be triggered in a variety of ways, such as, for example, by a user with a keyboard or mouse, an external device, or even the operating system.
- An action is the response the system has during the transition. An action, like an event, is either instantaneous or requires a limited, finite period. Often, an action can be thought of as a side effect of an event.

That said, it is sometimes possible to draw the same situation differently, especially when a single state or action can be split into a sequence of finer-grained states, events, and actions. We'll see an example of that in a moment, splitting the purchase state into substates.

Finally, notice that, at the outset, we said this chart is shown from the customer's point of view. Notice that, if we drew this from the system's point of view, it would look different. Maintaining a consistent point of view is critical when drawing these charts, otherwise you'll end up with a nonsensical diagram.

State-based testing uses a formal model, so we can have a formal procedure for deriving tests from the model. Following is a procedure that will work to derive tests that achieve state/transition coverage (i.e., 0 switch coverage):

1. Adopt a rule for where a test procedure or test step must start and where it may or must end. An example is to say that a test step must start in an initial state and may only end in a final state. The reason for the "may" or "must" wording on the ending part is because, in situations where the initial and final states are the same, you might want to allow sequences of states and transitions that pass through the initial state more than once.
2. From an allowed test starting state, define a sequence of event/condition combinations that leads to an allowed test ending state. For each transition that will occur, capture the expected action that the system should take. This is the expected result.

3. As you visit each state and traverse each transition, mark it as covered. The easiest way to do this is to print the state transition diagram and then use a marker to highlight each node and arrow as you cover it.
4. Repeat steps 2 and 3 until all states have been visited and all transitions traversed. In other words, every node and arrow has been marked with the marker.

This procedure will generate logical test cases. To create concrete test cases, you'd have to generate the actual input values and the actual output values. For this book, we intend to generate logical tests to illustrate the techniques, but remember, as we mentioned before, at some point before execution, the implementation of concrete test cases must occur.

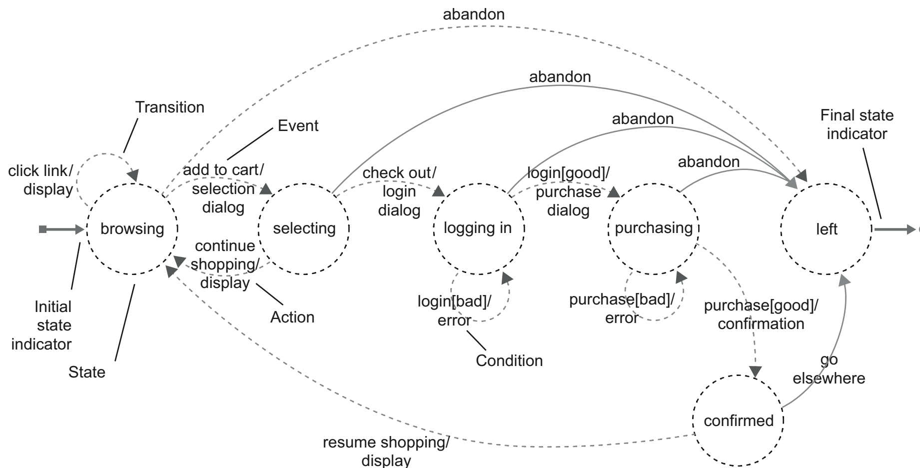


Figure 4–20 *Coverage check 1*

Let's apply this process to the example e-commerce application we've just looked at. In figure 4-20, we will redraw the state transition diagram of figure 4-19 using dashed lines to indicate states and transitions that have been covered. Here, we see two things.

First, we have the rule that says that a test must start in the initial state and must end in the final state.

Next, we generate the first logical test case.

1. (browsing, click link, display, add to cart, selection dialog, continue shopping, display, add to cart, selection dialog, checkout, login dialog, login[bad], error, login[good], purchase dialog, purchase[bad], error, purchase[good], confirmation, resume shopping, display, abandon, left).

At this point, we check completeness of coverage, which we've been tracking in our state transition diagram. As you can see in figure 4-20, we have covered all of the states and most transitions, but not all of the transitions. We need to create some more test cases.

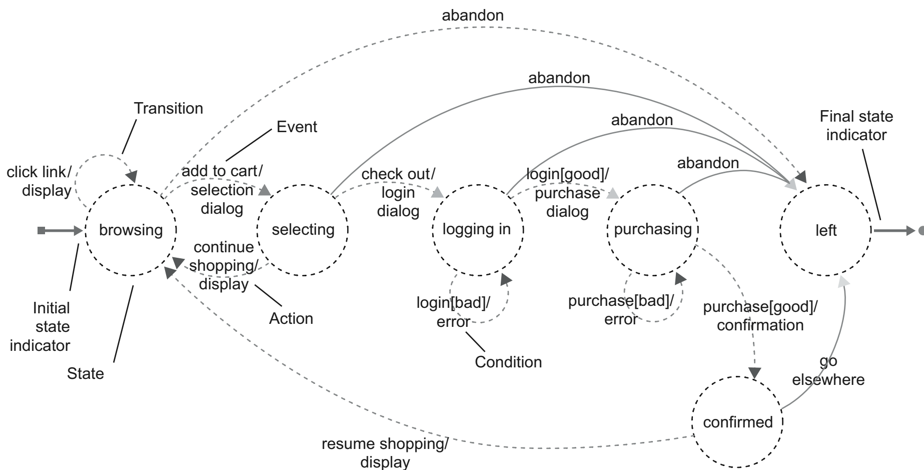


Figure 4–21 *Coverage check completed*

Figure 4-21 shows the test coverage achieved by the following additional test cases (case 1 was listed earlier):

2. (browsing, add to cart, selection dialog, abandon, <no action>, left)
3. (browsing, add to cart, selection dialog, checkout, login dialog, abandon, <no action>, left)
4. (browsing, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)
5. (browsing, add to cart, selection dialog, continue shopping, display, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, purchase[good], confirmation, go elsewhere, <no action>, left)

Remember that you're not done generating tests until every state and every transition has been highlighted, as shown in figure 4-21.

One rule of thumb that can help estimate the number of tests needed to get this minimum coverage: we usually need as many test cases as there are transitions entering the final state. In our example, there are five arrows pointing to the left state. While this rule of thumb often works, it is not officially part of the test design technique.

4.2.4.1 Superstates and Substates

In some cases, it makes sense to unfold a single state into a superstate consisting of two or more substates. In figure 4-22, you can see that we've taken the purchasing state from the e-commerce example and expanded it into three substates.

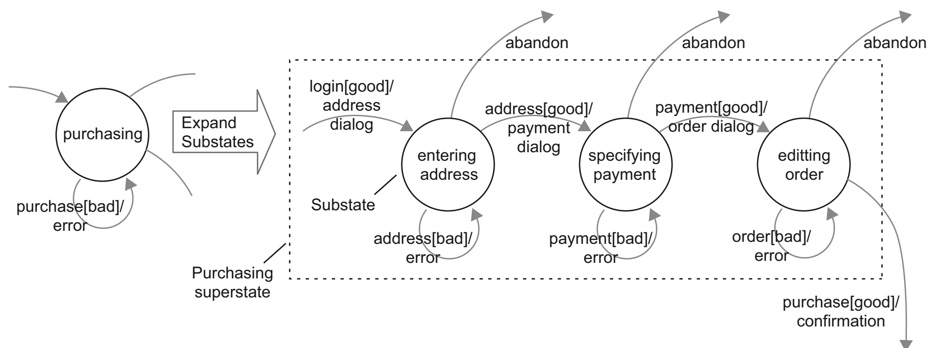


Figure 4–22 *Superstates and substates*

The rule for basic coverage here follows simply. Cover all transitions into the superstate, all transitions out of the superstate, all substates, and all transitions within the superstate.

Note that, in our example, this would increase the number of tests because we now have three "abandon" transitions to the "left" state out of the purchasing superstate rather than just one transition from the purchasing state. This would also add a finer-grained element to our tests—i.e., more events and actions—as well as make sure we tested at least three different types of bad purchasing entries.

ISTQB Glossary

state table: A grid showing the resulting transitions for each state combined with each possible event, showing both valid and invalid transitions.

4.2.4.2 State Transition Tables

State transition tables are useful because they force us—and the business analysts and the system designers—to consider combinations of states with event/condition combinations that they might have forgotten.

To construct a state transition table, you first list all the states from the state transition diagram. Next, you list all the event/condition combinations shown on the state transition diagram. Then, you create a table that has a row for each state with every event/condition combination. Each row has four fields:

- Current state
- Event/condition
- Action
- New state

For those rows where the state transition diagram specifies the action and new state for the given combination of current state and event/condition, we can populate those two fields from the state transition diagram. However, for the other rows in the table, we find undefined situations, i.e., situations where the behavior of the system is not specified.

We can now go to the business analysts, system designers, and other such people and ask, “So, what exactly should happen in each of these situations?”

You might hear them say, “Oh, that can never happen!” As a technical test analyst, you know what that means. Your job now is to figure out how to make it happen.

You might hear them say, “Oh, well, I’d never thought of that.” That probably means you just prevented a bug from ever happening, if you are doing test design during system design.

As an example, consider figure 4-23. Assume that the customer is in the browsing state, having previously put one or more items into the cart. If she decides to check out while in the browsing state, she cannot: that state-event/condition combination is not defined. We have found a missing requirement! If

we actually built the system in this way, we would force our customers to only be able to check out immediately after putting an item in the cart. Our customer now must put an additional item into the cart to be able to check out. Or, she might just decide to move to our competitor's site! A state transition table can be an invaluable tool when looking for missing requirements, much the same way a decision table can.

				Current State	**Event/cond**	**Action**	**New State**
				Browsing	Click link	Display	Browsing
		Click link		Browsing	Add to cart	Selection dia	Selecting
		Add to cart		Browsing	Continue shopping	Undefined	Undefined
		Continue shopping		Browsing	Check out	Undefined	Undefined
Browsing		Check out		Browsing	Login[bad]	Undefined	Undefined
Selecting		Login[bad]		Browsing	Login[good]	Undefined	Undefined
Logging		Login[good]		Browsing	Purchase[bad]	Undefined	Undefined
Purchasing	×	Purchase[bad]	=	Browsing	Purchase[good]	Undefined	Undefined
Confirmed		Purchase[good]		Browsing	Abandon	<no action>	Left
Left		Abandon		Browsing	Resume shopping	Undefined	Undefined
		Resume shopping		Browsing	Go elsewhere	Undefined	Undefined
		Go elsewhere		Selecting	Click link	Undefined	Undefined
				(Fifty-three rows, generated in the pattern shown above, not shown)			
				Left	Go elsewhere	Undefined	Undefined

Figure 4–23 *State transition table example*

Figure 4-23 shows an excerpt of the table we would create for the e-commerce example we've been looking at so far. We have six states:

- Browsing
- Selecting
- Logging in
- Purchasing
- Confirmed
- Left

We have 11 event/condition combinations:

- Click link
- Add to cart

- Continue shopping
- Check out
- Login[bad]
- Login[good]
- Purchase[bad]
- Purchase[good]
- Abandon
- Resume shopping
- Go elsewhere

That means our state transition table should have 66 rows, one for each possible pairing of a specific state with a specific event/condition combination.

To derive a set of tests that covers the state transition table, we can follow the following procedure. Notice that we build on an existing set of tests created from the state transition diagram to achieve state/transition or 0-switch cover:

1. Start with a set of tests (including the starting and stopping state rule), derived from a state transition diagram, that achieves state/transition coverage.
2. Construct the state transition table and confirm that the tests cover all the defined rows. If they do not, then either you didn't generate the existing set of tests properly or you didn't generate the table properly, or the state transition diagram is screwed up. Do not proceed until you have identified and resolved the problem, including re-creating the state transition table or the set of tests, if necessary.
3. Select a test that visits a state for which one or more undefined rows exists in the table. Modify that test to attempt to introduce the undefined event/condition combination for that state. Notice that the action in this case is undefined.
4. As you modify the tests, mark the row as covered. The easiest way to do this is to take a printed version of the table and use a marker to highlight each row as covered.
5. Repeat steps 3 and 4 until all rows have been covered.

Again, this procedure will generate logical test cases. Eventually, either formally (in a scripted test case) or informally (using experience-based testing), you'll need to choose data for execution of the tests.

As an example of deriving state table-based tests, we can build on the e-commerce example already shown. Start with an existing test from the state transition testing; here we select test 4 from earlier:

> (browsing, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)

Now, from here we start to create modified tests to cover undefined browsing event/conditions and those undefined conditions only.

One test is as follows:

> (browsing, *attempt:* continue shopping, *action undefined*, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)

Another test:

> (browsing, *attempt:* check out, *action undefined*, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left)

There are six other modified tests for browsing, which we've not shown. As you can see, it's a mechanical process to generate these tests. As long as you are careful to keep track of which rows you've covered—using the marker trick we mentioned earlier, for example—it's almost impossible to forget a test.

Now, you'll notice that we only included one undefined event/condition combination in each test step. Why? This is a variant of the equivalence partitioning rule that we should not create invalid test cases that combine multiple invalid event/conditions. In this case, each row corresponds to an invalid event/condition. If we try to cover two rows in a single test step, we can't be sure the system will remain testable after the first invalid event/condition.

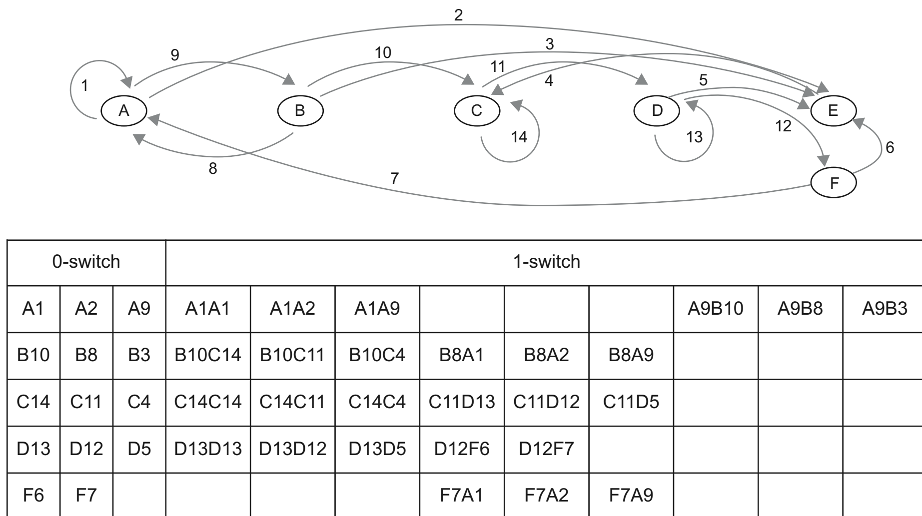
Notice that we indicated that the action is undefined. What is the ideal system behavior under these conditions? Well, the best-case scenario is that the undefined event/condition combination is impossible to trigger. If we cannot get there from here—no menu items, no buttons, no hot-key combinations, no possible URL edits—so much the better. Next best is that the undefined event/condition pair is ignored or—better yet—rejected with an intelligent error message. At that point, processing continues normally. In the absence of any

> **ISTQB Glossary**
>
> *N*-switch testing: A form of state transition testing in which test cases are designed to execute all valid sequences of *N*+1 transitions.

meaningful input from business analysts, the requirements specification, system designers, or any other authority, we would take the position that any other outcome is a bug, including some inscrutable error message like, “What just happened can’t happen.” (No, we are not making that up; an RBCS course attendee once told Rex she had seen exactly that message when inputting an unexpected value.)

4.2.4.3 Switch Coverage

Figure 4-24 shows how we can generate sequences of transitions using the concept of switch coverage. We will illustrate this concept with the e-commerce example we’ve used so far.



0-switch			1-switch								
A1	A2	A9	A1A1	A1A2	A1A9				A9B10	A9B8	A9B3
B10	B8	B3	B10C14	B10C11	B10C4	B8A1	B8A2	B8A9			
C14	C11	C4	C14C14	C14C11	C14C4	C11D13	C11D12	C11D5			
D13	D12	D5	D13D13	D13D12	D13D5	D12F6	D12F7				
F6	F7					F7A1	F7A2	F7A9			

Figure 4–24 *N-1 switch coverage example*

At the top of figure 4-24, you see the same state transition as before, except we have replaced the state labels with letters and the transition labels with numbers. Now a state/transition pair can be specified as a letter followed by a number. Notice that we are not bothering to list, in the table, a letter after the number

because it's unambiguous from the diagram what state we'll be in after the given transition. There is only one arrow labeled with a given number that leads out of a state labeled with a given letter, and that arrow lands on exactly one state.

The table contains two types of columns. The first contains the state/transition pairs that we must cover to achieve 0-switch coverage. Study this for a moment, and assure yourself that, by designing tests that cover each state/transition pair in the 0-switch columns, you'll achieve state/transition coverage as discussed previously.

Constructing the 0-switch columns is easy. The first row consists of the first state, with a column for each transition leaving that state. There are at most three transitions from the A state. Repeat that process for each state for which there is an outbound transition. Notice that the E state doesn't have a row, and that's because E is a final state and there's no outbound transition. Notice also that, for this example, there are at most three transitions from any given state.

The 1-switch columns are a little trickier to construct, but there's a regularity here that makes it mechanical if you are meticulous. Notice, again that after each transition occurs in the 0-switch situation, we are left in a state which is implicit in the 0-switch cells. As mentioned, there are at most three transitions from any given state. So that means that, for this example, each 0-switch cell can expand to at most three 1-switch cells.

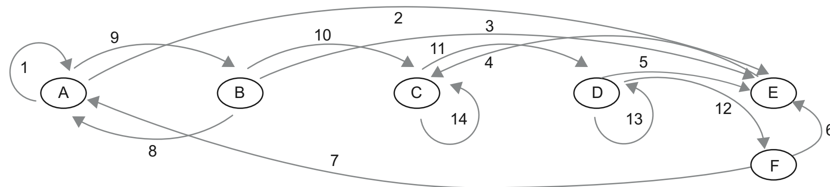
So, we can take each 0-switch cell for the A row and copy it into three cells in the 1-switch columns, for nine cells for the A row. Now, we ask ourselves, for each triple of cells in the A row of the 1-switch columns, what implicit state did we end up in? We can then refer to the appropriate 0-switch cells to populate the remainder of the 1-switch cell.

Notice that the blank cells in the 1-switch columns indicate situations where we entered a state in the first transition from which there was no outbound transition. In figure 4-24, that is the state labeled E, which was labeled "Left" on the full-sized diagram.

So, given a set of state/transition sequences like those shown—whether 0-switch, 1-switch, 2-switch, or even higher—how do we derive test cases to cover those sequences and achieve the desired level of coverage? Again, we're going to build on an existing set of tests created from the state transition diagram to achieve state/transition or 0-switch coverage.

1. Start with a set of tests (including the starting and stopping state rule), derived from a state transition diagram, that achieves state/transition coverage.
2. Construct the switch table using the technique shown previously. Once you have, confirm that the tests cover all of the cells in the 0-switch columns. If they do not, then either you didn't generate the existing set of tests properly or you didn't generate the switch table properly, or the state transition diagram is wrong. Do not proceed until you have identified and resolved the problem, including re-creating the switch table or the set of tests, if necessary. Once you have that done, check for higher-order switches already covered by the tests.
3. Now, using 0-switch sequences as needed, construct a test that reaches a state from which an uncovered higher-order switch sequence originates. Include that switch sequence in the test. Check to see what state this left you in. Ideally, another uncovered higher-order switch sequence originates from this state, but if not, see if you can use 0-switch sequences to reach such a state. You're crawling around in the state transition diagram looking for ways to cover higher-order sequence. Repeat this for the current test until the test must terminate.
4. As you construct tests, mark the switch sequences as covered once you include them in a test. The easiest way to do this is to take a printed version of the switch table and use a marker to highlight each cell as covered.
5. Repeat steps 3 and 4 until all switch sequences have been covered.

Again, this procedure will generate logical test cases.



0-switch			1-switch								
A1	A2	A9	A1A1	A1A2	A1A9				A9B10	A9B8	A9B3
B10	B8	B3	B10C14	B10C11	B10C4	B8A1	B8A2	B8A9			
C14	C11	C4	C14C14	C14C11	C14C4	C11D13	C11D12	C11D5			
D13	D12	D5	D13D13	D13D12	D13D5	D12F6	D12F7				
F6	F7					F7A1	F7A2	F7A9			

Figure 4–25 *Deriving tests example*

In figure 4-25, we see the application of the derivation technique covered in figure 4-24 to the e-commerce example we've used. After finishing the second step, that of assessing coverage already attained via 0-switch coverage, we can see that most of the table is already shaded. Those are the lighter-shaded cells, which are covered by the five existing state/transition cover tests.

Now, we generate five new tests to achieve 1-switch coverage. Those are shown below. The darker-shaded cells are covered by five new 1-switch cover tests.

1. (A1A1A2).
2. (A9B8A1A9B8A2).
3. (A9B10C14C14C4).
4. (A9B10C11D13D13D5).
5. (A9B10C11D12F7A1A9B10C11D12F7A9).

We need to mention something about this algorithm for deriving higher-order switch coverage tests, as well as the one given previously for row-coverage tests. Both build on an existing set of tests that achieve state/transition coverage. That is efficient from a test design point of view. It's also conservative from a test execution point of view because we cover the less challenging stuff first and then move on to the more difficult tests.

However, it is quite possible that, starting from scratch, a smaller set of tests could be built, both for the row coverage situation and for the 1-switch coverage situation. If the most important thing is to create the minimum number of tests, then you should look for ways to reduce the tests created, or modify the derivation procedures given here to start from scratch rather than to build on an existing set of 0–switch tests.

4.2.4.4 State Testing with Other Techniques

Let's finish our discussion of state-based testing by looking at a couple of interesting questions. First, how might equivalence partitioning and boundary value analysis combine with state-based testing? The answer is, quite well.

From the e-commerce example, suppose that the minimum purchase is $10 and the maximum is $10,000. In that case, we can perform boundary value analysis as shown in figure 4-26, performed on the purchase[good] and purchase[bad] event/condition combinations. By covering not only transitions,

rows, and transition sequences, but also boundary values, this forces us to try different purchase amounts.

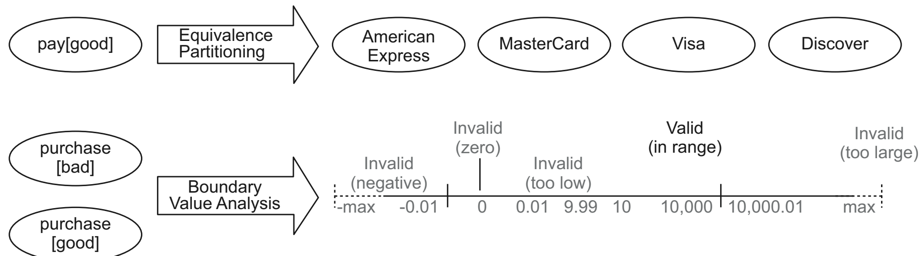


Figure 4–26 *Equivalence partitions and boundary values*

We can also apply equivalence partitioning to the pay[good] event/condition combination. For example, suppose we accept four different types of credit cards. By covering not only transitions, rows, and transition sequences, but also equivalence partitions, this forces us to try different payment types.

Now, to come full circle on a question we brought up at the start of the discussion on these two business-logic test techniques. When do we use decision tables and when do we use state diagrams?

This can be, in some cases, a matter of taste. The decision table is compact. If we're not too worried about the higher-order coverage, or the effect of states on the tests, many state-influenced situations can be modeled as decision tables, using conditions to model states. However, if the decision table's conditions section starts to become very long, you're probably stretching the technique. Also, keep in mind that test coverage is usually more thorough using state-based techniques.

In most cases, one technique or the other will clearly fit better. If you are at a loss, try both and see which feels most appropriate.

4.2.4.5 State Testing Exercise

This exercise consists of three parts:

1. Using the following semiformal use case, translate it into a state transition diagram, shown from the point of view of the Telephone Banker.
2. Generate test cases to cover the states and transitions (0-switch).
3. Generate a switch table to the 1-switch level.

Table 4–26

Actor	Telephone Banker
Preconditions	The Globobank Telephone Banker is logged into the HELLOCARMS System.
Normal Workflow	1. The Telephone Banker 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 his Desktop. 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 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.
Exception Workflow 1	During step 2 of the normal workflow, 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. If the decision is to proceed, then the Telephone Banker completes the remainder of step 2 and proceeds normally. If the decision is not to proceed, the Telephone Banker informs the Customer that the application is declined and the interview ends.
Exception Workflow 2	During step 4 of the normal workflow, if the System does not display any Home Equity Products as available, the Telephone Banker informs the Customer that the application is declined and the interview ends.
Exception Workflow 3	During step 5 of the normal workflow, if the product chosen by the Customer was a Home Equity Loan, the Telephone Banker offers the Customer the option of applying for life insurance to cover the loan. If the Customer wants to apply, the following steps occur: 1. The Telephone Banker interviews the Customer, entering health information into the HELLOCARMS System through a web browser interface on his Desktop. 2. The HELLOCARMS System processes the information as described in the previous exercise. One of two outcomes will occur: a. The HELLOCARMS System declines to offer insurance based on the health information given. The Telephone Banker informs the Customer that the insurance application was denied. This exception workflow is over and processing returns to step 5. b. The HELLOCARMS System offers insurance at a rate based on the loan size and the health information given. The Telephone Banker informs the Customer of the offer. 3. The Customer makes one of two decisions: a. Accept the offer. The Telephone Banker makes the life insurance purchase part of the overall application. This exception workflow is over and processing returns to step 5. b. Reject the offer. The Telephone Banker excludes the life insurance purchase from the overall application. This exception workflow is over and processing returns to step 5.

Exception Workflow 4	During any of steps 1 through 5 of the normal workflow, if the Customer chooses to end the interview without continuing the process or selecting a product, the application is cancelled and the interview ends.
Exception Workflow 5	If no Telephone Banker is logged into the system (e.g., because the system is down) and step 1 of the normal workflow begins, the following steps occur: 1. The Telephone Banker takes the information manually. At the end of the interview, the Telephone Banker informs the Customer that a Telephone Banker will call back shortly with the decision on the application. 2. Once a Telephone Banker is logged into the System, the application information is entered into HELLOCARMS and normal processing resumes at step 2. 3. The Telephone Banker calls the Customer once one of the following outcomes has occurred: a. Step 5 of normal processing is reached. Processing continues at step 5. b. At step 2 of normal processing, exception workflow 1 was triggered. Processing continues at step 2. c. At step 4 of normal processing, exception workflow 2 was triggered. No processing remains to be done.
Postconditions	Loan application is in LoDoPS system for origination

4.2.4.6 State Testing Exercise Debrief

1. Create state transition diagram.

Figure 4-27 shows the state transition diagram we generated based on the preceding semiformal use case.

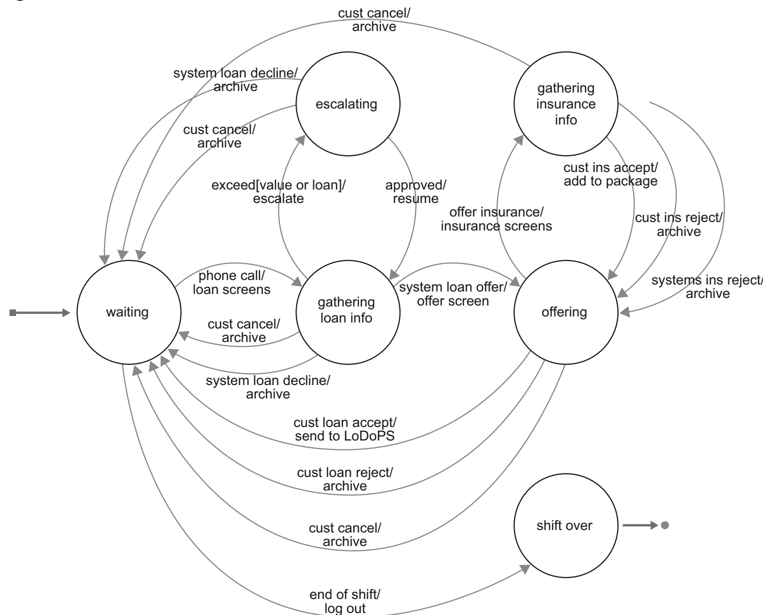


Figure 4–27 HELLOCARMS state transition diagram

2. Generate test cases to cover the states and transitions (0-switch).

Let's adopt a rule that says that any test must start in the initial waiting state and may only end in the *waiting* state or the *shift over* state. To achieve state and transition coverage, the following tests will suffice:

1. (waiting, phone call, loan screens, exceed[value], escalate, approved, resume, system loan offer, offer screen, offer insurance, insurance screens, cust ins accept, add to package, cust loan accept, send to LoDoPS, waiting)
2. (waiting, phone call, loan screens, exceed[loan], escalate, approved, resume, system loan offer, offer screen, offer insurance, insurance screens, cust ins reject, archive, cust loan accept, send to LoDoPS, waiting)
3. (waiting, phone call, loan screens, system loan offer, offer screen, offer insurance, insurance screens, system ins reject, archive, cust loan reject, archive, waiting)
4. (waiting, phone call, loan screens, exceed[loan], escalate, system loan decline, archive, waiting)
5. (waiting, phone call, loan screens, system loan decline, archive, waiting)
6. (waiting, phone call, loan screens, cust cancel, archive, waiting)
7. (waiting, phone call, loan screens, exceed[loan], escalate, cust cancel, archive, waiting)
8. (waiting, phone call, loan screens, system loan offer, offer screen, cust cancel, archive, waiting)
9. (waiting, phone call, loan screens, system loan offer, offer screen, offer insurance, insurance screens, cust cancel, archive, waiting)
10. (waiting, end of shift, log out, shift over)

Notice that we didn't do an explicit boundary value or equivalence partitioning testing of, say, the loan amount or the property value, though we certainly could have. Also, note that this is an example of when our rule of thumb for number of needed test cases did not work. This usually happens because there are multiple paths between two non-final states (in this case between *offering* and *gathering insurance info*) that must be tested with separate test cases.

3. Generate a switch table to the 1-switch.

First, redraw the state transition diagram of figure 4-27 to make it easier to work with. It should look like figure 4-28.

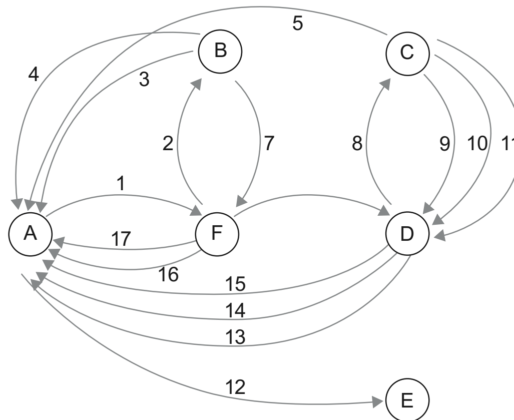


Figure 4–28 *HELLOCARMS state transition diagram for switch table*

From the diagram, we can generate the 1-switch table shown in table 4-27. Notice that we have used patterns in the diagram to generate the table. For example, the maximum number of outbound transitions for any state in the diagram is four, so we use four columns on both the 0-switch and 1-switch columns. We started with six 1-switch rows per 0-switch row because there are six states, though we were able to delete most of those rows as we went along. This leads to a sparse table, but who cares as long as it makes generating this beast easier.

Table 4–27

0-switch				*1-switch*			
A1	A12			A1F2	A1F6	A1F16	A1F17
B3	B4	B7		B3A1	B3A12		
				B4A1	B4A12		
				B7F2	B7F6	B7F16	B7F17
C5	C9	C10	C11	C5A1	C5A12		
				C9D8	C9D13	C9D14	C9D15
				C10D8	C10D13	C10D14	C10D15
				C11D8	C11D13	C11D14	C11D15
D8	D13	D14	D15	D8C5	D8C9	D8C10	D8C11
				D13A1	D13A12		
				D14A1	D14A12		
				D15A1	D15A12		
F2	F6	F16	F17	F2B3	F2B4	F2B7	
				F6D8	F6D13	F6D14	F6D15
				F16A1	F16A12		
				F17A1	F17A12		

4.2.5 Requirements-Based Testing Exercise

This exercise requires you to select which specification-based techniques to use to test requirement element 040-020-030 from the HELLOCARMS system Requirements Document.

The exercise consists of two parts:

1. Select appropriate techniques for test design.
2. Apply those techniques to generate tests, achieving the coverage criteria for the techniques.

As always, check your work on the first part before proceeding to the second part. The solutions are shown in the following section.

4.2.6 Requirements-Based Testing Exercise Debrief

The requirement in question reads as follows:

> *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.*

We have discussed four specification-based techniques in this section: equivalence partitioning, boundary value analysis, decision tables, and state transition testing. The former two will be very useful in being able to test this requirement; however, no clear value of using decision tables or state transition for this example comes immediately to mind.

The following analysis will be done with the requirement as it stands. While the CPU requirement is clear, the resource utilization is not. Which resources? Since there are dozens of resources that we can monitor, it is not really clear. This is often the case when we get requirements documents that have not been subjected to rigorous static review.

For this exercise, we have decided to pick a few main resources: the physical disk, memory utilization, page file usage, and network utilization as a percentage of bandwidth. We might expect (hope?) that we would get additional clarity via static review of the requirements.

One more clarification. On the resources we are looking at, we do not believe that any of them, other than CPU, can be run at a full 100 percent. For each one, there will be a maximum utilization somewhat below 100 percent. We

would expect the modeling portion of the performance test would establish exactly what the reasonable top end will be. For our equivalence class identification and boundary value analysis, we will call that top end 100 percent and manage our peak utilization to 80 percent of that value. For CPU, 80 percent would actually mean 80 percent usage.

Based on the requirement, all of the measurements would have the same equivalent classes and boundary values as shown in figure 4-29 and figure 4-30.

Figure 4–29 *Instantaneous measurement boundaries*

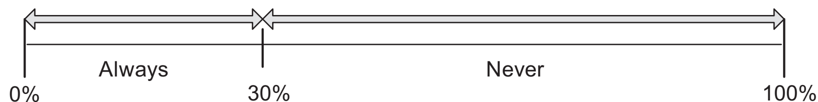


Figure 4–30 *Average measurement boundaries*

For instantaneous measurements, the valid high boundary would be 80 percent. The invalid boundary would be just over 80 percent. Epsilon would be dependent on the quality of our measuring tools (for this exercise, we will assume an epsilon of zero). For average measurements, over the life of the test, 30 percent would be the valid high boundary. Invalid high boundary would be just over 30 percent. Exact values are going to depend on the instrumentation used.

Testing would consist of modeling the system, the test environment, and other setup tasks consistent with the techniques discussed in chapters 5 and 9.

We would start executing the test via our performance tool by applying virtual users to the system slowly, making sure that functionality is working as expected. Monitoring tools would be used to measure instantaneous values and store them for later calculation of averages. The load would consist of virtual users performing all acceptable operations, modeling the real world usage of the system.

Over a defined time, we would increase loading until the system is clocking at the rated value of concurrent users. All of this presupposes that we have hard-

ISTQB Glossary

structure-based design technique (white-box test design technique)**:** Procedure to derive and/or select test cases based on an analysis of the internal structure of a component or system.

ware equivalent to that of production. This assumption is often wrong in our experience. If the hardware that we are testing on is appreciably smaller than production is expected to be, we would have to use extrapolation to try to determine what the actual scaled values the test should return.

4.3 Structure-Based

Learning objectives

(K2) List examples of typical defects to be identified by each specific structure-based[7] technique.

(K3) Write test cases in real-life using the following test design techniques (the tests shall achieve a given model coverage).

- Statement testing
- Decision testing
- Condition determination testing
- Multiple condition testing

(K4) Analyze a system in order to determine which structure-based technique to apply for specific test objectives.

(K2) Understand each structure-based technique and its corresponding coverage criteria and when to use it.

(K4) Be able to compare and analyze which structure-based technique to use in different situations.

Structural-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.

7. In the ISTQB Advanced Syllabus, 2007, the learning objective is actually incorrect. It says “specific specification-based technique” but clearly means “structure-based”.

The question that should come to mind is *why*. We have all kinds of specification-based (black-box) testing methods to choose from. We just spent dozens of pages going through four of the many different black-box design methods. Why do we need more? We don't have time or resources to spare for extra testing, do we?

Well, consider a world-class, excellent 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 use error guessing and checklist-based methods (which we will discuss later). At the end of that, have they done enough testing? Maybe. But research has shown that even with all of that testing, and all of that effort, they may have missed a few things.

Maybe as much as 70 percent of all of the code in the system might never have been executed once! Not once!

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 everyone 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 did not sometimes fail, and databases didn't 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 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 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 those levels.

In addition, black-box testing is predicated on having models which 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. Internal data flows often occur between hidden devices which have asynchronous timing triggers, invisible to black-box testers.

Much of white-box testing is involved with coverage—making sure that we have tested everything that we can 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 stuff we wanted to. In the following sections, we are going to discuss how to design, create, and execute white-box testing.

4.3.1 Control-Flow Testing

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, a way of abstracting a code module in order 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 build a system and send it out without running it at least once. At the other end of the spectrum, exhaustive testing would require an infinite amount of time and resources. In the middle, however, there is a wide range of coverages that are possible.

We will look at different reasons for testing to some of these different levels of coverage later. 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. Synonyms that are used for this include *instruction* and *code* coverage; each one means the same thing: Have we exercised, at one time or another, every single

line of 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.

The next step up in control-flow testing is called *decision* (or *branch*) coverage. This is determined by the total number of decisions in the code that we have exercised, both ways. We will honor the ISTQB decision to treat *branch* and *decision* testing as synonyms. There are very slight differences between the two, but those differences are insignificant at the level we will examine them.[8]

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

The ISTQB Advanced syllabus lists a level of coverage called *condition determination* (we will use the term *decision/condition* coverage) where we test all combinations of outcomes for individual conditions that can affect a decision outcome. Closely related to that is a mouthful we call *multiple condition/decision* coverage (also known as *MC/DC*).

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

Then we will look at various path coverage testing schemes, including one called *Linear Code Sequence and Jump* (*LCSAJ*) coverage.

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 technique essentially builds on the shortcoming of the previous technique.

4.3.1.1 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.

---

8. 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 *Software Verification Tools Assessment Study, FAA, June 2007.*

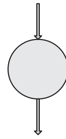


Figure 4–31 *The process block*

In figure 4-31, 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 important 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 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. 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.

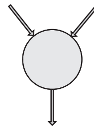


Figure 4–32 *The junction point*

The second structure we need to discuss, seen in figure 4-32, is called a junction point. 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. Again, no decisions are made in this block; the indicated roads lead to it with only one road out.

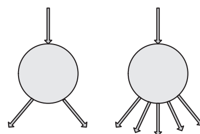


Figure 4–33 *Two decision points*

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 go is made and control-flow continues out that path while ignoring all of the other possible choices. In figure 4-33, 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.

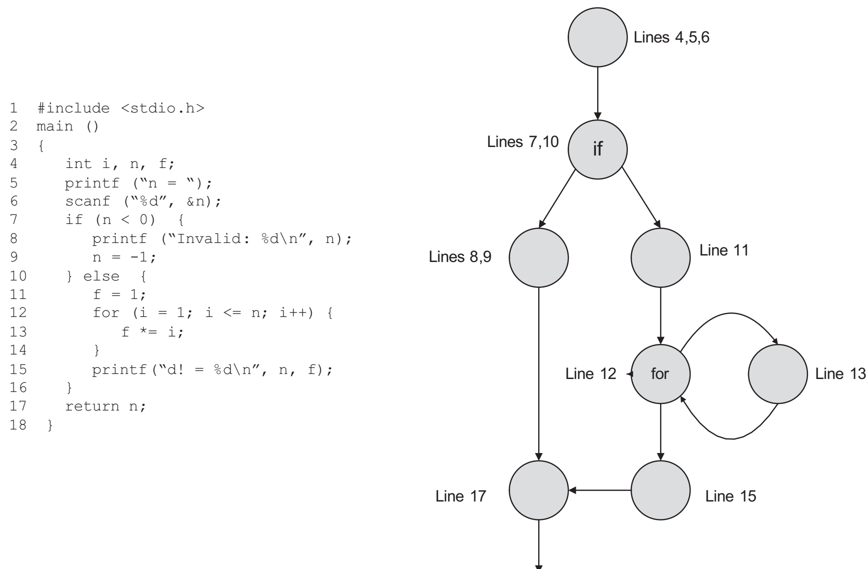


Figure 4–34 *A simple control-flow graph*

In figure 4-34, 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 one. The following are the first three factorials:

1! = 1
2! = 1 * 2 ==> 2
3! = 1 * 2 * 3 ==> 6 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 that 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 can 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 resolves to FALSE, the thread jumps to line 10 to execute the *else* clause.
4. Line 11 sets a value, and then goes to line 12 where another decision is made. This is 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 goes 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).

4.3.1.2 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

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

statements are the basis for test design selection. To achieve statement coverage, we pick test data that force the thread of execution to go through each line of code that the system contains.

To calculate the current level of coverage that we have attained, we divide the number of code statements that are executed by the number of code statements in the entire system; if the quotient is equal to one, we have 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. 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 that the developers they work with are familiar with the concepts we are discussing here. Achieving (at least) statement coverage should be done while unit testing.

There is incontrovertibly no guarantee that even college-educated developers learned how important this coverage level is. Jamie has an undergraduate computer science major and 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.

IEEE, in the 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*, 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 go and look at how to do it.

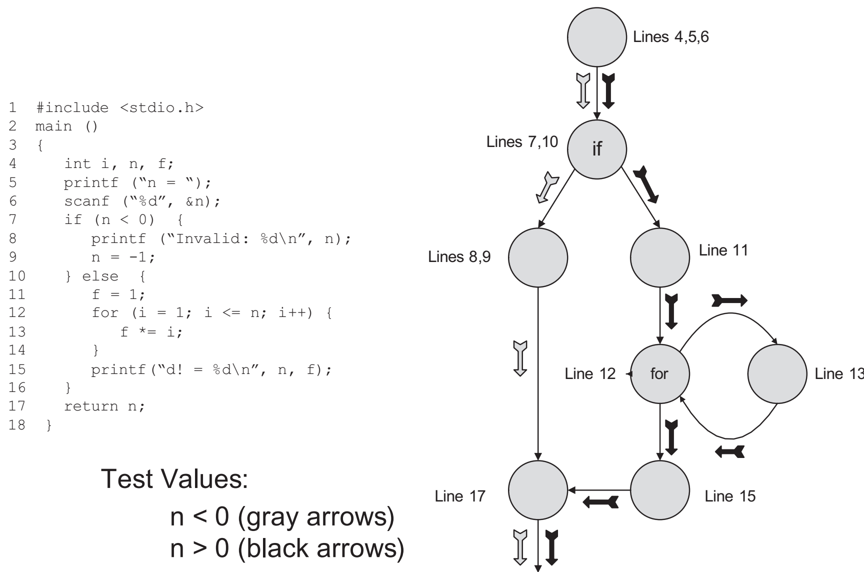


Figure 4–35 *Statement coverage example*

Looking at the same code and control-flow graph we looked at before, figure 4-35 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 code, but not all. Based on the graph, we still have a way to go to reach this minimum 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 one, the loop will fire at least once (more times if the input is greater than one). When *i* increments to larger than *n*, 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 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 4-35 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, we are wasting 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 statement coverage. We will discuss those in chapter 9.

The real question we should ask is, "Did we test enough for the context of our project?" And the answer, of course, is, "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. Maybe we can do a little more testing and get a lot better coverage.

Figure 4-36 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.

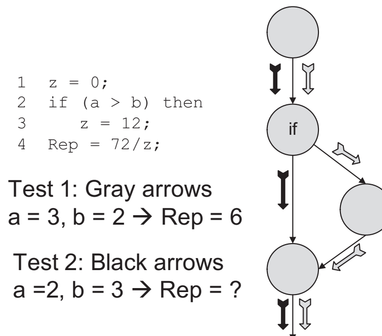


Figure 4–36 *Where statement coverage fails*

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

> **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.*

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 which is almost guaranteed to miss defects. We need to look at another, higher level of coverage that can catch this particular kind of defect.

4.3.1.3 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 seems 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 which 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 decision level of coverage, every decision made by the code must be tested both ways, TRUE and FALSE. That means—at minimum—two test cases must be run: one with data that cause the evaluation to resolve TRUE and a separate test case where the data cause the decision to resolve FALSE. If you omit one test or the other, then you do not achieve decision coverage.

Our bug hypothesis was proved out in figure 4-36. An untested branch may leave a landmine 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 code. 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 the extent of decision coverage.

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 decision 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* is greater than *b*, if n is less than zero. 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 which resolves to a single Boolean value:

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

This expression has three sub-expressions in it, separated by Boolean operators. First, we evaluate the sub-expression that determines if the sum of *x* plus *y* is equal to -1. That will be either TRUE or FALSE. We then AND that to the third sub-expression, which calculates whether *d* is FALSE or not. The result of that calculation is then ORed to the TRUE or FALSE result testing whether *a* is greater than *b*. If that order of execution is not intuitive, it is based on the rules that govern the relative order of expression evaluation: ANDs are evaluated before ORs, much the same way multiplications are evaluated before additions in arithmetic.

While this expression appears to be ugly, the compiler will evaluate the entire predicate 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 4–28

Either/or Decisions	Loop decisions

if (expr)
  {}
else
  {}
	while(expr)
	  {}

switch (expr) {
  case const_1: {} break;
  case const_2: {} break;
  case const_3: {} break;
  case const_4: {} break;
  default {}
}
	do
	  {}
	while (expr)
	
	for (expr_1; expr_2; expr_3)
	  {}

Here, in table 4-28, you can see a short list of the decision constructs that the popular language C uses. Other popular languages use similar constructs. The commonality between all of these (and all of the other decision constructs in all of the other languages) is that each makes a decision that can go only two ways: TRUE or FALSE.

Looking back at our original example in figure 4-35, with just two test cases we do not achieve decision coverage, even though we do attain statement coverage. We did not "*not*" execute the *for* loop. 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 4-37. When 0 is entered, the first decision evaluates to FALSE, so we take the else path. At line 12, the predicate (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.

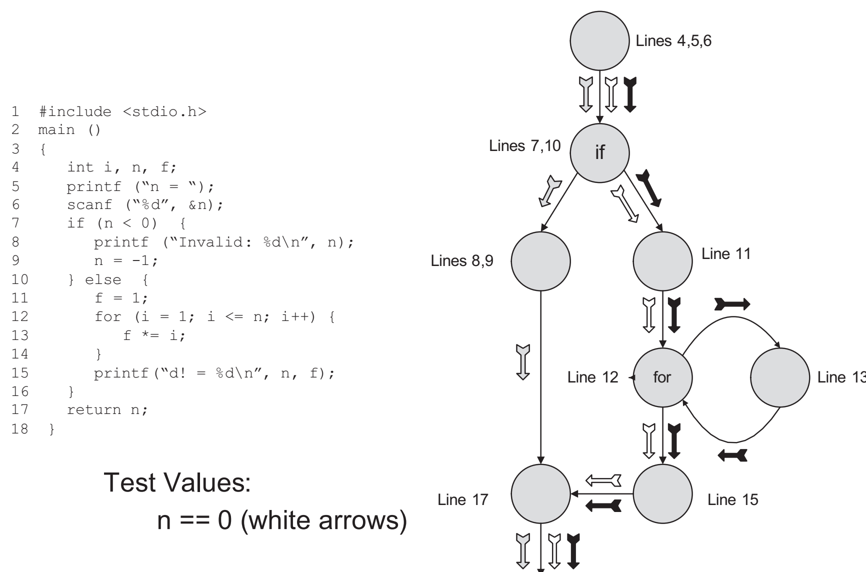


Figure 4–37 *Decision coverage example*

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.

4.3.1.4 Loop Coverage

While not strictly discussed in the ISTQB Advanced syllabus, loop testing should be considered an important 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 is not 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. 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 Beizer'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.

```
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 Values:

n = 0 (white arrows)
n = 1 (black arrows)

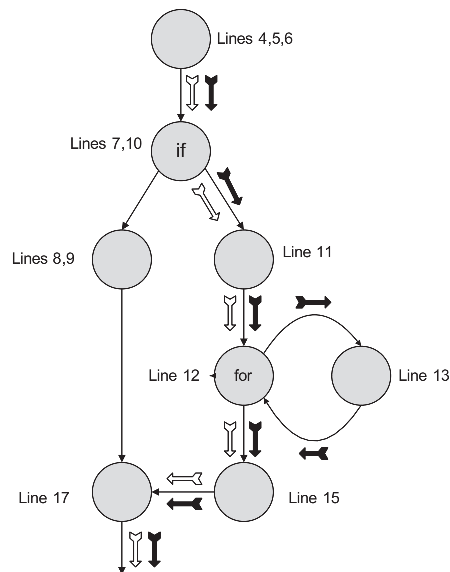


Figure 4–38 *Loop coverage 0 and 1 times through*

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. At this point, 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; 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 4-29, we show a table with factorial values.

Table 4–29

n	n!
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5,040
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 4-39, 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(“d! = %d\n”, n, f);
16    }
17    return n;
18 }
```

Test Values:
n = 12 (black arrows)
Loops 12 times
n = 13 (not shown)
Overflow failure

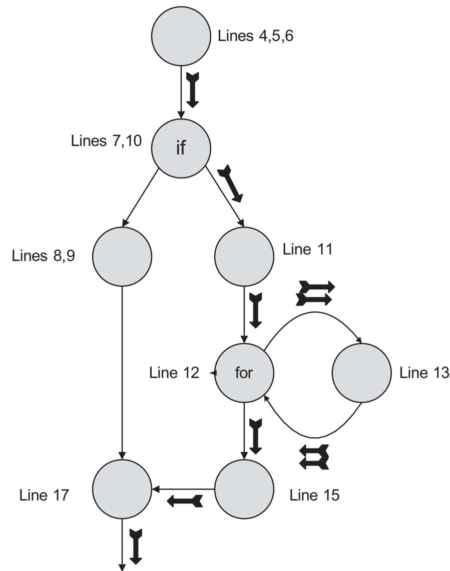


Figure 4–39 *Loop coverage max times through*

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 being able 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 previous section.

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. Try to test 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. Try to test one less than the maximum value.
8. Try to test the maximum number of loops.
9. Try to test 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 shown previously.
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.

4.3.1.5 Hexadecimal Converter Exercise

In figure 4-40, you'll find a C program 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.

If you test with input strings "24ABd690BBCcc" and "ABCdef1234567890", 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, "hexcvt: 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 4–40 *Hexadecimal converter code*

ISTQB Glossary

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

4.3.1.6 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

In order to get loop coverage, you would have to add the following:

- At least one empty string to cause the *while* loop on line 17 not to loop
- One test that inputs the maximum number of hex digits that can be stored (based on the data type of *nhex*)

There are also some interesting test cases based on placement that we would likely run. 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

4.3.1.7 Condition 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, we saw that they could be simple—such as *A* is greater than *B*—or arbitrarily complex as follows:

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

If we input data to force the entire condition to TRUE, then we go one way; force it 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 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 that each atomic condition is tested both ways, TRUE and FALSE.

An atomic condition is defined as "*the simplest form of code that can result in a TRUE or FALSE outcome.*"[9]

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.

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 inte-

9. Definition from *The Software Test Engineer's Handbook*, Bath and McKay

ger, as 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 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 whole expression.

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

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

(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 following the same rule. In the last sub-expression, (*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 it 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 it evaluated 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)!=TRUE) needs two test cases.

> **ISTQB Glossary**
>
> **condition determination testing:** A white-box test design technique in which test cases are designed to execute single condition outcomes that independently affect a decision outcome.

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

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

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

Test 1: *A* == FALSE, *B* == TRUE resolves to FALSE

Test 2: *A* == TRUE, *B* == 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 decision to resolve to TRUE. Condition coverage does not automatically guarantee decision coverage.

We will look at two more levels of coverage that give stronger coverage than simple condition coverage, at the cost of more testing, of course.

4.3.1.8 Decision/Condition Coverage

The first of these we will call decision/condition coverage. In the ISTQB Advanced syllabus, this is called condition determination 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 last 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 assuring that the overall expression be tested both ways, TRUE and FALSE.

To test to this level of coverage, we must assure 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.

Decision/condition coverage guarantees condition coverage, decision coverage, and statement coverage.

Going back to the previous example again where we have condition coverage but not decision coverage, we already had two test cases as follows: one where *A* is set to FALSE and *B* is set to TRUE and one where *A* is set to TRUE and *B* is set to FALSE. By adding a third test case where both *A* and *B* are set to TRUE, we now force the expression to evaluate to TRUE, so we now have decision coverage also.

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

Well maybe.

4.3.1.9 Modified Condition/Decision Coverage (MC/DC)

There is yet a stronger level of coverage that we must discuss. This one is called modified condition/decision coverage (usually abbreviated to MC/DC).

This level of coverage is considered stronger because we add another factor 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 one more factor: *Each condition must affect the outcome decision independently while the other atomic conditions are held fixed.*

To get to this level of coverage, we determine the test cases needed for decision/condition and then evaluate our tests to see if each atomic factor had the possibility of affecting the outcome decision independently while not varying

the other atomic condition values. Our bug hypothesis states that we might find an example of a bug hiding in that last little space that we have not tested.

Many references use the term *MC/DC* as a synonym for exhaustive testing. However, while achieving MC/DC will force a lot of testing, we do not feel it rises to the level of exhaustive testing, which would include running every loop every possible number of times, etc.

MC/DC coverage may not be useful to all testers. It was originally created at Boeing for use when specific languages were to be used in safety-critical programming. It is the required level of coverage under FAA DO/178B when the software is judged to be Level A (possible catastrophic consequences in case of failure). We discussed this standard in chapter 2.

Testing for MC/DC coverage can be complicated. There are two issues that must be discussed: short circuiting and multiple occurrences of a condition.

First, we will look at short circuiting. What does short circuiting mean?

Some programming languages are defined in such a way that Boolean expressions may be resolved without going through every sub-expression, depending on the Boolean operator that is used.

Table 4–30

Value of A	Value of B	A||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 ($A \,||\, B$) (see table 4-30). 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 exhibiting this behavior. Some flavors of C short-circuit expressions. Pascal does not short-circuit, but Delphi 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 ($A \,||\, B$). When the Boolean operator is an OR, it will short-circuit when the first term is TRUE since the second term does not matter. If the

Boolean operator was an AND (&&), it would short-circuit when the first term was FALSE, since no value of the second term would prevent the expression evaluating as FALSE.

There may be a very serious bug issue with short circuiting. If an atomic condition is supplied by 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 instead of

(A || B)

the actual expression is

(A || funcB())

Further suppose that a side effect of *funcB()* is to initialize a data structure that is to be used later. In her code, the developer could easily assume that the data structure is already initialized since she knew that the predicate had been used in a conditional. When she tries to use the noninitialized data structure, it causes a failure at runtime. To quote Robert Heinlein, for testers, "*there ain't no such thing as a free lunch*."

If *B* 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 the short circuiting.

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 || (! A && B)

In this example, *A* and *!A* are said to be coupled. They cannot be varied independently as MC/DC coverage says they must. 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 condition to vary at once; the tester must perform analysis, on a case-by-case basis, the logic of the predicate to ensure that only one 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.

In the interest of looking at an example that does not lure us deep into the weeds, let us assume we are using a compiler that does not short-circuit and we will not use any coupled conditions. How do we achieve MC/DC coverage?

Consider the code snippet that follows:

if *((A OR B) AND C)* then…

We have three atomic conditions, *A*, *B* and *C*. We can achieve decision/condition coverage by running two test cases as shown:

1. *A* set to TRUE, *B* set to TRUE, *C* set to TRUE where the expression evaluates to TRUE
2. *A* set to FALSE, *B* set to FALSE, *C* set to FALSE where the expression evaluates to FALSE

Note that *B* never independently affects the outcome of the decision since the value of *A* always overrides it.

We can achieve the desired MC/DC coverage by changing the test inputs slightly and adding a single test.

1. *A* set to TRUE, *B* set to FALSE, *C* set to TRUE, which evaluates to TRUE
2. *A* set to FALSE, *B* set to TRUE, *C* set to TRUE, which evaluates to TRUE
3. *A* set to TRUE, *B* set to TRUE, *C* set to FALSE, which evaluates to FALSE

As you can see, each one of the atomic conditions changes the output evaluation at least once in the test set as long as the other conditions are held steady. If *A* changed value in test 1, the result would be inverted, likewise *B* in test 2 and *C* in test 3.

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

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).

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 in any module for the project.

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

4.3.1.10 Multiple Condition Coverage

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

Creating a set of tests does not take much analysis. Simply 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 directly proportional to the number of atomic conditions, using the formula 2^n power where n is the number of atomic conditions.

The bug hypothesis is really pretty simple also: Bugs could be anywhere!

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.

Using the same predicate we discussed earlier

if *((A OR B) AND (C))* then ...

we can devise the truth table shown in table 4-31. Since there are three atomic conditions, we would expect to see 2^3, or 8 unique rows in the table.

Table 4–31

A	B	C	(A OR B) AND C
FALSE	FALSE	FALSE	FALSE
TRUE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	FALSE
TRUE	TRUE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE
TRUE	FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE

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

Why theoretically? Well, we 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 shortcircuiting compiler.

As a further example, assume that we have five atomic conditions, *A*, *B*, *C*, *D*, and *E*. Assuming that we have a compiler that does not short-circuit, we will have 32 test cases to run, as shown by table 4-32. No matter how they are interconnected logically, these will be our test data.

Table 4–32

Test	A	B	C	D	E
1	TRUE	TRUE	TRUE	TRUE	TRUE
2	TRUE	TRUE	TRUE	TRUE	FALSE
3	TRUE	TRUE	TRUE	FALSE	TRUE
4	TRUE	TRUE	TRUE	FALSE	FALSE
5	TRUE	TRUE	FALSE	TRUE	TRUE
6	TRUE	TRUE	FALSE	TRUE	FALSE
7	TRUE	TRUE	FALSE	FALSE	TRUE
8	TRUE	TRUE	FALSE	FALSE	FALSE
9	TRUE	FALSE	TRUE	TRUE	TRUE
10	TRUE	FALSE	TRUE	TRUE	FALSE
11	TRUE	FALSE	TRUE	FALSE	TRUE
12	TRUE	FALSE	TRUE	FALSE	FALSE
13	TRUE	FALSE	FALSE	TRUE	TRUE
14	TRUE	FALSE	FALSE	TRUE	FALSE

15	TRUE	FALSE	FALSE	FALSE	TRUE
16	TRUE	FALSE	FALSE	FALSE	FALSE
17	FALSE	TRUE	TRUE	TRUE	TRUE
18	FALSE	TRUE	TRUE	TRUE	FALSE
19	FALSE	TRUE	TRUE	FALSE	TRUE
20	FALSE	TRUE	TRUE	FALSE	FALSE
21	FALSE	TRUE	FALSE	TRUE	TRUE
22	FALSE	TRUE	FALSE	TRUE	FALSE
23	FALSE	TRUE	FALSE	FALSE	TRUE
24	FALSE	TRUE	FALSE	FALSE	FALSE
25	FALSE	FALSE	TRUE	TRUE	TRUE
26	FALSE	FALSE	TRUE	TRUE	FALSE
27	FALSE	FALSE	TRUE	FALSE	TRUE
28	FALSE	FALSE	TRUE	FALSE	FALSE
29	FALSE	FALSE	FALSE	TRUE	TRUE
30	FALSE	FALSE	FALSE	TRUE	FALSE
31	FALSE	FALSE	FALSE	FALSE	TRUE
32	FALSE	FALSE	FALSE	FALSE	FALSE

Now, let's assume that we are using a compiler that does short-circuit predicate evaluation. The first example, using the same five atomic conditions, are logically connected as follows:

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

Necessary test cases, taking into consideration short circuiting, are seen in table 4-33. Note we started off with 32 different rows depicting all of the possible combinations that five atomic conditions could take, as shown in table 4-32.

Table 4–33

Test	A	B	C	D	E	Decision
1	FALSE	-	-	-	-	FALSE
2	TRUE	FALSE	-	-	-	FALSE
3	TRUE	TRUE	FALSE	FALSE	-	FALSE
4	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE
5	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE
6	TRUE	TRUE	TRUE	-	-	TRUE

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 are all 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* || *B*) && (*C* || *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 4-32. 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 since the short circuiting normally goes from left to right.

This predicate results in a different number of tests, as seen in table 4-34.

Table 4–34

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

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 calculation in the 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 expression outside the parentheses is 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* since the sub-expression inside the parentheses evaluates to FALSE. Tests 8 and 9 must evaluate to the end since 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 sub-expression 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).

4.3.1.11 Control-Flow Exercise

Following is a snippet of Delphi code that Jamie wrote for a test management tool. This code controls 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 the code snippet in figure 4-41:

1. Determine the total number of tests needed for multiple condition coverage.
2. If the compiler is set to short-circuit, which of those tests are actually needed?
3. Determine the tests required for decision/condition (condition determination) coverage.

```
// 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 DDShiftPressed)
      AND (DDSourceNode.Parent = DDTgtNodeParent)
      AND (NOT DoingDuplicate)) Then Begin
   Raise TstBudExcept.Create("You may not copy test to its own parent.')
End;
```

Figure 4–41

The answer is shown in the following section.

4.3.1.12 Control-Flow Exercise Debrief

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

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 DDShiftPressed)

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 F)) then***

where

> ***A = DDSourceNode.StringData2 = 'Test'***
>
> ***B = DDTgtNode.StringData2 = 'Test'***
>
> ***C = DDCtrlPressed***
>
> ***D = DDShiftPressed***
>
> ***E = DDSourceNode.Parent = DDTgtNode.Parent***
>
> ***F = DoingDuplicate***

With six 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 4-35 contains the truth tables to cover the six variables; hence 64 tests would be needed for multiple condition coverage.

Table 4–35

	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	T	T
B	T	T	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
C	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
D	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F	T	T	T	T	F	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	T	T	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	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	F	F
B	T	T	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
C	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
D	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F	T	T	T	T	F	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	T	T	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	T	F	T	F

2. 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 (i.e., the compiler generates code to evaluate from left to right subject to the general rules of scoping and parentheses, and as soon as the outcome is determined, it stops evaluating).

Test cases:

1. A = FALSE, all others don't matter, resolves to FALSE. With A FALSE, it does not matter what any of the other atomic conditions are; the whole condition evaluates to FALSE. Tests 33–64 are all covered by this one test.
2. A = TRUE, B FALSE, others don't matter, resolves to FALSE. With A TRUE and B FALSE, it does not matter what the other atomic conditions are; the whole condition evaluates to FALSE. Tests 17–32 are covered by this one test.
3. A,B = TRUE, C,D = FALSE, all others don't matter, resolves to FALSE. With A and B TRUE and both C and D FALSE, the remaining atomic conditions will not be evaluated; the entire condition evaluates to FALSE. This test covers tests 13–16.
4. A,B = TRUE, one or both C,D = TRUE, E = FALSE, F does not matter, resolves to FALSE. With A, B = TRUE and either one or both of C,D held TRUE, if E is FALSE, it does not matter what value F takes; the entire condition evaluates to FALSE. This test covers tests 3, 4, 7, 8, 11, and 12.

All the rest of the tests (1, 2, 5, 6, 9, and 10) are singletons that must be tested because the deciding factor in each is the value of F, the last atomic condition. That is, A, B, C, D, and E together all evaluate to TRUE. They are ANDed to (NOT F) to resolve the full expression, meaning the value of F determines the final result. In these cases, when F is TRUE, the expression is FALSE and vice versa.

5. A,B,C,D,E,F = TRUE resolves to FALSE (1)
6. A,B,C,D,E = TRUE, F = FALSE resolves to TRUE (2)
7. A,B,C,E,F = TRUE, D = FALSE resolves to FALSE (5)
8. A,B,C,E = TRUE, D,F = FALSE resolves to TRUE (6)
9. A,B,D,E,F = TRUE, C = FALSE resolves to FALSE (9)
10. A,B,D,E = TRUE, C,F = FALSE resolves to TRUE (10)

Therefore, we would actually need to run 10 test cases to achieve multiple condition coverage when the compiler short circuits.

3. Determine the tests required for decision/condition (condition determination) coverage.

The final portion of this exercise is to determine test cases needed for decision/condition (condition determination) testing.

Remember from the earlier section that we need two separate rules to be satisfied:

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

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

Table 4–36

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

Tests 1 and 2 in table 4-36 evaluate each of the atomics both ways, and test 3 gives us decision coverage. Note that this does not seem like much testing. We might want to go from here to MC/DC coverage. While this was not required in the exercise, it is good practice.

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 condition must affect the outcome independently when the other conditions are held steady.

ISTQB Glossary

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

Table 4–37

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

Test 1: All values are set to TRUE with an output of FALSE.

Test 2: F affects output (if it moved to TRUE, it would change output).

Test 3: A affects output (if it moved to TRUE, it would change output).

Test 4: B affects output (if it moved to TRUE, it would change output).

Tests 5 and 6:
C and D affect output (if either one moved to TRUE while the other held, it would change output).

Test 7: E affects output (if it moved TRUE, it would change output).

4.3.2 Path Testing

We have looked at many different control-flow coverage schemes. But there are other ways to design test cases using the structure as a test basis. In this section, we will look at path testing. Remember that we have covered three main different ways of approaching test design so far:

1. Statement testing, where the statements themselves drove the coverage.
2. Decision testing, where the branches drove the coverage.
3. Condition, decision/condition, modified condition/decision coverage, and multiple condition coverage, all of which looked at sub-expressions and atomic conditions of a particular decision.

ISTQB Glossary

dd-path: A path of execution (usually through a graph representing a program, such as a flow chart) that does not include any conditional nodes such as the path of execution between two decisions.

LCSAJ: Linear Code Sequence and Jump, consists of the following three items (conventionally identified by line numbers in a source code listing): the start of the linear sequence of executable statements, the end of the linear sequence, and the target line to which control-flow is transferred at the end of the linear sequence.

Now we are going to look at path testing. Rather than concentrating 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 same tests we got through control-flow testing. On the other hand, we might come up with some other interesting tests.

When we discuss path testing, we can start out with a bad idea. Brute force testing: Test every independent path through the system. Got infinity? That's how long it would take for any non-trivial system. Loops are the killers here. Every loop is a potential black hole where each different iteration through the loop creates more paths. We would need infinite time and infinite resources. Well, it was an idea...

Perhaps we can subset the infinite loops to come up with a smaller number of possible test cases that are interesting. There are a couple ways of doing this.

4.3.2.1 LCSAJ

First, let's start with Linear Code Sequence and Jump testing. Since that is such a mouthful, we will simply call it LCSAJ testing. LCSAJs are small blocks of code that fit a particular profile. There are three artifacts that constitute an LCSAJ, each one a number that represents a line of code.

First, there is the starting line. There are two ways an LCSAJ can start: either at the beginning of a module or at a line that was jumped to by another LCSAJ.

Second, there is the end of the LCSAJ. There are two ways one can end: either at the end of a module or at a line of code where a jump occurs. A jump means that we do not move to the next sequential line of code but rather load a different address and jump to it, beginning execution there.

Third, there is the target of the jump. This is also a line number.

ISTQB, in the Advanced syllabus, has deemed that LCSAJ is synonymous to DD-Path testing (where DD stands for decision to decision).[10]

The LCSAJ analysis method was devised by Professor Michael Hennell of the University of Liverpool so he could perform quality assessments on mathematical libraries for nuclear physics in 1976. The value of using LCSAJ will be discussed in the upcoming section; for now, we can consider it a relatively high overhead methodology for testing.

Formally, we say that software modules are made up of linear sequences of code which are jumped to, executed, and then jumped from. Our earlier definition, small blocks of code, matches this formal definition well. Building tests from LCSAJs starts with first identifying them and then creating data to force their traversal.

Our coverage measurement should look familiar; it is the number of LCSAJs that are executed divided by the total number that exist. This metric is indeed interesting and we will discuss it later.

Finally, the bug hypothesis is pretty much what you might think: Bugs might hide in blocks of code that escape execution.

We are going to show an example that is small enough to be clear and large enough to be non-trivial. This example was taken from Wikipedia. Figure 4-42 contains the code and an LCSAJ table.

The algorithm for designing a test is straightforward:

1. A path starts at either the start of the module or a line that was jumped to from somewhere.
2. Execution follows a linear, sequential path until it gets to a place where it must jump.
3. Find data that force execution through that path and use it in a test case.

---

10. Not every source agrees with this analysis. Edward F. Miller was credited with devising decision-to-decision path testing as a structural testing technique in the tutorial Program Testing Techniques, held during the first Computer Software and Applications Conference (COMPSAC), 1977. Since LCSAJ coverage is a relatively minor and seldom-used code coverage metric that does not have the practical, real-world applicability of statement, branch/decision, condition decision, modified condition decision, and multiple condition coverage, it's of limited value to try to get to the root of the disagreement.

One hundred percent coverage depends on identifying all paths and then testing them. Again, we will discuss this after the example.

We have identified eight separate LCSAJs in the code in figure 4-42.

```
1. #include <stdlib.h>
2. #include <string.h>
3. #include <math.h>
4.
5. #define MAXCOLUMNS 26
6. #define MAXROW 20
7. #define MAXCOUNT 90
8. #define ITERATIONS 750
9.
10. int main (void)
11. {
12.   int count = 0, totals[MAXCOLUMNS], val = 0;
13.
14.   memset (totals, 0, MAXCOLUMNS * sizeof(int));
15.
16.   count = 0;
17.   while (count < ITERATIONS)
18.   {
19.     val = abs(rand()) % MAXCOLUMNS;
20.     totals[val] += 1;
21.     if ( totals[val] > MAXCOUNT)
22.     {
23.       totals[val] = MAXCOUNT;
24.     }
25.     count++;
26.   }
27.
28.   return (0);
29.
30. }
```

LCSAJ #	Start	Finish	Jump To
1	10	17	28
2	10	21	25
3	10	26	17
4	17	17	28
5	17	21	25
6	17	26	17
7	25	26	17
8	28	28	-1

Figure 4–42 *LCSAJ example code and table*

The first starts at the beginning of the module (line 10), executes to the *while* loop (line 17), and then jumps to the end of the *while* loop (meaning it never executed the loop). This block is a perfect example of why LCSAJs are so problematic for testing. In order for this block of code to execute this way, we would need to find a time that the *while* loop does not loop. The Boolean expression for the *while* loop (*count* < *ITERATIONS*) will always evaluate to TRUE and thence always loop given the values that they are initialized to. The variable *count* is initialized to 0 in line 16. That is always going to happen when this function is started. Likewise, *ITERATIONS* is a named constant initialized to 750, a magic number that is also invariant. Since 0 is always going to be less than 750 upon first execution, this LCSAJ can never be executed unless we force it to

through the use of a debugger. The question to be asked is, Is it worthwhile forcing a test for a condition that can literally never happen?

LCSAJ 2 is going to execute. It also starts at the beginning of the module and continues executing until it reaches line 21. That means the *while* loop fires. *Val* is set with a random number between 0 and *MAXCOLUMNS* (26), the array totals at *index* [*val*] is set to 1 (0 + 1 actually), and the *if* condition is evaluated. Since the array at the index was just set to 1, the *if* statement on this first iteration will always evaluate to FALSE, causing the jump to line 25.

LCSAJ 3 is never going to execute for the reason just given in LCSAJ 2. This block would require the system to start, go to the *while* and loop, go to the *if* statement in line 21, and evaluate TRUE. Since this is impossible given the values involved, it cannot be executed without forcing values through a debugger.

LCSAJ 4 will execute once each time through the function. It starts at line 17 and jumps from 17 to the end of the *while* loop at line 28. When does this happen? The last time we evaluate the condition (*count* < *ITERATIONS*)—that is, after the 750th time the *while* loops. At that point, count will have been incremented 750 times and the condition will determine that (750 < 750), which is FALSE. At this point the jump occurs.

LCSAJ 5 will occur a great number of times, every time the *if* condition evaluates to FALSE. We start at the *while* loop (line 17), go to the *if* statement (line 21), evaluate FALSE, and jump to the end of the *if*, line 25.

LCSAJ 6 may or may not execute, depending on the random numbers that are generated. We start at the *while* loop, go into the loop, and continue to the *if* statement. If the particular array cell pointed to by *val* has been incremented enough times, then the condition comparing that cell's value with *MAXCOUNT* may actually be TRUE. At this point, with the condition evaluating TRUE, we would continue into the *then* part of the *if*, executing line 23, continuing on to line 25, and finally looping back to 17 from line 26.

LCSAJ 7 will occur every time we are in the loop and the *if* statement at line 21 evaluates to FALSE. The jump lands us at line 25, we increment *count* and then loop back to 17, the *while* evaluation.

And finally, the last LCSAJ, 8, will occur after the *while* loop finally ends. We jump to 28 to start the block, execute line 28 (the body of the block) and then end the function.

Make sure you understand each of these blocks before continuing on.

There are a number of issues that we must discuss when trying to use LCSAJs. As shown, a number of LCSAJs are really not possible to execute—there is no known way of testing them without radically changing initializations or forcing the code by setting values with a debugger. That means we can almost never get to 100 percent coverage. There are those who insist that forcing the code to execute is essential for good testing, even when it would appear to be nonsensical. We wonder whether the time needed to execute the "impossible" might not be better spent executing other test cases.

A paper detailing LCSAJ testing states it this way:

> *"The large amount of analysis required for infeasible LCSAJs is the main reason LCSAJ coverage is not a realistically achievable test metric"*[11]

All things are relative of course. If our lives depended on this code never failing and we had unlimited resources, we might try to achieve LCSAJ coverage. We certainly don't believe that this is a reasonable technique to use on every project we work on.

There are also some pragmatic reasons to think long and hard before investing time using this particular path design methodology. Since identifying LCSAJs can only be done after the code has been delivered, we get no early testing when performing this technique. And, since small changes to the code can cause radical changes to the LCSAJs, it is a particularly brittle technique to use, which is likely to lead to very expensive maintenance issues.

The holy grail of testing would be to achieve 100 percent path coverage. Testing every single unique path would allow us to find all bugs, right? Well, maybe not. We would still have failures due to different configurations, timing and race conditions, etc. LCSAJs might lead to a partial solution by finding some defects we might otherwise have missed, but the additional cost for marginal results might be problematic.

Path coverage is likely to remain low, frustratingly so if you do the math. The standard way for determining coverage is to divide the entities tested by the full number of entities there are, in this case the number of paths tested divided by the full number of paths. Suppose we did a huge amount of testing resulting

---

11. *IPL Structure Testing.pdf*, Information Processing Ltd. (IPL.com); it is available on IPL's website.

ISTQB Glossary

basis test set: A set of test cases derived from the internal structure of a component or specification to ensure that 100% of a specified coverage criterion will be achieved.

in testing 1 million paths. Since there are likely to be an infinite number of paths, our percentage of paths tested approaches zero. Not too encouraging...

So, if full path coverage is not happening, and LCSAJ is too—shall we say—iffy, how can we get some practical usage out of path theory? Perhaps we could try to identify a practical subset of paths that would be interesting to test and give us some information we might not otherwise find.

4.3.2.2 Basis Path/Cyclomatic Complexity Testing

One suggestion is to test the structure of the code to a reasonable amount.

Thomas McCabe was instrumental is coming up with the idea of basis path testing. In December 1976, he published his paper "*A Complexity Measure*"[12] in the IEEE Transactions on Software Engineering. He theorized that any software module has a small number of unique, independent paths (excluding iterations) through it. He called these *basis paths*. The theory states that the structure of the code can be tested by executing through this small number of paths and that all of the infinity of different paths are actually just using and reusing the basis paths.

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.

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

A basis path is defined as a unique independent path through the module—with no iterations or loops allowed. The basis path set is the smallest number of basis paths that cover the structure of the code.

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* refers 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 the 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 test the set of basis paths. The complexity depends not on the size of the module, but in the number of decisions that are in it.

McCabe, in his paper, pointed out that the higher the complexity, the more likely there will be a higher bug count. Subsequent studies predominately have shown such a correlation; modules with the highest complexity tend to also contain the highest number of defects. 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 smaller, less-complex modules.

We can measure cyclomatic complexity by creating a directed control-flow graph. This works well for small modules, and we will do that next. However, in real life, tools are generally used to measure module complexity.

In our control graph, we have nodes to represent entries, exits, and decisions. Edges represent non-branching code statements that do not add to the complexity.

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

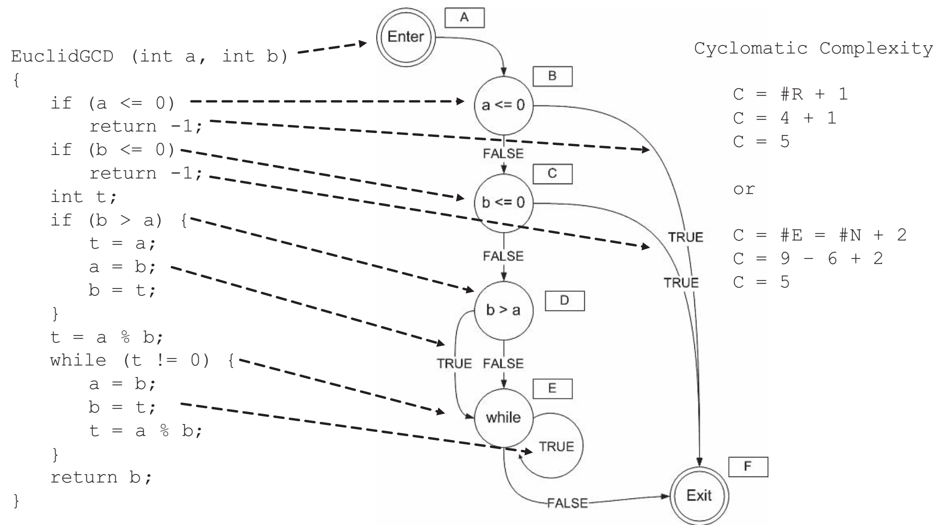


Figure 4–43 *Cyclomatic complexity example*

On the left side of figure 4-43 we have a function to calculate the greatest common divisor of two numbers using Euclid's algorithm. The dotted lines from the code to the McCabe flow diagram in the center of the figure show how non-branching sequences of zero or more statements become edges (arrows) and how branching and looping constructs become nodes (bubbles).

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.

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, simple 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 and the *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 *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.

When we introduced McCabe's theory of cyclomatic complexity a bit ago, we mentioned basis paths and basis tests. Figure 4-44 shows the basis paths and basis tests on the right side.

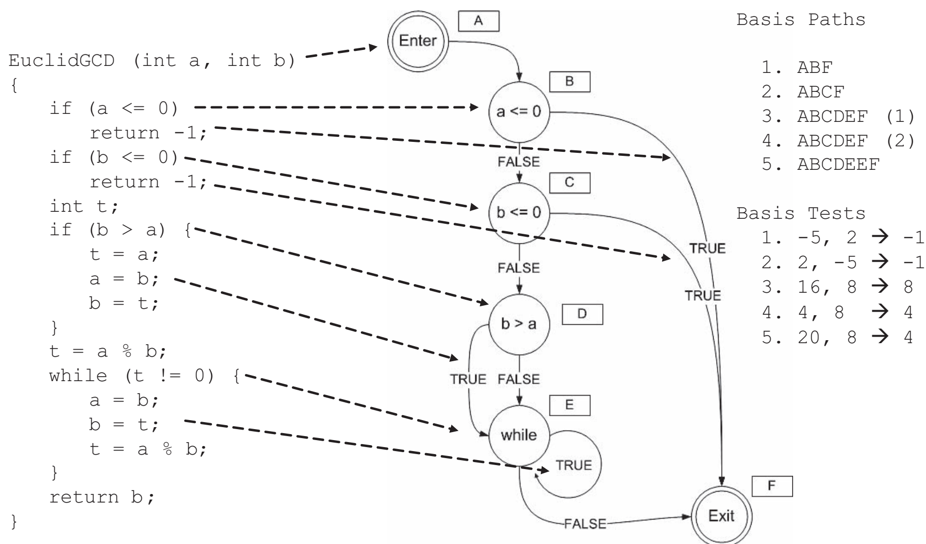


Figure 4–44 *Basis tests from directed flow graph*

The number of basis paths is equal to the Cyclomatic complexity. You construct the basis paths by starting with an arbitrary path through the diagram, from entry to exit. 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. Usually, the basis tests will correspond with the tests required to achieve branch coverage. This makes sense because 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 situa-

tion where "more than one edge from a node" represents a branching or looping construct where a decision is made.

What use is this 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. 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. That implies, as was mentioned earlier, that they did not cover the equivalence partitions.

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

$$C = E - N + P$$

whereas we list it as

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

A careful reading of the original paper does show both 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 he 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 must be added when not shown (as in our example). The P value stands for the number of connected components.[13] 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 exit to the entrance node). The mathematics of this is beyond the 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 talk about static analysis.

13. A connected component, in this context, would be a called sub-routine.

4.3.2.3 Cyclomatic Complexity Exercise

The following C code function loads an array with random values.

```
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.

4.3.2.4 Cyclomatic Complexity Exercise Debrief

The directed control-flow graph should look like figure 4-45. Note that the edges D1 and D2 are labeled; D1 is where the *if* conditional in line 14 evaluates to TRUE, D2 is where it evaluates to FALSE.

We could use three different ways to calculate the cyclomatic complexity of the code as shown in the box on the right in figure 4-45.

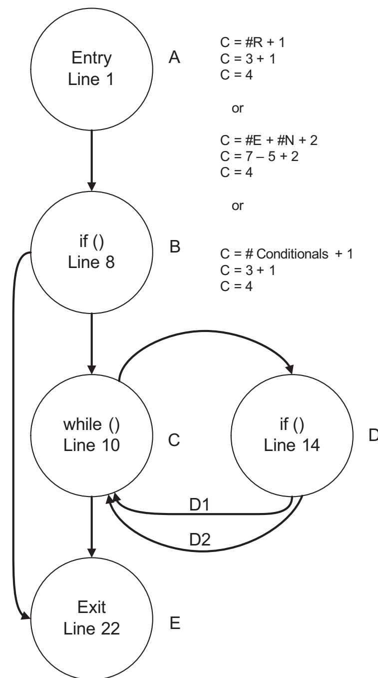


Figure 4–45 *Cyclomatic complexity exercise*

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-D and is enclosed by line D1. The third is the region with the same top edge, C-D, and is enclosed by D2. Here is the formula:

C = # 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. ABCD(D2)CE

4.3.3 A Final Word on Structural Testing

Before we leave structural testing, a final word: Boris Beizer wrote a pithy argument seemingly *against* doing white-box structure-based testing in his book *Software Testing Techniques*, first 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.

4.3.4 Structure-Based Testing Exercise

Using the code in figure 4-46, 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 DO/178B 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 4–46 *Code for structure-based exercise*

4.3.5 Structure-Based Testing Exercise Debrief

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

The directed control-flow graph would look something like figure 4-47. It looks a bit messy, but if you look closely at it, it does cover the code.

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.

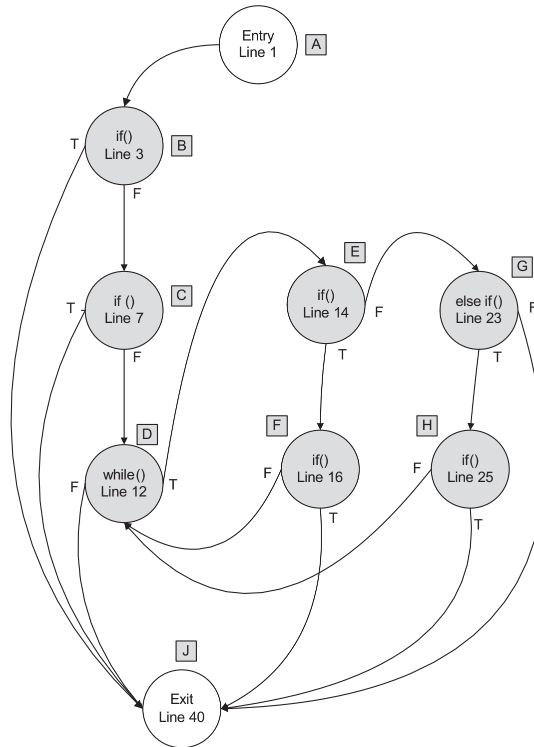


Figure 4–47 *Structure-based testing exercise directed control-flow graph*

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:

Line 3 : *if*

Line 7: *if* (Remember, the else in line 10 is not a decision.)

Line 12: *while*

Line 14: *if*

Line 16: *if*

Line 23: *else if* (This is really just an if, often called a ladder if construct.)

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. ABJ
2. ABCJ
3. ABCDEFJ
4. ABCDEFDEFJ
5. ABCDEGHJ
6. ABCDEGHDEGHJ
7. ABCDEGJ
8. ABCDJ

Notice that test 8 is a bit of a problem. If you look closely at the code, it cannot happen. Famous last words for a tester, right? It should not ever happen. We would have to use a debugger to test it by changing the value of *branch* to 0.

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 4-38. 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 4–38

	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	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	F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
C	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F	T	T	T	T	F	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	T	T	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	T	F	T	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).

While the exercise does not require us to figure out the number of tests required in case of short circuiting, it is an interesting question. In table 4-39 are the test cases we would need for the first *if()* statement (line 3) to achieve coverage.

Table 4–39

	A	B	C	D	E	Expression Eval
1.	FALSE	-	-	-	-	FALSE
2.	TRUE	FALSE	-	-	-	FALSE
3.	TRUE	TRUE	TRUE	-	-	TRUE
4.	TRUE	TRUE	FALSE	FALSE	-	FALSE
5.	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE
6.	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE

Test 1: If the first term goes FALSE, the other terms do not affect anything.

Test 2: If the second term goes FALSE while the first term is TRUE, the others do not matter.

Test 3: If the third term goes TRUE when A and B are TRUE, C and D do not matter.

Test 4: When A and B are TRUE, if C and D are FALSE, E does not matter.

Test 5: No short circuit is possible.

Test 6: No short circuit is possible.

Only two of these test cases evaluate to TRUE. Remember, this value gets negated, so only these two tests would survive to continue on to line 7. That will not be enough testing to get decision coverage throughout the rest of the code. We will need to add test cases to achieve that level; likely we would want to test some other combinations of atomic conditions just for kicks.

Test 1: Shown in table 4-39 as test 1. [***ABJ***]

Test 2: Shown in table 4-39 as test 2. [***ABJ***]

Test 3: Shown in table 4-39 as test 4. [***ABJ***]

Test 4: Shown in table 4-39 as test 6. [***ABJ***]

Test 5: Assume that the values from test 3 also have (*root* == 0) so the test ends immediately after creating a new *ObjNode* in line 9. [***ABCJ***]

Test 6: Assume that the values from test 5 are included when *root* != 0. Further assume that the object is new and belongs on the left side (*w* < *CurrentObj*). After creating a new *ObjNode* and populating it, test ends. [***ABCDEFJ***]

Test 7: Assume that you pick a set of values for the first *if* (line 3) that causes the expression to trigger FALSE. *Root* != 0. Obj lies to the left and down one level and is not currently in the tree. [***ABCDEFDEFJ***]

Test 8: Assume that the values from test 5 are included when *root* != 0. Further assume that the object is new and belongs on the right side (*w* > *CurrentObj*). After creating a new *ObjNode* and populating it, test ends. [***ABCDEGHJ***]

Test 9: Assume that you pick a set of values for the first *if* (line 3) that causes the expression to trigger FALSE. *Root* != 0. Obj lies to the right and down one level and is not currently in the tree. [***ABCDEGHDEGHJ***]

Test 10: Assume that you pick a set of values for the first *if* (line 3) that causes the expression to trigger FALSE. *Root* != 0. Object is the first object tested and so the counter is incremented. [***ABCDEGJ***]

Note: With 10 tests, we do achieve multiple-condition-level coverage (assuming that the compiler writes code to short-circuit the testing); however, we do not have loop coverage for the *while()* on line12 because it cannot execute 0 times. At the minimum, we would also want to test the loop a lot of times (i.e., a very large binary tree where the object does not already exist).

3. If this code were in a system that was subject to FAA DO/178B 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 (see table 4-40).

Table 4–40

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.

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 that each atomic condition can affect the outcome independently.

Table 4–41

	A	B	C	D	E	Expression Eval
1.	F	T	T	T	T	FALSE
2.	T	F	T	T	T	FALSE
3.	T	T	F	F	T	FALSE
4.	T	T	F	T	F	FALSE
5.	T	T	F	T	T	TRUE

Test 1: With these values, A controls the output independently. If it went TRUE, the output would toggle.

Test 2: With these values, B controls the output independently. If it went TRUE, the output would toggle.

Test 3: With these values, C controls the output independently. If it goes TRUE, the output would toggle.

Test 4: With these values, E controls the output independently. If it goes TRUE, the output would toggle.

Test 5: With these values, D controls the output independently. If it goes FALSE, the output would toggle.

With these five tests, we can meet all three objectives (for the first *if()* statement). Among the five tests, each atomic condition is tested both ways. Decision coverage is achieved by test 5 and any other single test. And in each case, as shown, each atomic condition can affect the output independently.

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 it's TRUE, it does a quick return, which must be executed. When it's FALSE, we get to go further in the code.

The second *if()* (line 7) and 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.

4.4 Defect- and Experience-Based

Learning objectives

(K2) Describe the principle and reasons for defect-based techniques and differentiate its use from specification- and structure-based techniques. (K2) Explain by examples defect taxonomies and their use. (K2) Understand the principle of and reasons for experience-based techniques and when to use them.

(K3) Specify, execute, and report tests using exploratory testing.

(K3) Specify tests using the different types of software fault attacks according to the defects they target.

(K4) Analyze a system in order to determine which specification-based, defect-based, or experience-based techniques to apply for specific goals.

Now we're going to move from systematic techniques for test design into less-structured but nonetheless useful techniques. We start with defect-based and defect-taxonomy-based techniques.

Conceptually, we are doing defect-based testing anytime the type of the defect sought is the basis for the test. Usually, the underlying model is some list

ISTQB Glossary

defect-based technique: A procedure to derive and/or select test cases targeted at one or more defect categories, with tests being developed from what is known about the specific defect category.

experience-based technique: Procedure to derive and/or select test cases based on the tester s experience, knowledge and intuition.

of defects seen in the past. If this list is organized as a hierarchical taxonomy, then the testing is defect taxonomy based.

To derive tests from the defect list or the defect taxonomy, we create tests designed to reveal the presence of the defects in the list.

Now, for defect-based tests, we tend to be more relaxed about the concept of coverage. The general criterion is that we will create a test for each defect type, but it is often the case that the question of whether to create a test at all is risk weighted. In other words, if the likelihood or impact of the defect doesn't justify the effort, don't do it. However, if the likelihood or impact of the defect were high, then you would create not just one test, but perhaps many. This should be starting to sound familiar to you, yes?

The underlying bug hypothesis is that programmers tend to repeatedly make the same mistakes. In other words, a team of programmers will introduce roughly the same types of bugs in roughly the same proportion from one project to the next. This allows us to allocate test design and execution effort based on the likelihood and impact of the bugs.

4.4.1 Defect Taxonomies

Table 4-42 shows an example of a defect taxonomy. It occurs in Rex's book *Managing the Testing Process* but was originally derived from Boris Beizer's taxonomy in Software Testing Techniques.

ISTQB Glossary

defect taxonomy: A system of (hierarchical) categories designed to be a useful aid for reproducibly classifying defects.

Table 4–42

• Functional – Specification – Function – Test • System – Internal Interface – Hardware Devices – Operating System – Software Architecture – Resource Management • Process – Arithmetic – Initialization – Control or Sequence – Static Logic – Other	• Data – Type – Structure – Initial Value – Other • Code • Documentation • Standards • Other • Duplicate • Not a Problem • Bad Unit • Root Cause Needed • Unknown

There are eight main categories along with five "bookkeeper" categories that are useful when classifying bugs in a bug tracking system. Let's go through these in detail.

In the category of Functional defects, there are three subcategories:

- **Specification**: The functional specification—perhaps in the requirements document or in some other document—is wrong.
- **Function**: The specification is right, but the implementation of it is wrong.
- **Test**: Upon close research, we found a problem in test data, test designs, test specifications, or somewhere else.

In the category of System defects, there are five subcategories:

- **Internal Interface**: The internal system communication fails; in other words, there is an integration problem of some sort internal to the test object.

- **Hardware Devices**: The hardware that is part of the system or that hosts the system fails.
- **Operating System**: The operating system—which presumably is external to the test object—fails.
- **Software Architecture**: Some fundamental design assumption proves invalid, such as an assumption that data could be moved from one table to another or across some network in some constant period.
- **Resource Management**: The design assumptions were okay, but some implementation of the assumption was wrong; for example, the design of the data tables introduces delays.

In the category of Process defects, there are five subcategories:

- **Arithmetic**: The software does math wrong. This doesn't mean just basic math; it can also include sophisticated accounting or numerical analysis functions, including problems that occur due to rounding and precision issues.
- **Initialization**: An operation fails on its first use, when there are no data in a list, and so forth.
- **Control or Sequence**: An action occurs at the wrong time or for the wrong reason, like say, seeing screens or fields in the wrong order.
- **Static Logic**: Boundaries are misdefined, equivalence classes don't include the right members and exclude the wrong members, and so forth.
- **Other**: A control-flow or processing error that doesn't fit in the preceding categories has occurred.

In the category of Data defects, there are four subcategories:

- **Type**: The wrong data type—whether a built-in or user-defined data type—is used.
- **Structure**: A complex data structure or type is invalid or inappropriately used.
- **Initial Value**: A data element's initialized value is incorrect, like a list of quantities to purchase that defaults to zero rather than one.
- **Other**: A data-related error occurs that doesn't fit in the preceding buckets.

The category of Code applies to some simple typo, misspelling, stylistic error, or other coding error occurring and resulting in a failure. Theoretically, these

couldn't get past a compiler, but in these days of scripting on browsers, this stuff does happen.

The category of Documentation applies to situations where the documentation says the system does X on condition Y, but the system does Z—a valid and correct action—instead.

The category of Standards applies to situations where the system fails to meet industry, governmental, or vendor standards or regulations or to follow coding or user interface standards or to adhere to naming conventions, and so forth.

Other: The root cause is known, but fits none of the preceding categories, which should be rare if this is a useful taxonomy.

The five housekeeping categories are as follows:

- Duplicate: You find that two bug reports describe the same bug.
- Not a Problem: The behavior noted is correct. The report arose from a misunderstanding on the part of the tester about correct behavior. This situation is different than a test failure because this occurs during test execution and is an issue of interpretation of an actual result.
- Bad Unit: The bug is a real problem, but it arises from a random hardware failure that is unlikely in the production environment.
- Root Cause Needed: Applies when the bug is confirmed as closed by test but no one in development has supplied a root cause.
- Unknown: No one knows what is broken. Ideally, this applies to a small number of reports, generally when an intermittent bug doesn't appear for quite awhile, leading to a conclusion that some other change fixed the bug as a side effect.

Notice that this taxonomy is focused on root cause, at least in the sense of what ultimately proved to be wrong in the code that caused the failure.

You could also have a "process root cause" taxonomy that showed at which phase in the development process that bug was introduced. Notice that such a taxonomy would be useless to us for designing tests. Even the root-cause-focused taxonomy can be a bit hard to use for purposes of test design.

One concern that we always have when dealing with taxonomies: the categories that say *other*. Like a backyard swimming pool without a fence, these are

trouble waiting to happen. We have seen cases where an analysis of a taxonomy showed more than 80 percent of the issues to be set to *other*.

For a taxonomy to work, the users must be educated to think through each issue thoroughly before assigning it to the *other* bucket. When Jamie wrote a test management tool, he would allow the *other* category to be chosen only when an associated field was filled with an explanation of why the issue did not fit other categories. If there is no training, education, and mentoring on the use and importance of categorizing issues correctly in a taxonomy, a taxonomy can degrade to a simple waste of time.

Table 4-43 shows an example of a bug taxonomy—a rather coarse-grained one—gathered from the Internet appliance case study we've looked at a few times in this book. This one is focused on symptoms. Notice that this makes it easier to think about the tests we might design. However, its coarse-grained nature means that we'll need to have additional information—lists of desired functional areas, usability and user interface standards and requirements, reliability specifications, and the like—to use this to design tests.

We've also showed the percentage of bugs we actually found in each category during system test.

Table 4–43

Description	**Count**	**%**
Failed functionality	425	46%
Missing functionality	179	19%
Poor usability	106	11%
Build failed	62	7%
Bad system design/architecture	51	5%
Reliability problem	49	5%
Data loss	18	2%
Slow performance	16	2%
Code obsolete	16	2%
Deviation from specification	8	1%
Bad user documentation	2	0%
Total	932	100%

ISTQB Glossary

error guessing: A test design technique where the experience of the tester is used to anticipate what defects might be present in the component or system under test as a result of errors made and to design tests specifically to expose them.

4.4.2 Error Guessing

Error guessing is a term introduced by Glenford Myers.[14]

Conceptually, error guessing involves the tester taking guesses about a mistake that a programmer might make and then developing tests for it. Notice that this is what might be a called a "gray-box" test since it requires the tester to have some idea about typical programming mistakes, how those mistakes become bugs, how those bugs manifest themselves as failures, and how we can force failures to happen.

Now, if error guessing follows an organized hierarchical taxonomy—like defect-taxonomy-based tests, then the taxonomy is the model. The taxonomy also provides the coverage criterion, if it is used, because we again test to the extent appropriate for the various elements of the taxonomy. Usually, the error guessing follows mostly from tester inspiration.

As you can tell so far, the derivation of tests is based on the tester's intuition and knowledge about how errors (the programmer's or designer's mistake) become defects that manifest themselves as failures if you poke and prod the system in the right way. Now, error guessing tests can be part of an analytical, predesigned, scripted test procedure. However, they often aren't, but are added to the scripts (ideally) or used instead of scripts (less ideal) at execution time.

The underlying bug hypothesis is very similar to that of defect-taxonomy-based tests. Here, though, we not only count on the programmer to make the same mistakes, we also count on the tester to have seen bugs like the ones in this system before and to remember how to find them.

At this point, you're probably thinking back to our earlier discussion about quality risk analysis. Some of these concepts sound very familiar, don't they?

---

14. You can find this technique and others we have discussed in the first book on software testing, *The Art of Software Testing* by Glenford Myers.

Certainly, error guessing is something that we do *during* quality risk analysis. It's an important part of determining the likelihood. Perhaps we as test analysts don't do it, but rather we rely on the developers to do it for us. Or maybe we participate with them. Depends on our experience, which is why error guessing is an experience-based test technique.

Maybe a bug taxonomy is a form of risk analysis? It has elements of a quality risk analysis, in that it has a list of potential failures and frequency ratings. However, it doesn't always include impact.

Even if a bug taxonomy does include impact ratings for potential bugs, it's still not quite right. Remember that the way we described it, a quality risk analysis is organized around quality risk categories or quality characteristics. A bug taxonomy is organized around categories of failures or root causes rather than quality risk categories or quality.

4.4.3 Checklist Testing

With checklist testing, we now get into an area that is often very much like quality risk analysis in its structure. Conceptually, the tester takes a high-level list of items to be noted, checked, or remembered. What is important for the system to do? What is important for it not to do?

The checklist is the model for testing, and the checklist is usually organized around a theme. The theme can be quality characteristics, user interface standards, key operations, or any other theme you'd like to pick. To derive tests from a checklist, either during test execution time or during test design, you create one to evaluate the system per the test objectives of the checklist.

Now, we can specify a coverage criterion, which is that there be at least one test per checklist item. However, the checklist items are often very high-level items. By very high-level, we don't mean high-level test case. A high-level or logical test case gives rules about how to generate the specific inputs and expected results but doesn't necessarily give the exact values as a low-level or concrete test case would.

The level of a test checklist is even higher than that. It will give you a list of areas to test, characteristics to evaluate, general rules for determining test pass or failure, and the like. You can develop and execute one set of tests, while another competent tester might cover the same checklist differently. Notice that

this is not true for many of the specification-based techniques we covered, where there was one right answer.

The underlying bug hypothesis in checklist testing is that bugs in the areas of the checklist are either likely, important, or both. Notice again that much of this harkens back to risk-based testing. But note that, in some cases, the checklist is predetermined rather than developed by an analysis of the system. As such, checklist testing is often the dominant technique when the testing follows a methodical test strategy. When testing follows an analytical test strategy, the list of test conditions is not a static checklist but rather is generated at the beginning of the project and periodically refreshed during the project, through some sort of analysis, such as quality risk analysis.

Table 4-44 is an example of a checklist, this one for reviewing code[15]. We'll see more of these types of checklists in chapter 6 when we discuss reviews.

Table 4–44

- Declarations
  - Are literal constants correct?
  - Are all variables always initialized, especially in changed code?
  - Are unsigned integers used when signed should be?
- Data Items
  - Are all strings null-terminated—especially after manipulation?
  - Are buffer size checks always done when manipulating them?
  - Are bitfield manipulations portable to other architectures?
  - Is *sizeof()* called with a pointer rather than the object pointed to?

Under declarations we ask the following questions:

- Are all the literal constants correct? Are we using magic numbers that might change in the future? Are we using them in multiple places? Should we change those to declared constant values?
- Have we ensured that every single variable has been initialized, especially when we are in maintenance mode and have added more or changed existing variables.
- Are our data types correct? Are our variables big enough? Might there be a good reason to use signed over unsigned (or vice versa)?

15. This particular checklist comes from Brian Marick's book *The Craft of Software Testing.*

ISTQB Glossary

exploratory testing: An informal test design technique where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests.

Under data items, we might ask these questions:

- Are all strings handled correctly? If explicit buffers are used for them, are they big enough? If null-terminated, do we manipulate them directly such that the null-value is misplaced or lost?
- Is every buffer usage checked for overrun? The hackers and crackers will be checking your buffers. Did you?
- When dealing with bitfields, is the code going to be portable? Does word size matter when manipulating the bits? Will they work in both big-endian and small-endian systems?
- Have you used *sizeof()* correctly? If a pointer is passed in rather than a reference, this function is going to return the size of the pointer.

Okay, so how do we use this for testing? Well, imagine going through the code, methodically, checking off each of these main areas. If you see problems, you report a bug against this heuristic for the module where you saw the problem. Notice the subjectivity involved in the evaluation of test pass or fail results.

4.4.4 Exploratory Testing

Exploratory testing, like checklist testing, often follows some basic guidelines, like a checklist, and often relies on tester judgment and experience to evaluate the test results. However, exploratory testing is inherently more reactive, more dynamic, in that most of the action with exploratory testing has to happen during test execution.

Conceptually, exploratory testing is happening when a tester is simultaneously learning the system, designing tests, and then executing those tests. The results of each test, largely, determine what we test next.

Now, that is not to say that this testing is random or driven entirely by impulse or instinct. For example, we could use a list of usability heuristics not only as a preplanned checklist, but also as a heuristic to guide us to important or

problematic software areas. The best kinds of exploratory testing usually do have some type of model, either written or mental.

According to this model, we derive tests by thinking how best to explore some area of testing interest. In some cases, to keep focus and provide some amount of structure, the test areas to be covered are constrained by a test charter.

As with checklist testing, we can specify a coverage criterion, which is that there be at least one test per charter (if we have them). However, the charters, like checklist items, are often very high level.

The underlying bug hypothesis is that the system will reveal buggy areas during test execution that would be hidden during test basis analysis. In other words, you can only learn so much about how a system behaves and misbehaves by reading about it. This is, of course, true. It also suggests an important point, which is that exploratory testing, and indeed experience-based tests in general, make a good blend with scripted tests because they offset each other's weak spots.

The focus of testing in the ISTQB Foundation and Advanced syllabi is primarily analytical; that is, following an analytical test strategy. Analytical strategies are good at defect prevention, risk mitigation, and structured coverage.

Experience-based tests usually follow a dynamic or reactive test strategy. One way that analytical and dynamic strategies differ is in the process of testing.

The exploratory testing process, unlike the ISTQB Fundamental testing process, is very much focused on taking the system as we find it and going from there. The tester simultaneously learns about the product and its defects, plans the testing work to be done (or, if she has charters, adjusts the plan), designs and executes the tests, and reports the results.

Now, when doing exploratory testing, it's important not to degrade into frenzied, aimless keyboard-pounding. Good exploratory tests are planned, interactive, and creative.

As we mentioned, the test strategy is dynamic, and one manifestation of this is that the tester dynamically adjusts test goals during execution.

ISTQB Glossary

test charter: A statement of test objectives, and possibly test ideas about how to test. Test charters are used in exploratory testing.

4.4.4.1 Test Charters

Because there is so much activity happening during exploratory test execution—activity that, in an analytical strategy, happens before execution—as you can imagine some trade-offs must occur. One of those is in the extent of documentation of what was tested, how that related to the test basis, and what the results were.

The inability to say what was tested during exploratory testing, to any degree of accuracy, has long been seen as its Achilles heel. The whole confidence-building objective of testing is undermined if we can't say what we've tested and how much we've tested it as well as being able to say what we haven't yet tested. Notice that analytical test strategies do a good job of this, at least when traceability is present.

Another issue that comes up anytime we have multiple test analysts working concurrently on the same test object is redundancy; that is, to what extent multiple analysts are testing the exact same test items.

One way people have come up with to reduce these problems with exploratory testing is to use test charters. A test charter specifies the tasks, objectives, and deliverables, but in very few words. The charters can be developed well in advance—even, in fact, based on analysis of risks as we have done for some clients—or they can be developed just immediately before test execution starts and then continually adjusted based on the test results.

Right before test execution starts for a given period of testing, exploratory testing sessions are planned around the charters. These plans are not formally written following IEEE 829 or anything like that. In fact, they might not be written at all. However, consensus should exist between test analyst and test manager about the following:

- What the test session is to achieve
- Where the test session will focus
- What is in and out of scope for the session

- What resources should be used, including how long it should last (often called a timebox)

Now, given how lightweight the charters are—as you'll see in a moment—you might expect that the tester would get more information. And, indeed, the charters can be augmented with defect taxonomies, checklists, quality risk analyses, requirements specifications, user manuals, and whatever else might be of use. However, these augmentations are to be used as reference during test execution rather than as objects of analysis—i.e., as a test basis—prior to the test execution starting.

Exploratory Testing Session Log

Tester name ____________________ Date _______________
Time on-task ____________________ Charter completed ________________

Charter

Test the security of the login page. See if it is possible to log in without a password.

Bugs reported

937 - Log in form vulnerable to SQL injection.
939 - System identifies a valid user name when the password is wrong.

Follow-up issues

* Lockout feature on three unsuccessful login attempts does not seem to work.

Figure 4–48

In figure 4-48 we see an example of how this works. This replicates an actual exploratory testing session log from a project we did for a client recently. We were in charge of running an acceptance test for the development organization on behalf of the customers. If that sounds like a weird arrangement, by the way, yes, it was.

At the top of the log, we have captured the tester's name and when the test was run. It also includes a log of how much time was spent and whether the tester believes that the charter was completely explored—remember, this is subjective, as with the checklists.

Below the heading information you see the charter. Now, it's not unusual to have a test procedure called "test login security," or even a sentence like this in the description of a test procedure or listed as a condition in a test design specification. However, understand that *this is all there is*. There are no further details specified in the test for the tester. You see how we are relying very heavily on the tester's knowledge and experience?

Under the charter section are two sections that indicate results. The first lists the bugs that were found. The numbers correspond to numbers in a bug tracking system. The second lists issues that need follow-up. These can be—as they are here—things that might be bugs but that we didn't have time to finish researching. Or, it could indicate situations where testing was incomplete due to blockages. In that case, of course, we'd expect that the charter would not be complete.

4.4.4.2 Exploratory Testing Exercise

Consider the use of exploratory testing on the HELLOCARMS project.

The exercise consists of four parts:

1. Identify a list of test charters for testing an area.
2. Document your assumption about the testers who will use the charters.
3. Assign a priority to each charter and explain why.
4. Explain how you would report results.

As always, check your work on the preceding part before proceeding to the next part. The solutions are shown in the next section.

4.4.4.3 Exploratory Testing Exercise Debrief

We selected requirements element 010-010-170, which has to do with allowing applications over the Internet.

> *Support the submission of applications via the Internet, which includes the capability of untrained users to properly enter applications.*

We read that to mean we should use a mix of PC configurations, security settings, connection speeds, customer personas, and existing customer relationships to test applications over the Internet.

Let's see what that might look like.

First, we would use pairwise techniques[16] to generate a set of target PC configurations, including browser brand and version, operating system, and connection speeds. We would build these configurations prior to test execution, storing drive images for a quick restore during testing.

Next, we would create a list of customer personas. Personas refers to the habits that a customer exhibits and experience that a customer has:

- Nervous customer: Uses Back button a lot, revises entries, has long think time
- Novice Internet user: Makes a lot of data input mistakes, has long think time
- Power user: Types quickly, makes few mistakes, uses copy-and-paste from other PC applications (e.g., account numbers), has very short think time
- Impatient customer: Types quickly, makes many mistakes, has very short think time, hits Next button multiple times

Now we would create a list of existing customer banking relationship types:

- Limited accounts, none with Globobank
- Limited accounts, some with Globobank
- Limited accounts, all with Globobank
- Extensive accounts, none with Globobank
- Extensive accounts, some with Globobank
- Extensive accounts, all with Globobank

Notice that these two lists allow a lot of tester latitude and discretion. Notice also that for the existing customer banking relationships, as with the PC configurations, it would again make a lot of sense for the tester to create this customer data before test execution started.

We would create some semiformal rules to cover testing with the test charters, as shown in table 4-45.

16. These are covered in *Advanced Software Testing Vol. 1* and are required for the ISTQB Advanced Test Analyst certification, but not for the ISTQB Advanced Technical Test Analyst certification.

Table 4–45

General rules for test charters:
• For each of the following charters, restore your test PC to a previously untested PC configuration prior to starting the charter. Make sure each configuration is tested at least once.
• For each of the following charters, select a persona. Make sure each persona is tested at least once.
• For each of the following charters, select an existing customer banking relationship type. Make sure each customer banking relationship type is tested at least once.
• Allocate 30–45 minutes for each application; thus, each charter is 30–120 minutes long.

We would then write our charters as shown in table 4-46.

Table 4–46

Charters:
1. Test successful applications with both limited and extensive banking relationships, where customer declines insurance.
2. Test a successful application where customer accepts insurance.
3. Test a successful application where the system declines insurance.
4. Test a successful application where property value escalates application.
5. Test a successful application where loan amount escalates application.
6. Test an application that was unsuccessful due to credit history.
7. Test an application that was unsuccessful due to insufficient income.
8. Test an unsuccessful application due to excessive debt.
9. Test an unsuccessful application due to insufficient equity.
10. Test cancellation of an application from all possible screens (120 minutes).
11. Test a fraudulent application, where material information provided by customer does not match decisioning mainframe's data.
12. Test a fraudulent application, where material information provided by customer does not match LoDoPS data.

Yes, these charters might revisit some areas covered by our other, specification-based tests. However, because the testers will be taking side trips that wouldn't be in the scripts, the coverage of the scenarios will be broader.

Now, what is our assumption about the testers who will use these charters? Obviously, they have to be experienced testers because the test specification provided is very limited and provides a lot of discretion. They also have to understand the application well because we are not giving them any instructions on how to carry out the charters. They also understand PC technology at least well enough to restore a PC configuration from a drive image, though it

> **ISTQB Glossary**
>
> **software attacks**: Directed and focused attempt to evaluate the quality, especially reliability, of a test object by attempting to force specific failures to occur.

would be possible to give unambiguous directions to someone on how to do that.

In terms of the priority to each charter, we have listed them in priority order. Notice that we start with the simplest case, a successful application with no insurance, and then add complexity from there. Our objective is to use the exploratory tests during the scripted tests, in parallel. As the test coverage under the scripts gets greater and greater, so also the complexity of the exploratory scenarios.

As for results reporting, we would have each charter tracked as a test case in our test management system. For bug reports, though, it would be very important that the tester perform adequate isolation to determine if the configuration, the persona, the banking relationship, or the functionality itself was behind the failure.

4.4.5 Software Attacks

Finally, we come to a technique that you can think of as an interesting synthesis of all the defect- and experience-based techniques we've covered in this section, software attacks. It combines elements of defect taxonomies, checklist-based tests, error guessing, and exploratory tests.

Conceptually, you can think of a software attack as a directed and focused form of testing that attempts to force specific failures to occur. As you can see, this is like a defect taxonomy in that way.

However, it's more structured than a defect taxonomy because it is built on a fault model. The fault model talks about how bugs come to be and how and why bugs manifest themselves as failures. We'll look at this question of how bugs come to be in a second when we talk about the bug hypothesis.

This question of manifestation is very important. It's not enough to suspect or believe a bug is there. In dynamic testing, since we aren't looking at the code, we have to look at behaviors.

Now, here comes the similarity with checklists. Based on this fault model, James Whittaker and his students at Florida Tech (who originated this idea) developed a simple list of attacks that go after these faults. This hierarchical list of attacks—organized around the ways in which bugs come to be—provides ways in which we can force bugs to manifest themselves as failures.

To derive tests, you analyze how each specific attack might apply to the system you're testing. You then design specific tests for each applicable attack. The analysis, design, and assessment of adequate coverage are discretionary and dependent on the skills, intuition, and experience of the tester. The technique provides ideas on when to apply the attack, what bugs make the attack successful, how to determine if the attack has forced a bug into the open as a failure, and how to conduct the attack. However, two reasonable and experienced testers might apply the same attack differently against the same system and obtain different results.

So, from where does the fault model say bugs come? The underlying bug hypothesis is that bugs arise from interactions between the software and its environment during operation and from the capabilities the software possesses. The software's operating environment consists of the human user, the file system, the operating system, and other cohabitating and interoperating software in the same environment. The software's capabilities consist of accepting inputs, producing outputs, storing data, and performing computations.

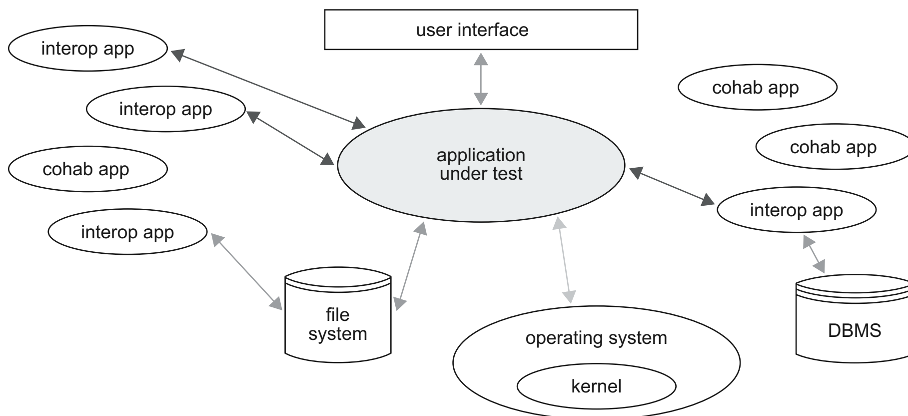


Figure 4–49 *Application in its operating environment*

In figure 4-49, you see a picture of the application under test in its operating environment. The application receives inputs from and sends outputs to its user interface. It interacts in various direct ways with interoperating applications; e.g., through copy-and-paste from one application to another or by sending data to and retrieving data from an application that in turn manages that data with a database management system.

The application also can interact indirectly by sharing data in a file with another application. Yet other applications, with which it does not interact, can potentially affect the application—and vice versa—due to the fact that they cohabit the same system, sharing memory, disk, CPU, and network resources.

The application sends data to and from the file system when it creates, updates, reads, and deletes files. It also relies on the operating system, both its libraries and the kernel, to provide various services and to intermediate interaction with the hardware.

So, how can we attack these interfaces? To start with the file system, the technique provides the following attacks:

- Fill the file system to capacity. In fact, you can test while you fill and watch the bugs start to pop up.
- Related to this is the attack of forcing storage to be busy or unavailable. This is particularly true for things that do one thing at a time, like a DVD writer.
- You can damage the storage media, either temporarily or permanently. Soil or even scratch a CD.
- Use invalid file names, especially file names with special characters.
- Change a file's access permissions, especially while it's being used or between uses.
- And, one of Rex's favorites—for reasons you'll find out in a moment—vary or corrupt file contents.

For interoperating and cohabiting software interfaces, along with the operating system interfaces, the suggestions are to try the following attacks:

- Force all possible incoming errors from the software/OS interfaces to the application.
- Exhaust resources like memory, CPU, and network resources.
- Corrupt network flows and memory stores.

All of these attacks can involve the use of tools, either homegrown, freeware, or commercial.

You'll notice that much of the technique focuses on the file system. We think this is a shame, personally. We would like to see the technique extended to be more useful for system integration testing, particularly for test types like interoperability, end-to-end tests, data quality/data integrity tests with shared databases, and the like. As it is, the technique's interface attacks seem very much focused on PC-based, stand-alone applications.

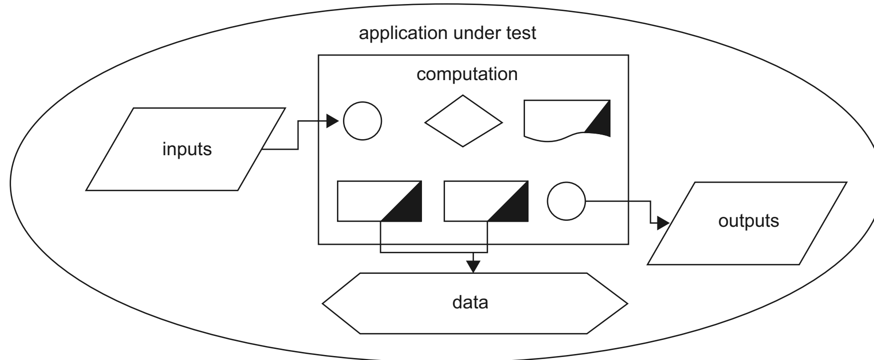


Figure 4–50 *Capabilities to attack*

In figure 4-50, you see a picture of the application under test in terms of its capabilities. The application accepts inputs. The application performs computations. Data go into and come out of data storage (perhaps being persisted to the file system or a database manager in the process, which we've not shown here). The application produces outputs.

So, how can we attack these interfaces? In terms of inputs, we can do the following:

- We can apply various inputs that will force all the error messages to occur. Notice that we'll need a list of error messages from somewhere: user's guide, requirements specification, programmers, or wherever.
- We can force the software to use, establish, or revert to default values.
- We can explore the various allowed inputs. You'll notice that equivalence partitioning and boundary value analysis are techniques we can draw upon here.
- We can overflow buffers by putting in really long inputs.

- We can look for inputs that are supposed to interact and test various combinations of their values, perhaps using techniques like equivalence partitioning and decision tables (when we understand and can analyze the interactions) or pairwise testing and classification trees[17] (when we do not understand and cannot analyze the interactions).
- We can repeat the same or similar inputs over and over again.

In terms of outputs, we can do the following:

- We can try to force outputs to be different, including for the same inputs.
- We can try to force invalid outputs to occur.
- We can force properties of outputs to change.
- We can force screen refreshes.

The problem here is that these attacks are mostly focused on stuff we can do, see, or make happen from the user interface. Now, that's fertile ground for bugs, but there are certainly other types of inputs, outputs, data, and computations we're interested in as testers. Some of that involves access to system interfaces—which is definitely the province of the technical test analyst.

4.4.5.1 An Example of Effective Attacks

As you might guess, Rex has to use PowerPoint a lot. It's an occupational hazard. If this sounds like less than a ringing endorsement, it's because PowerPoint interoperates poorly with other Office applications and suffers from serious reliability and data integrity problems.

During the time he was creating part of a live presentation for a live course, he accidentally perpetrated two attacks on PowerPoint that resulted in some serious grief for himself.

The first attack occurred when he tried to use some features in Word, Excel, and PowerPoint that share data. Specifically, what he did was copy tables to and from PowerPoint, Word, and Excel. Now, Word and Excel shared the tables well. However, PowerPoint did not. In fact, he had to resort to things like copying and pasting from Notepad (stripping off formatting) and copying and pasting column by column.

---

17. Pairwise testing and classification trees are both covered in Rex's book *Advanced Software Testing, Vol. 1.*

Now, since he was copying formatted text from Word to PowerPoint, particularly putting that text into text boxes, he found another old nemesis in PowerPoint—doing that can cause unrecoverable errors in PowerPoint files. All of a sudden, in the middle of your work, you get a message that says your file is corrupted. You have to exit. All data since the last save are lost. And, worse yet, sometimes the data that *were* saved are actually the data causing this problem!

To get a better sense of this, you could deliberately run the attack of varying or corrupting file contents. Rex has a program that will allow him to randomly change as few as 1 or 2 *bits* in a file. Now, if he uses this on a PowerPoint presentation, that will often render the file unreadable, though special recovery software (sold at an extra charge, naturally) can recover all of the text.

However, you don't need to use his special program to corrupt your PowerPoint file. Just assume that you can share data across Office and other PC applications and try to do exactly that. Do it long enough, and PowerPoint will clobber your file for you.

This situation where PowerPoint creates its own unrecoverable error and file corruption is particularly absurd. You get an error message that says, "PowerPoint has found an error from which it cannot recover" or something like that. Sorry, but how can that be? It is its own file! No one else is managing the file. No one else is writing the file. If an application does not understand its own file format and how to get back to a known good state when it damages one of its own files, that application suffers from serious data quality issues.

4.4.5.2 Other Attacks

So far, we've been describing the specific techniques for attacks referenced in the Advanced syllabus, which is Whittaker's technique. However, you should be aware that if you are testing something other than reliability, interoperability, and functionality of stand-alone, typically PC-based applications, there are other types of attacks you can and should try.

For example, here is a list of basic security attacks:

- Denial of service, which involves trying to tie up a server with so much traffic that it becomes unavailable.
- Distributed denial of service, which is similar, is the use of a network of attacking systems to accomplish the denial of service.

- Trying to find a back door—an unsecured port or access point into the system—or trying to install a back door that provides us with access (which is often accomplished through a Trojan horse like an electronic greeting card, an e-mail attachment, or pornographic websites).
- Sniffing network traffic—watching it as it flows past a port on the system acting as a sniffer—to capture sensitive information like passwords.
- Spoofing traffic, sending out IP packets that appear to come from somewhere else, impersonating another server, and the like.
- Spoofing e-mail, sending out e-mail messages that appear to come from somewhere else, often as a part of a phishing attack.
- Replay attacks, where interactions between some user and a system, or between two systems, are captured and played back later, e.g., to gain access to an account.
- TCP/IP hijacking, where an existing session between a client and a server is taken over, generally after the client has authenticated.
- Weak key detection in encryption systems.
- Password guessing, either by logic or by brute-force password attack using a dictionary of common or known passwords.
- Virus, an infected payload that is attached or embedded in some file and then run, causes replication of the virus and perhaps damage to the system that ran the payload.
- A worm, similar to a virus, can penetrate the system under attack itself—generally through some security lapse—and then cause its replication and perhaps damage.
- War-dialing is finding an insecure modem connected to a system, which is rare now, but war-driving is finding unsecured wireless access points, which is amazingly common.
- Finally, social engineering is not an attack on the system, but on the person using the system. It is an attempt to get the user to, say, change her password to something else, to e-mail a file, etc.

Security testing, particularly what's called penetration testing, often follows an attack list like this. The interesting thing about such tests is that in security testing, as in some user acceptance testing, the tester attempts to emulate the "user"—but in this case the user is a hacker.

4.4.5.3 Software Attack Exercise

Considering the user interface for the Telephone Banker:

1. Outline a set of attacks that could be made.
2. For each attack, depict how you might make that attack.

For each attack, list the type of defect you are trying to force .

4.4.5.4 Software Attack Exercise Debrief

We would contemplate the following attacks:

- Trigger all error messages. We would get a list of error messages that are defined by the developers, analyze each message, and try to trigger each one. The defect we are trying to isolate is where the developer tried to handle an error condition but failed to do it effectively.
- Attack all default values. We would bring up each screen. For any input field that has a default value, we would remove it. At that point, we would try to process the screen. If an error was correctly thrown due to the field being NULL, we would try a different illegal value there. Often failure will occur because the developer, having put a default value in the field, assumes it does not need other error handling.
- Attack input fields. We would want to try putting in illegal values (including NULL). These would include the standards (chars, symbols, non-digit values in integer fields) but would also include trying to put complex expressions in numeric fields, using special characters that may affect the programming language used (escape characters, pipes, redirection, etc.). If the system does not parse inputs correctly, it may fail.
- Overflow buffers. Our standard attack is to paste in 4,000 characters into every input field. If the developer does not check length before using a buffer, bad things are going to happen.
- Attack dates. Try different formats, illegal values, leap year anomalies, etc. We don't find that this works often since developers tend to use well-tested libraries for dates. But, every now and then...
- Change calculated outputs. Sometimes, after inputting some values, a system will calculate some values and then ask for more input. If we can change that intermediate output, we will and see what happens when the

other input is accepted. Sometimes, the system re-inputs the values that were outputted without checking whether their values changed.

- Look for recursive input fields. In HELLOCARMS, there may be recursive fields for entering current debts or some such. We would try entering the same debt over and over and find out how the system handles it. The developer may have a limit of how many are allowed and did not document it. She might also not check for redundancies.

Use inverted values. If the system is expecting positive, put in negative numbers and vice versa. This includes silly values, like a negative interest rate. Again, we want to check lack of rigor in error handling.

4.4.6 Specification-, Defect-, and Experience-Based Exercise

Consider the following test techniques that we've covered in this chapter:

- Equivalence partitioning
- Boundary value analysis
- Decision tables
- State-based tests
- Defect-taxonomy tests
- Error-guessing tests
- Checklist-based tests
- Exploratory tests
- Software attacks

Without redundancy to previous exercises or examples, identify uses for the techniques on the HELLOCARMS project. List specification element numbers and descriptions of the application as appropriate.

The solutions are shown next.

4.4.7 Specification-, Defect-, and Experience-Based Exercise Debrief

Table 4-47 shows a listing of where we could apply the different techniques covered in this chapter to the HELLOCARMS project.

Table 4–47

Test Technique	Requirements Section or Element/Description
Equivalence partitioning	010-010-040 Armed with the list of the valid inputs for each field, check every input field to ensure that they can reject invalid values.
Boundary value analysis	010-010-040 Extend the testing of input validation using boundary value analysis.
Decision tables	010-020-010, 010-020-020, 010-020-030 Develop a decision table based on the credit policies (presumably in another document), then design tests from that decision table.
State-based tests	010-010-060 Develop a state-transition diagram for the application (rather than the Telephone Banker as was done in a previous exercise). Test the application's state-based behaviors, including the ability to interrupt and return to the interview.
Defect-taxonomy tests	010-010-040 Create a defect taxonomy for every security-related failure observed at Globobank for similar applications, augmented by information on security-related failures at other banks for similar applications. Design tests to check for these defects.
Error-guessing tests	Entire system Obtain a list of known and/or past interfacing problems between LoDoPS, GLADS, and other applications that will interoperate with HELLOCARMS. Design tests to provoke those problems, where possible.
Checklist-based tests	010-010-020 Identify every screen, flow between screens, and script. Ensure that each was tested.
Exploratory tests	010-010-170 Use a mix of PC configurations, security settings, connection speeds, customer personas, and existing customer relationships to test applications over the Internet.
Software attacks	000 Introduction Attempt attacks, especially security attacks, on the structure of the system.

4.4.8 Common Themes

So, what is true about applying these defect- and experience-based techniques?

You probably noticed a distinctly lower level of formality than the specification-based techniques. In addition, the coverage criteria are informal and usually subjective.

Testers must apply knowledge of defects and other experiences to utilize these techniques. Since many of them are defect focused, they are a good path to defect detection.

The extent to which they are dynamic and detection-focused rather than analytical and prevention-focused varies. They can be quick tests integrated into—or dominating—the test execution period. In these tests, the tester has no formally preplanned activities to perform. They can involve preplanned sessions with charters but no detail beyond that. They can involve the creation of scripted test procedures.

They are useful in almost all projects, but are particularly valuable under the following circumstances:

- There are no specifications available.
- There is poor documentation of the system under test.
- You are forced to cope with a situation where insufficient time was allowed for the test process earlier in the lifecycle; specifically, insufficient time to plan, analyze, design, implement.
- Testers have experience with the application domain, with the underlying technology, and perhaps most important, with testing.
- We can analyze operational failures and incorporate that information into our taxonomies, error guesses, checklists, explorations, and attacks.

We particularly like to use defect- and experience-based techniques in conjunction with behavior-based and structure-based techniques. Each of the techniques covered in the Advanced syllabus—both for test analysts and technical test analysts, have their strengths and weakness. So, using defect- and experience-based tests fills the gaps in test coverage that result from systematic weaknesses in these more-structured techniques.

Table 4–48

Staff	**7 Technicians**	**3 Engineers + 1 Mgr.**
Experience	<10 years total	> 20 years total
Test Type	Precise scripts	Chartered exploratory
Test Hrs/Day	42	6
Bugs Found	928 (78%)	261 (22%)
Bug Effectiveness	22	44
Scripts run	850	0
Inputs submitted	~5,000-10,000	~1,000
Results verified	~4,000-8,000	~1,000

Table 4-48 shows a case study for the Internet appliance project we have discussed several times. In it, we used a mixture of dynamic, chartered, exploratory testing and analytical, risk-based, scripted testing. The test manager and the three test engineers, who together had over 20 years total experience, did the exploratory testing. Test technicians did the scripted testing; some of the test technicians had no testing experience and others had just a little.

During test execution, the technicians each spent about six hours per day running test scripts. The rest of the time, 3 to 4 hours per day, was spent reading e-mail, attending meetings, updating bug reports, doing confirmation testing, and the like.

The engineers and managers, being heavily engaged in other tasks, could only spend 1 to 2 hours per day doing exploratory testing. Even so, due to their extensive experience, you can see that the experienced testers were star bug finders.

However, when we start looking at coverage, we can see the picture change. The technicians ran about 850 test scripts over the three months of system test. That covered a lot of ground, well-documented ground yielding well-documented results that we could show to management. The exploratory testing didn't really leave any clear documentation behind. We weren't using the session-log approach that we showed earlier, in part because we were relying on the technicians to gather the coverage evidence with the scripts.

Now, how about sheer volume of input. We can't say for sure, but we'd estimate that the manual scripted tests resulted in somewhere between 5 and 10 thousand inputs of various kinds—strings, dates, radio buttons, etc—while the exploratory testing was probably at best a fifth of that. Similarly, scripted tests probably resulted in many more explicit checks of results. Now, hour for hour the exploratory testing was probably just as effective, but it would have been less effective if we'd had to gather the session logs because that would have slowed us down.

So which was better? Ah, it wasn't that kind of experiment. It wasn't an experiment at all; it was a proven way of mixing two strategies, each with different strengths.

The exploratory testing was very effective at finding bugs on an hour-per-hour basis, and we found a number of bugs that wouldn't have been found by the scripts. The reusable test scripts gave us good regression risk mitigation, good risk mitigation, and good confidence building. Overall, a successful blended approach.

ISTQB Glossary

static analysis: Analysis of software 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.

4.5 Static Analysis

Learning objectives

(K3) Use the algorithms "Control-flow analysis" and "Data-flow analysis" to verify if code has no control or data-flow anomaly.

(K4) Interpret the control and data-flow results delivered from a tool in order to assess if code has any control or data-flow anomaly.

(K2) Explain the use of call-graphs for the evaluation of the quality of architecture. This shall include the defects to be identified, the use for test design and test planning, limitations of results.

So far in this chapter, we have seen a number of different ways to detect defects, all of which require the tester to execute code to force failures to show themselves. There are other ways to find defects however; we will look at three of them.

One of the most productive ways to find defects is manual static testing, or reviews. In the Foundation syllabus, we discussed informal reviews, walkthroughs, technical reviews, and inspections. We will review those a bit and add some more material; we will do that in chapter 6.

The second way to find defects that we will discuss is called static analysis. We examined this 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. Here is the formal definition of static analysis, as given in the Advanced syllabus:

> *Static analysis is concerned with testing without actually executing the software under test and may relate to code or the system architecture.*

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.

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

> *Analysis of software 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 too restricting. While tools are likely to be used, there certainly may be times when we do the static analysis manually.

We will examine such topics as control-flow analysis, data-flow analysis, compliance to standards, certain code metrics, and call-graphing. Finally, we will look at dynamic analysis wherein we utilize tools while executing system code to help us find a range of failures that we might otherwise miss; that will be in the next section.

Remember to review the benefits of reviews and static analysis from the Foundation syllabus; we will try not to cover the same materials.

4.5.1 Complexity Analysis

The more complex the code is, the more likely it is to have bugs. 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 three to seven cyclomatic complexity range—who regularly had far lower defect density.

This kind of anecdotal proof is not sufficient for us to decide whether complexity is truly the enemy. There have been a number of studies that concluded that there is a strong correlation between higher complexity and higher defect density. Other studies question direct causation.

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.

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 speed of execution was just too slow. 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 of it was highly problematic; it took much longer to bring it to a high level of quality than it did other 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 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 (while suggesting a written explanation when the cyclomatic complexity is structured that high).

Earlier in this chapter we discussed cyclomatic complexity and showed a manual way to determine a module's complexity. Now we will look at the output of an automation tool that measures complexity, a much more realistic way to deal with it.

The cyclomatic complexity graphs in figure 4-51 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 can focus both testing and reengineering on the modules that are most likely to need it.

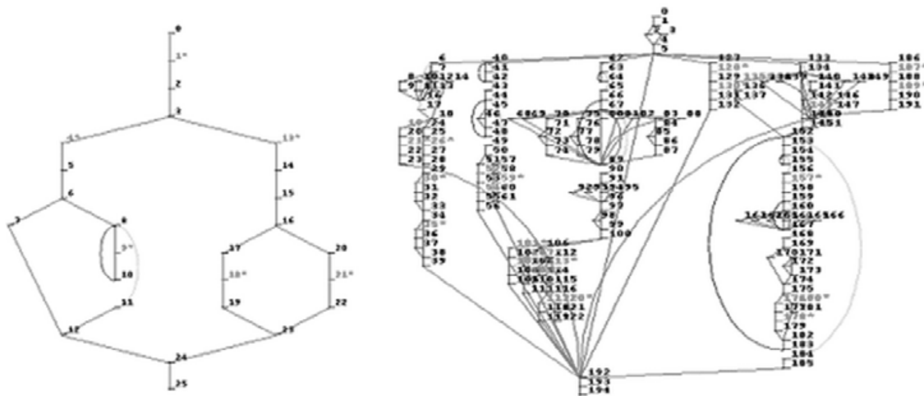


Figure 4–51 *Simple vs. complex graphs*

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.

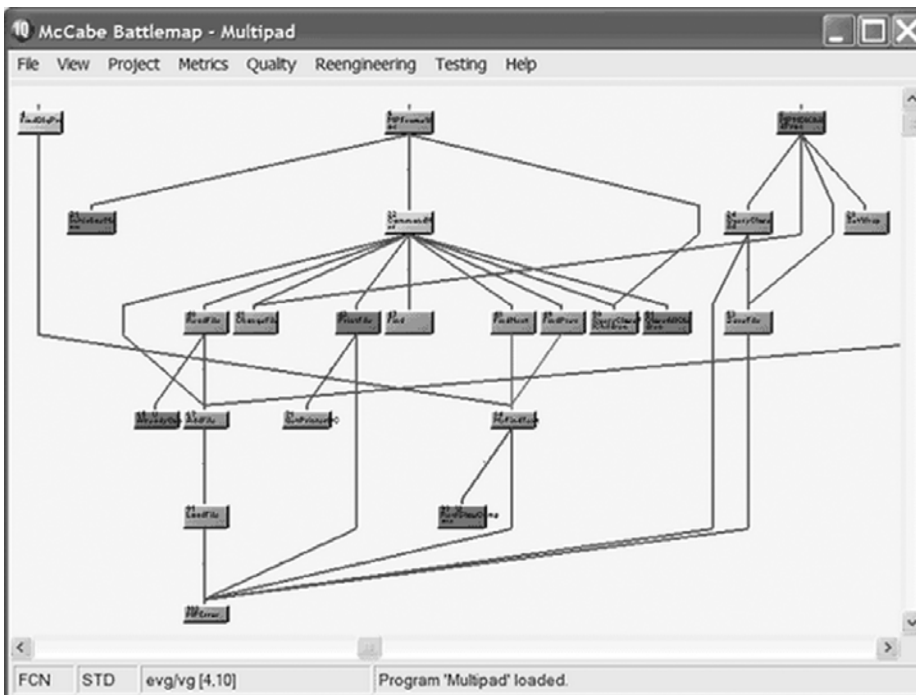


Figure 4–52 *Module complexity view*

If figure 4-51 was a 100-foot-high view of two different modules, figure 4-52 is an overall view of our system from above 1,000 feet. 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. Different colors are assigned to different levels of complexity. (In the original full-color image green modules have low complexity ratings, yellow are considered somewhat complex, and red means highly complex.) Clearly, 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 effort.

While these tools tend to be somewhat pricey, their cost needs to be compared with the price tag of releasing a system that might collapse under real usage.

4.5.2 Code Parsing Tools

In figure 4-53 , we show the output of a static code parsing tool that is looking for problems that the compiler would not flag because they are not syntactically incorrect. The code on the left is an example of a very poorly written C language module. The problem is that it does not necessarily look bad. Many developers figure that as long as the compiler does not complain, the code probably will work okay. Then they spend hours debugging it when it does not do what they want. 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)
3. Assignment of into 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 4–53 *Code parsing tools*

Notice in item 1, the variable *c* is evaluated before it is set. That is not illegal, just dim-witted. The variable *c* is set to whatever 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 new languages, such as C#, will flag this as an error when compiled; most programming languages allow it. Other programming languages allow it but also 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 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 may 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 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.

Oftentimes the output of this kind of static analysis tool is obtuse and difficult to understand. Sometimes the warning messages are for conditions that the programmers understand and did on purpose. For these and many other reasons, programmers often start ignoring various warning messages. Some of these tools allow the user to selectively turn off certain classes of messages to avoid the clutter.

As testers, however, it is likely worth our 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.

As we mentioned earlier (and will further discuss when we get to non-functional testing), maintainability is an important concept for an organization that plans on surviving for more than a short time.

4.5.3 Standards and Guidelines

Programmers sometime consider programming standards and guidelines to be wasteful and annoying. We can understand this viewpoint to a limited extent; developers 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 too onerous. Testers need to remind developers that, long term, higher maintainability of the system will save time and effort later.

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 exam-

ple, if we are writing an emergency patch, we may be more relaxed about following exact commenting guidelines 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 Credence 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 his or her 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 caps. Variables might have their data type encoded. 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 everyone's 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, etc.
- 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 because the change is made 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."[18] This document reflects the Java language coding standards presented in the *Java Language Specification*[19] by Sun Microsystems. 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, he 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. However, using that knowledge may become problematic if the owner of that class decides to change the code. Those changes might end up breaking the 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.

18. http://java.sun.com/docs/codeconv/html/CodeConventions.doc.html
19. http://java.sun.com/docs/books/jls/

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.

4.5.4 Data-Flow Analysis

Our next 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 is about the lifecycle 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 lifecycle, doing the wrong thing at the right time, or the trifecta, doing the wrong thing to the wrong data at the wrong time.

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.*[20]

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.

20. The earlier work was a paper: Rapps, S., and E. J. Weyuker. "Data-flow analysis techniques for test data selection." Sixth International Conference on Software Engineering, Tokyo, Japan. September 13-16, 1982.

- 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.
- Redefining a variable 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.

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 the 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 she is doing (sometimes a dodgy assumption, we testers think).

Different languages have data scoping rules that can interact in very subtle ways. Personally, we often find ourselves making mistakes in C++ because of its scoping rules. A variable may be declared global or local, static or stack based; it may 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 power is given to the programmer by having special ways of dealing with data, the programmer 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 sign would be legal and the single equal sign 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 strange 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 are 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 slip. 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 your testing arsenal.

4.5.5 Set-Use Pairs

Data-flow notation comes in a few different flavors; we are going to look at one of them here. This one is sometimes called *set-use pair* notation.

We will split the lifecycle of a data variable into three separate patterns:

d: This stands for the time when the variable is created, defined, or initialized.

u: This stands for used. The variable may be used in a computation or in a decision predicate.

k: This stands for killed, destroyed. or has become out of scope.

These three atomic actions are then combined to show a data flow. A ~ (tilde) is often used to show the first or last action that can occur.

Table 4–49

	Anomaly		Explanation
1.	~d	first define	Allowed.
2.	du	define-use	Allowed, normal case.
3.	dk	define-kill	Potential bug; data were never used.
4.	~u	first use	Potential bug; data were 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 are killed before definition.
8.	ku	kill-use	Serious defect; data are used after being killed.
9.	kd	kill-define	Usually allowed. Data are killed and then redefined. Some theorists believe this should be disallowed.
10.	dd	define-define	Potential bug; double definition.
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 used in another context.
14.	u~	use last	Allowed. Variable was used in this routine but not killed off.
15.	k~	kill last	Allowed; normal case.

Referring to table 4-49, there are 15 potential combinations of the atomic actions we are concerned with:

1. ~d, or first define. This is the normal way a variable is originated; it is declared. Note that this particular data-flow analysis scheme is somewhat hazy as to whether at this point the value is defined or not. A variable is declared in order 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, 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.
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 since the data were 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 are 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 define 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. This is a potential bug. The question should be asked why the variable was defined the first time only to be defined again before using it. At the very least, it is inefficient. As you will see in our example, there may be a time when this is useful.
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.

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 check all global variable use very carefully.
14. u~, or use last. This is common; it usually happens when variables simply run out of scope at the end of a routine.
15. k~, or kill last. This is the normal case.

Following is an example to show how we use set-use pair analysis.

4.5.6 Set-Use Pair Example

Assume that a telephone company provides the following cell phone plan: If the customer uses up to 100 minutes (inclusive), then there is a flat fee of $40 for the plan. For all minutes used from 101 to 200 minutes (inclusive), there is an added fee of $0.50 cents per minute. Any minutes used after that are billed at $0.10 per minute. Finally, if the bill is over $100 or over, a 10 percent discount on the entire bill is given.

A good tester would immediately ask the question as to how much is billed if the user does not use the cell phone at all. Our assumption would be that they still have to pay the $40. However, the code as given in figure 4-54 lets the user off scot-free.

```
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 4–54

Line 3 looks at the number of minutes billed. If none were billed, the initial $40 is not billed. Instead, it evaluates to FALSE and a $0.00 bill is sent. Frankly, we wouldn't mind if this were our phone company, but if we worked there, we would flag this as an error.

Assuming a little time was used, however, the bill is set to $40 in line 4. In line 6 we see if more than 100 minutes were 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 cents 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 or equal to $100.00.

For each variable, we create a d-u-k (define-use-kill) pattern list that tracks all the changes to that variable through the module. In each case, we track the line(s) in which an action takes place.

For the code in figure 4-54, table 4-50 shows the data-flow information for the variable *Usage*:

Table 4–50

	Pattern	Explanation
1.	~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~ (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.[21]
2. There is one du (define-use) pair at (1-3). This is simply saying that the variable is defined on line 1 and used on line 3. This is expected behavior.
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. The *Usage* variable is used on line 3 and line 6. Then comes (6-7), (7-8), and (7-11). Notice that 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 4-51.

Table 4–51

	Pattern	Explanation
1.	~d (2)	normal case
2.	dd (2-4)	suspicious
3.	du (2-18)(4-8)(4-11)(11-12)	normal case
4.	ud (8-8)(11-11)(13-13)	acceptable
5.	uu (12-13)(12-18)	normal case
6.	uk (18-19)	normal case
7.	k~ (19)	normal case

21. 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.

1. ~d signifies that this is the first time this variable is defined. This is normal for a local variable declaration.
2. dd (define-define) should 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 was 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 a statement in which a variable is declared. 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:
   - The pair (2-18) happens when there are no phone minutes and ensures that we do not return an undefined value.
   - The pair (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 *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 occur only 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 earlier in this chapter? Data-flow testing would tell us the code was sound; unfortunately, it would not tell us that when inputting 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 4-55, you see the equivalent 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.

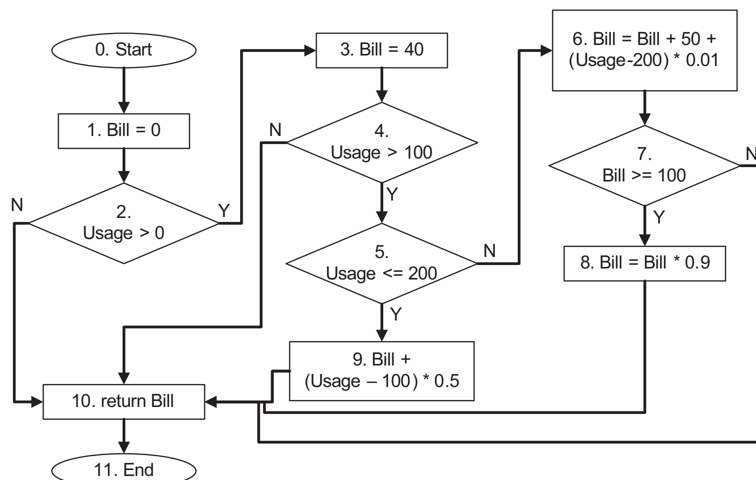


Figure 4–55 *Control-flow diagram for example*

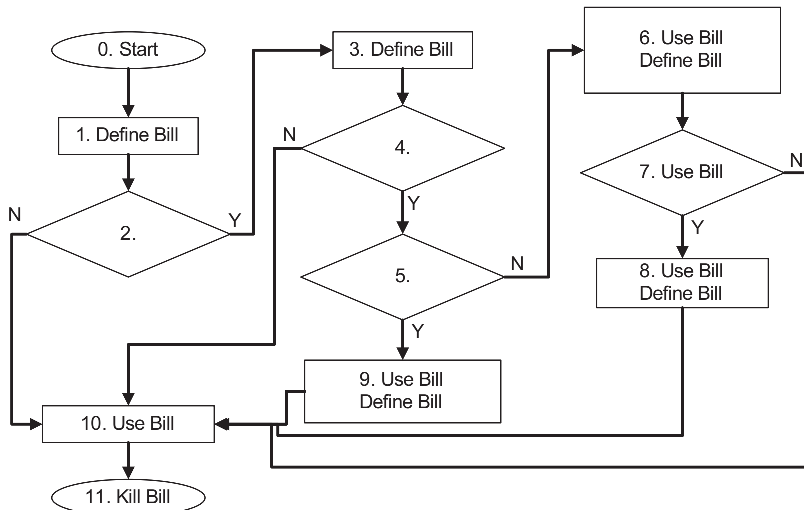


Figure 4–56 *Annotated flow graph for variable Bill*

Using the control-flow diagram from figure 4-55, we can build an annotated flow graph for the variable Bill as seen in figure 4-56. It is much easier to find the data connections when they are 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.

Table 4–52

Anomaly		Explanation
~d	1	Normal case
dd	1-2-3	Potential bug
du	1-2-10	Normal case
	3-4-5-6	Normal case
	3-4-5-9	Normal case
	3-4-10	Normal case
	6-7	Normal case
	8-10	Normal case
	9-10	Normal case
ud	6-6	Normal case
	8-8	Normal case
	9-9	Normal case
uu	7-8	Normal case
	7-10	Normal case
uk	10-11	Normal case
k~	11	Normal case

From the annotated flow graph, we can create a table of set-use pairs as shown in table 4-52. 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.

4.5.7 Data-Flow Exercise

Using the control-flow and 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.

4.5.8 Data-Flow Exercise Debrief

Figure 4-57 shows the annotated data-flow for the code.

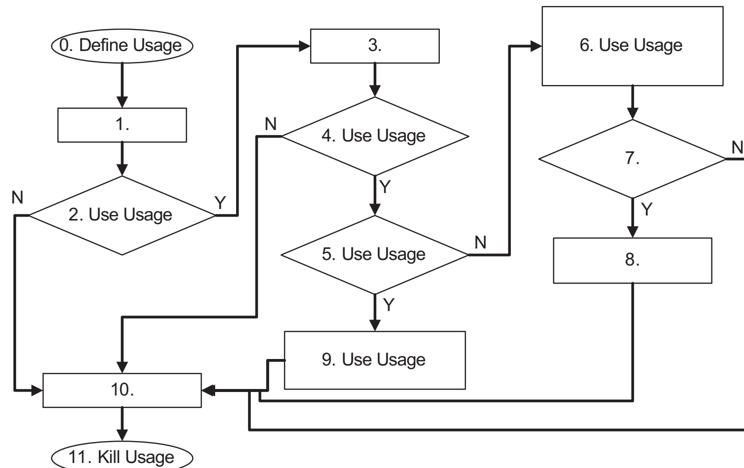


Figure 4–57 *Annotated flow graph for Usage*

Table 53 represents the data flows.

Table 4–53

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.

4.5.9 Data-Flow Strategies

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 all 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 a memory location. In this particular line of code, *Z* is a *define* type use, so it's 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 they appear directly in a condition expression. Consider this particular line of code:

if (A < X) {}

This is an *if()* statement where the condition 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, in expressions used to evaluate the selected branch of a case statement, or 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. 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 one 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[22]. While the math is beyond 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*.[23] A somewhat more dense discussion can be found in *Software Testing Techniques*.[24]

22.By off-type, we mean if the main tests are computational and we need more tests, then use predicate use values (or vice versa).

23. *The Compiler Design Handbook: Optimizations and Machine Code Generation*, Second Edition, by Y.N. Srikant and Priti Shankar.

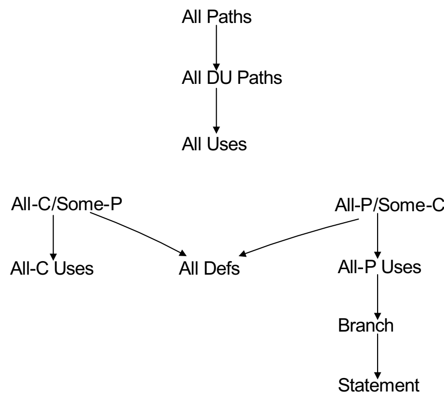


Figure 4–58 *Rapps/Weyuker hierarchy of coverage metrics*

Figure 4-58 may help as it tries to put the preceding discussion into context. At the very top of the structure is all paths—the holy grail of testing that, in any non-trivial system, is impossible to achieve. Below that is *ADUP*, which gives the very best 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 side of the figure. These are not comparable with their mirror image shown on the right side; mathematically, they are inherently weaker than the other strategies at their same level. The *all p-uses* (*APU+C* and *APU*) are comparable with and 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 4-58 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.

4.5.10 Static Analysis for Integration Testing

In the Foundation 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

24. *Software Testing Techniques, Second Edition*, by Boris Beizer

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 *backbone,* method).

What these methods have in common is the concept of test harnesses. Consisting of drivers and stubs, they represent non-shippable code that we write to be able to test a partial system.

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 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."*[25]

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 know where the problem is.

In some organizations, however, the biggest roadblock to good integration testing is the large amount of non-shippable 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 to a project manager when other testing techniques (i.e., the big bang) are available. In addition, when the system is nontrivial, 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.

25. Foundation syllabus, Section 2.2.2

4.5.11 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 work together using graph theory.

Two arguments against incremental integration are the large number of builds needed and the amount of non-shippable 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 bugs, 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 non-shippable code) without exacerbating the latter two (finding bugs late and difficulty of narrowing down where the bugs are).

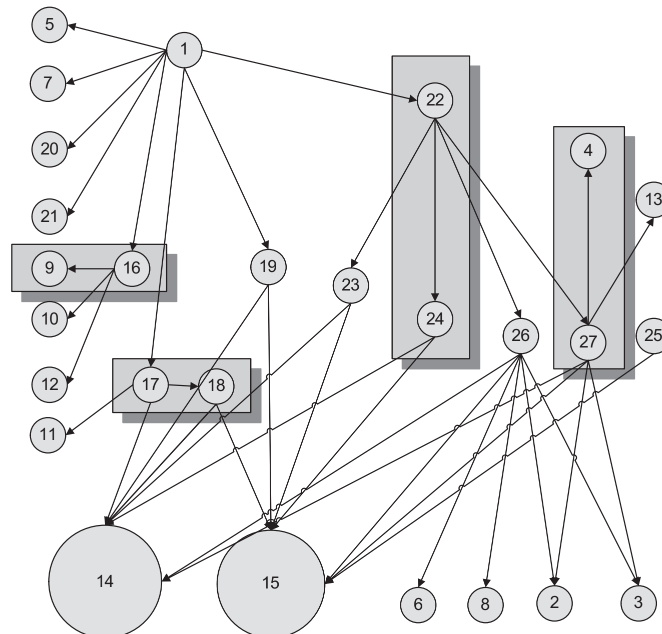


Figure 4–59 *Pairwise graph example*

Take a look at figure 4-59. This graph from Paul Jorgensen's book 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 where the system is broken when we find a failure. Since we are not concentrating on systemwide 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 non-shippable code that is required is reduced to almost nothing since the entire build is being tested.

In the graph in figure 4-59, four sessions are shown, modules 9-16, 17-18, 22-24, and 4-27.

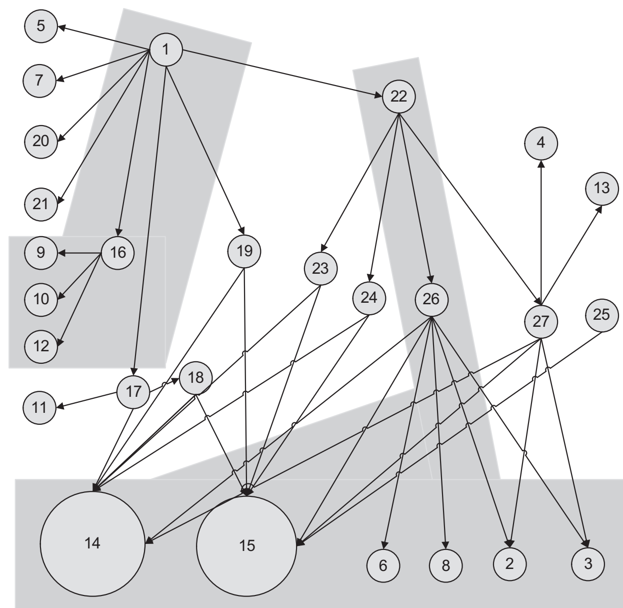


Figure 4–60 *Neighborhood integration example*

The second technique, shown in figure 4-60, is called neighborhood integration. This uses the same integration graph that you saw in figure 4-59 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 4-60, 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, while testing, 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 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.

4.5.12 McCabe Design Predicate Approach to Integration Testing

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 give us the basis set—the minimum number of test cases we needed.

It turns out that McCabe also has some theories about how to approach integration testing. This technique, which is called McCabe's design predicate approach, consists of three steps:

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 moment.
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 we need to cover the structure of the system.

We will use the basis path/basis tests terminology to describe these tests.

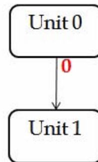


Figure 4–61 *Unconditional call*

The first design predicate (figure 4-61) 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 test-

ing, those are all ignored; we are interested in testing how modules communicate and work together.

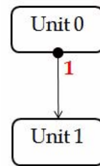


Figure 4–62 *Conditional call*

We might decide to call another module, or we might not. figure 4-62 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 this book, we would hope the answer is always 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 when we do not.

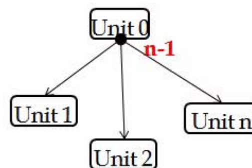


Figure 4–63 *Mutually exclusive conditional call*

The third design predicate (figure 4-63) 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 2 (two), calculated by the number of possibilities minus one. Note in the graph that a filled-in circle at the tail of the arrows shows that some of the targets will not be called.

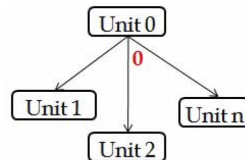


Figure 4–64 *Unconditional calls*

In the next graph (figure 4-64), 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, they are unconditional connections, hence there is no increase in complexity.

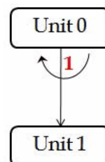


Figure 4–65 *Iterative call*

In (figure 4-65), 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.

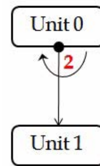


Figure 4–66 *Iterative conditional call*

Last but not least, we have the iterative conditional call. As seen in this last graph (figure 4-66), 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 earlier in the chapter.

4.5.13 Hex Converter Example

In an exercise earlier in this chapter, you saw the basic hex converter code. The code in figure 4-67 is a hex converter, but now we have enhanced it for the real world.

This code is for UNIX or Linux; it will not work with Windows without modification of the interrupt handling. Here is the explanation of the code.

When the program is invoked, it calls the *signal()* function. If the return value tells m*ain()* that **SIGINT** is not currently ignored, then m*ain()* 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 *fprint()* 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 appended 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.

```
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, qotnum;
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, "hexcvt: 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 4–67 *Enhanced hex code converter code*

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.

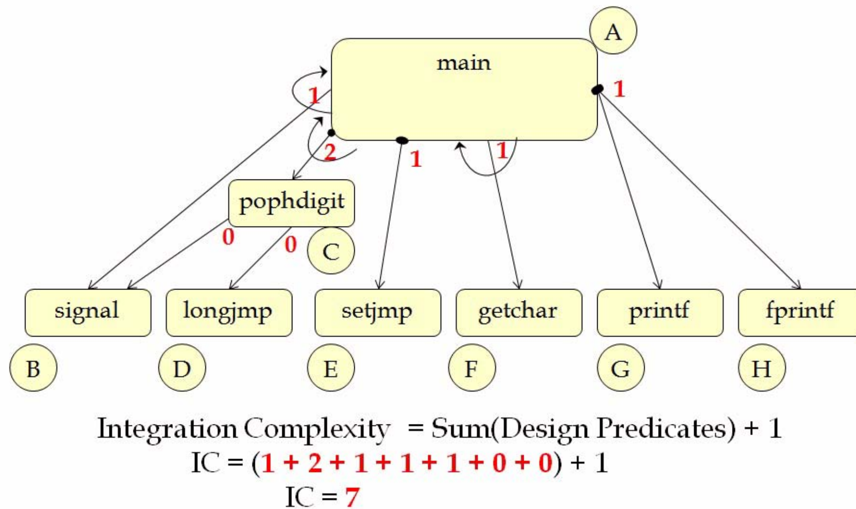


Figure 4–68 *Enhanced hex converter integration graph*

Figure 4-68 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 *pophdigit()* (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 time, 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 *pophdigit* (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 go in only 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 not really what we have been saying. The refrain we keep coming back to is that 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 risk-based testing (RBT) because he kept on insisting that as long as there was one test per requirement, 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 (see figure 4-68).

1. We want to input just an *A* 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()*, and *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 and 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 *ABBEFFFCBDFFFG* and should execute in the following order: *main(), signal, signal, setjmp(), getchar(), getchar(), getchar(), pophdigit(), 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 *pophdigit* (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-Cs that were inputted.

Would we want to test this further? Sure. 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!

4.5.14 McCabe Design Predicate Exercise

Calculate the integration complexity of the call-graph in figure 4-69.

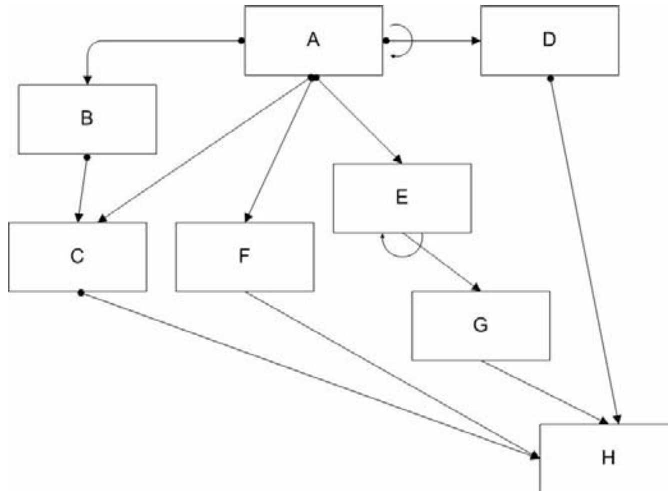


Figure 4–69 *Design predicate exercise*

4.5.15 McCabe Design Predicate Exercise Debrief

We tend to try to be systematic when going through a problem like this. We have modules A through 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)

ISTQB Glossary

dynamic analysis: The process of evaluating behavior, e.g., memory performance, CPU usage, of a system or component during execution.

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

4.6 Dynamic Analysis

Learning objectives
(K2) Explain how dynamic analysis for code can be executed and summarize the defects that can be identified using that technique, and its limitations.

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 almost 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.

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.

wild pointer: A pointer that references a location that is out of scope for that pointer or that does not exist.

Dynamic analysis is a solution to a common problem 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, 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 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.

It may be a wild pointer that changes a byte on the stack erroneously; the byte that was changed is in the instruction flow so that when it is executed, it does the wrong thing; instead of adding two values, it rotates the current word in the processor.

It may be an application programming interface (API) call to the operating system that has the wrong arguments, so the operating system allocates too small a buffer and the input data from a device are 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 errors.

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 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 them 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 or a bad API call is made, 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 (integrated development environment) 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 since they save data to the log that indicate 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 it very useful for testers to use them 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 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 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 tools tend to be more pricey, they may well be 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 as the timing of the system can change radically. And some of these tools require development artifacts be available, including code modules, MAP files, etc.

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.

4.6.1 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, and 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. Some tools then write that information to a log, while others 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 4–70 *Memory leak logging file*

Figure 4-70 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.

4.6.2 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 on 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. Unfortunately, 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 are stored in a permanent location where they 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.

4.6.3 API Misuse Detection

The final target for dynamic analysis tools that we will address is API errors. APIs (application programming interfaces) are everywhere and becoming more prevalent. Essentially, an API is created when we have remote functionality that can be shared. The operating system supplies almost all of its services through APIs, as do networks, the Web, and most remote services. If we need to open a file, allocate some memory, listen to the keyboard or external device, put something on the screen or monitor the mouse, we need to call the correct APIs in the proper order and with the correct arguments. COM/DCOM objects in Windows, ODBC and most middleware, sockets and synchronization structures: these all depend on the proper usage of APIs.

With so much of the functionality of the working system supplied by other entities through APIs, the possibilities for failure are endless. Often, many of the APIs that are called are hidden by the compiler. For example, if we want to open a stored file to read it, most programming languages have a simple function—let's call it *OpenFile()*—that does the work for us. What the compiler generates, however, are a number of API calls to the operating system to actually do the dirty work. Ideally, the compiler generates good code that makes calls in the correct way. As a rule, the *OpenFile()* call will return an error code if there are any problems during the task execution.

API errors are usually not surfaced explicitly to the user of a system. Instead, errors are returned to the calling system, often as a return code value. If the programmer does not evaluate the return code, the error goes unnoticed. So if our programmer does not process the return value of *OpenFile()*, the failure to correctly open the file may not be caught immediately.

Unfortunately, simply because the error was not globally surfaced does not mean it did not occur. Something has not worked as expected—in this case the file didn't open correctly, and the repercussions to that failure tend to snowball. Often, a user-detected symptom at a given point may have been caused by a simple API error billions of computer cycles in the past. It goes without saying that trying to track down failures like this is time consuming and ultimately frustrating.

Dynamic analysis tools work by catching the API call before it executes and checking the given arguments against the expected parameter list. Likewise, the order of related API calls are monitored. When a violation of given rules is found, the system may log the anomaly or, in some cases, halt execution immediately.

Clearly, this level of interaction with the running program slows the system down quite a bit. Most of the tools that we have used allow the user to limit the scope of what is being monitored. This allows the user of a tool to fine-tune it as needed.

Depending on the type of testing we are doing and the kind of system we are testing, an appropriate dynamic analysis tool may be as important a tool as we have in our toolbox.

4.7 Sample Exam Questions

1. While performing the post-mortem evaluation on a project, you are perusing the fixes to a particular code module. You notice that there are a number of errors made using the operators >, >=, <, and <=. Which of the following specification-based test design techniques would be best suited to catch these type errors?

   A. Equivalence class partitioning

   B. Boundary value analysis

   C. State transition table

   D. State transition diagram

2. You are testing a data input screen for a software package. The data that is collected is used in calculating the taxes for the organization being processed. You are at the analysis and design phase of your lifecycle, going field by field

trying to determine the minimum number of test cases you need to give you BVA (boundary value analysis) coverage. Your stated objective is to perform both positive and negative equivalence class testing on the screen. You have an automated process that will automatically test all field values other than the correct data type: that is, if the field requires an integer, the automated program will test chars, symbols, spaces, real numbers, nulls, etc. Therefore, you are going to leave all of those types of negative test cases out of your design. One field you must consider is the customer type field, which requires a single integer input. For historical purposes, there are five different type customer classifications (1, 2, 4, 5, 6). What is the minimum number of tests you must design for this field to achieve the desired coverage?

A. 2

B. 5

C. 6

D. 8

3. A bank has several different levels of awards/penalties that it showers on its customers. Both are based on the amount of dollars in customers' various accounts. Assume that you have a test client with five (5) different accounts that are subject to these rules. Different types of accounts are calculated differently, even though the trigger amounts are the same for all accounts. The rules for any given period are as follows:

- Negative balances are penalized by a fee.
- Balances at or below $25 are penalized by loss of interest.
- Balances at or below $1,000 are penalized by low interest.
- Balances above $1,000 get full interest.
- Balances above $100,000 get full interest and a $200 interest bonus.

Assuming that we are interested in equivalence class testing, how many test cases do we need minimum for this client?

A. 4

B. 5

C. 25

D. 40

4. Consider the bank scenario as defined in question 3. Assuming that calculations are made to the cent, and that we are interested in boundary value testing, what is the minimum set of test values we would need for an account?

   A. (-0.01; 0.00; 24.99; 25.00; 999.99; 1,000.00; 999, 999.99; 100,000.00)

   B. (-0.01; 0.00; 25.00; 25.01; 1,000.00; 1,000.01; 100,000.00; 100,000.01)

   C. (0.00; 0.01; 25.00; 25.01; 1,000.00;1,000.01;100,000.00;100,000.01)

   D. (-0.001;0.001;24.999;25.001;999.999;1,000.001;999,999.999;100,000.001)

5. We have built a decision table to help us develop a solution to Glenford Myers's triangle test problem. A piece of software inputs three integer numbers. These numbers represent the lengths of the sides of a geometric figure: a, b and c. The values are compared to see if they actually represent a legal triangle. A triangle with three equal sides is called an equilateral triangle. One with two equal sides is called an isosceles triangle. One with no equal sides is called a scalene triangle. Using what you know of triangles and decision tables, collapse the given decision table and determine the number of tests needed to achieve minimum coverage criteria.

Conditions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Legal triangle	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
a = b	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
a = c	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
b = c	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Results																
Scalene triangle																
Isosceles triangle																
Equilateral triangle																
Not legal/ impossible																

   A. 6

   B. 7

   C. 9

   D. 12

6. Refer to the following decision table. Which of the following is correct based on what you see in the decision table?

Conditions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Balance OK?	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Number of checks OK?	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
Savings account?	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Payment late?	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Results																
Standard account fee?			Y	Y												
Charge excess fee?					Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Charge late fee?	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Charge no fee?	Y	Y														

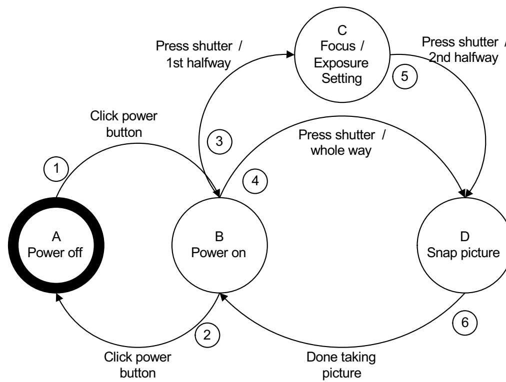
A. There are no non-exclusive rules in this decision table.

B. There is one non-exclusive rule: Balance OK?

C. All of the conditions in the decision table are non-exclusive rules

D. There is one non-exclusive rule: Payment late?

7. You are going to be testing a small camera. To keep it simple, there are only two buttons: the power button and the shutter button. The camera will act as an automatic focus and exposure camera when the shutter button is pressed halfway and then stopped for 0.4 seconds before the button press is completed. If the shutter button is simply pressed all the way without stopping, the camera will act as a single-exposure, infinity focus camera. Given the following state transition diagram, how many rows would its matching state transition table contain?



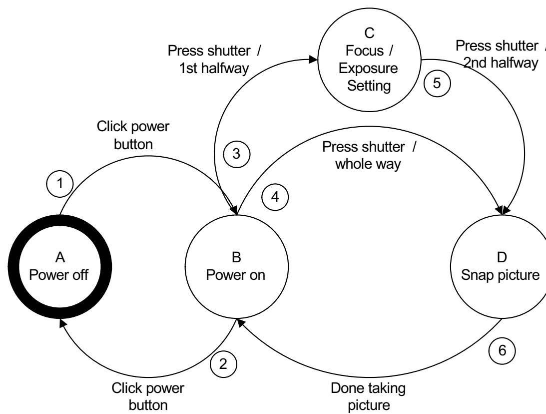
A. 24

B. 20

C. 18

D. 6

8. Refer to the scenario as defined in question 7. Given the following partial switch coverage table, how many 1-switch segments would there be?



0-switch				1-switch
A1				
B2	B3	B4		
C5				
D6				

A. 5

B. 7

C. 9

D. 11

9. 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.xyz.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

10. 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)

11. 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

12. In a module of code you are testing, you are presented with the following *if*() statement. A number of tests are given in the form of 3-tuples, where A will get the first value, B will get the second value, C will get the third. Which test must be added to achieve full MC/DC coverage?

```
if ((A && B) || C) {
  Execute some statements;
}
Else {
  Execute some statements;
}
```

Test 1: (F, F, T)
Test 2: (F, F, F)

A. (F, T, F)
B. (T, F, T)
C. (T, F, F)
D. (T, T, F)

13. 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 program')
```

Which of the following 3-tuple represents a valid LCSAJ?

A. (1,2,14)

B. (2,2,14)

C. (2,4,10)

D. (5,10,12)

14. 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

15. The QA group has spent the last month mining the defect tracking system looking for all of the failures that had been reported, not only in testing but in production. The output of this research has been turned over to the test group for use in the next software release. The use of this information for testing would represent which of the following test design techniques?

    A. Taxonomy based

    B. Error guessing

    C. Checklist based

    D. Exploratory

16. Your test consists of starting up a huge number of applications to load both existing RAM and virtual memory to capacity. You then will proceed to execute a number of scenarios using the application under test. What kind of testing are you most likely doing?

    A. Charter-based exploratory testing

    B. Error guessing

    C. Structure-based testing

    D. Software attack-based testing

17. 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.xyz.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 through 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.

18. Which of these data-flow coverage metrics is the strongest of the four listed?

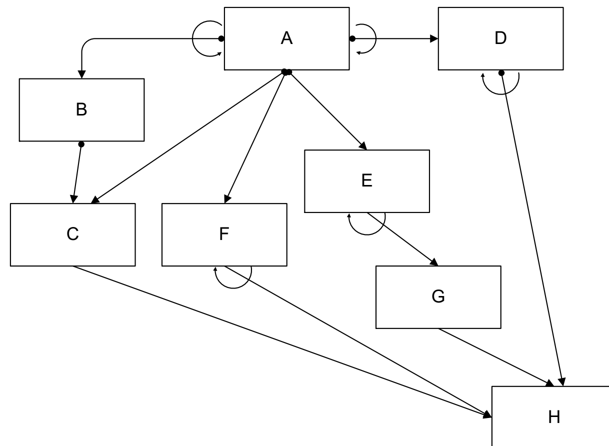
    A. All-P uses

    B. All-C uses

    C. All Defs

    D. All DU paths

19. Given the following integration call-graph, and using McCabe's cyclomatic complexity theory, how many basis paths are there to test?



    A. 12

    B. 10

    C. 13

    D. 14

20. Consider the following list:

    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

    Which of these are most likely to be found through the use of a dynamic analysis tool?

    A. I, III, IV, and VII

    B. I, II, III, IV, and VI

    C. I, II, and IV

    D. II, IV, and V

5 Tests of Software Characteristics

–You can't just turn on creativity like a faucet. You have to be in the right mood.
–What mood is that?
–Last-minute panic.

Bill Watterson, from Calvin and Hobbes
(Hobbes would have made a great tester!)

The fifth chapter of the Advanced syllabus is concerned with tests of software characteristics. In this chapter, the Advanced syllabus expands on a concept introduced in the Foundation syllabus, that of ISO 9126 software quality characteristics, to explain testing as it relates to various attributes of functional and non-functional software quality. There are three sections.

1. Introduction
2. Quality Attributes for Domain Testing
3. Quality Attributes for Technical Testing

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

5.1 Introduction

Learning objectives

Recall of content only

At the beginning of chapter 4, we introduced a taxonomy—a classification system—for tests. Figure 5-1 shows that taxonomy. If you recall, we mentioned the distinction between functional and non-functional black-box tests, based on the ISO 9126 standard. We then went on in chapter 4 to talk about useful black-

box techniques, without returning to this distinction or to the characteristics and subcharacteristics of quality defined in ISO 9126.

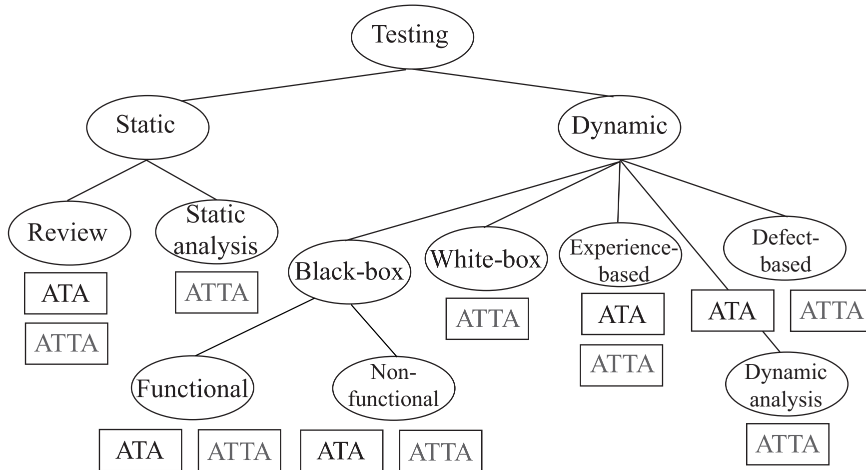


Figure 5–1 *Advanced syllabus testing techniques*

In this chapter, we will return to those topics. Here, we consider how to apply the techniques from chapter 4 to evaluate the quality of software applications or systems. While the focus will be different for technical test analysts in this book than in the companion volume for test analysts (*Advanced Software Testing, Vol. 1*), the common element is that we need to understand quality characteristics in order to recognize typical risks, develop appropriate testing strategies, and specify effective tests.

As we look at ISO 9126, it is important to understand exactly what the standard entails. There are four separate sections to this standard.

The first section, ISO 9126-1, is the quality model itself, enumerating the six categories and the subcategories that go along with them.

ISO 9126-2 is for external (dynamic) measurements. 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.

ISO 9126-3 is for internal (static) measurements. 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. ISO 9126-4 is for quality-in-use metrics[1]. 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.

5.2 Quality Attributes for Domain Testing

Learning objectives

(K2) Characterize non-functional test types for domain testing by typical defects to be targeted (attacked), their typical application within the application lifecycle, and test techniques suited to be used for test design.

(K4) Specify test cases for particular types of non-functional test types and covering given test objectives and defects to be targeted.

Functional testing focuses on what the system does rather than how it does it. Non-functional testing is focused on how—or how well—the system does what it does. Both functional and non-functional testing are predominately black-box tests, being focused on behavior. White-box tests are focused on how the system works internally, i.e., on its structure.

Non-functional testing may depend in part on the requirements, but many of the tests will come from implicit requirements and from trying to ensure that the system does what it is supposed to do—for lack of a better phrase—in an elegant way.

Non-functional testing will vary by test level; we will discuss the proper levels to test using different non-functional test types in the upcoming sections. The technical test analyst can employ various test techniques during domain[2]

1. Metrics that are only available when the final product is used in actual production environment under actual conditions.

2. The ISTQB Advanced syllabus appears to use the term *domain testing* for work that it assigns to test analysts (i.e., the functional attribute of ISO 9126). However, there are many types of functional tasks that call out for technical skills. These particular tasks are covered in this book; less-technical tasks are covered in *Advanced Software Testing Vol. 1*.

ISTQB Glossary

accuracy: The capability of the software product to provide the right or agreed-upon results or effects with the needed degree of precision.

accuracy testing: The process of testing to determine the accuracy of a software product.

testing at any level. All of the techniques discussed in chapter 4 will be useful at some point or another.

You should keep in mind that the technical test analyst is a role, not a title, job description, or position. In other words, some people play the role of technical test analyst exclusively, but others play that role as part of another job. So when dedicated, professional testers do non-functional testing, they are technical test analysts both in position and in role. However, when technical experts do the analysis, design, implementation, or execution of functional tests, they are working as test analysts.

Technical test analysts, according to the ISTQB Advanced syllabus, should be able to identify opportunities to use test techniques that are appropriate for testing:

- Accuracy
- Suitability
- Interoperability
- Usability
- Security

We'll look more closely at each of these areas in the following sections.

5.2.1 Accuracy

Accuracy is defined as the capability of a system to provide the provably correct (or at least agreed-upon) results to the needed level of precision. Ideally, the requirements will give exact specifications for accuracy. Of course, in the real world, this is often not the case. Technical test analysts often have to search for a meaningful test oracle to determine what specifications the system is supposed to perform to.

An important type of testing to determine accuracy will be boundary analysis. Of course, to do meaningful testing of exactly where the boundaries are, the

tester must be familiar with the data representations that are going to be used. This is especially critical as data is moved from one store to another. For example, when data are moved from the database to the active process, are all data transforms done correctly without loss of precision?

Decision tables will be required when the system calculates values based on multiple inputs that interact. That interaction may cause disparate data types to be combined during the calculations, possibly affecting the accuracy of the final values.

A good synonym for *accuracy* is *correctness*; does the system always give the correct answer to a given input or set of inputs?

So when is accuracy testing liable to occur? During unit testing, the tester should use the correct data types to ensure that the correct data is stored in memory. Likewise, when sent to permanent storage, the data must be stored in such a way that it does not change the values in a material way.

During integration testing, we need to ensure that precision is not lost as data travels between interfaces. Truncated and rounded values are a possible source of problems during this transfer. Likewise, incorrect buffer size could easily be an issue. When dealing with APIs to COM and DCOM[3] objects, the operating system, interoperating systems, remote procedure calls, middleware, etc., data may be manipulated, massaged, and squeezed, causing a lack of precision. A technical test analyst must be ready to test the entire data path.

During system testing, the correctness of data and calculations is a primary concern. Luckily, this is where we are most likely going to have good specifications to test to. A technical test analyst should be involved early, understanding expected behavior with respect to correctness and accuracy. Reviewing the low-level design and documentation and code will be useful in making sure that we have a platform where accuracy is ensured.

Finally, in acceptance testing, if we do our job correctly throughout the rest of the process, the customers should be happy with the results of their scenarios, reports, and queries against the system. If we are accurate to the requirements but the customer is unhappy, there is obviously a problem in our determination of what correctness means.

3. COM: Component Object Model; DCOM: Distributed Component Object Model

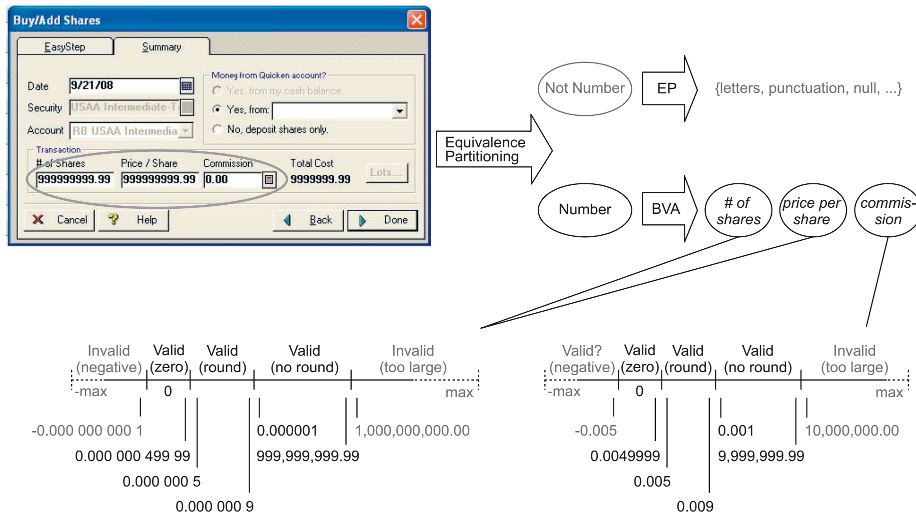


Figure 5–2 *Boundary testing for accuracy*

As an example, look at figure 5-2. This is a screen that you saw earlier, in chapter 4. Let's take a look at the test designs we might do for Quicken's stock buy/add screen, specifically the number of shares, price per share, and commission fields. First we should investigate to make sure the data types behind the fields are sufficient to hold the range of expected values. Are they adequate to support the precision that the fields require? When there are long calculations, with a lot of partial products, are we likely to lose precision?

We applied equivalence partitioning and boundary value analysis to these fields and identified 13 specific input values for each field. In this case, we would also want to add testing of the total cost field. This is a calculated output field. It is calculated by using the three input fields. As you can see in this slide, there is something not right with the calculation. The combination of maximum number of shares and price per share is not giving us the right result in the total cost field. Or perhaps the number is right, internally, but is overflowing the display space. Either way, we'd report this as a bug.

In testing HELLOCARMS, there are a good number of calculations that are made throughout the process, including different customers with different income patterns (married, co-signer, custodian, power of attorney, etc.), different debt loads, and differing types of wealth. There is a huge variety of different

ISTQB Glossary

suitability: The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives.

suitability testing: The process of testing to determine the suitability of a software product.

combinations that can occur for any given customer. Are all of the combinations going to be tested? Unlikely; we probably do not have the resources to do exhaustive testing.

What we will need to do is try to test a representative number of interesting combinations of offerings, with the number of actual tests commensurate with the risk we perceive and the resources we can expend. Good solid unit testing will be essential to accuracy.

5.2.2 Suitability

Suitability testing is focused on the appropriateness of a set of functions relative to its intended, specific tasks. In other words, given the problems we need to solve, can the system solve them?

Technical test analysts will need to be on familiar terms with the domain well enough to understand the users' needs and wants, and then be able to apply that understanding to the system to verify that what is being built will supply those needs. Use cases, the existing system (if any), and interviews with users will all likely be part of this process.

Like accuracy, this particular quality characteristic will be shared with the test analysts. We would expect that we would share use cases and scenarios and discover useful test oracles with them. Static testing at the low-level design and code phases would be predominately done by technical testers with domain testers being more involved in the requirements and high-level design phases. Once code becomes available, technical test analysts would likely be more involved at the lower levels (unit and integration testing) and test analysts would be more involved at the system test level. Good coordination with them would certainly be critical so there is little overlap of testing efforts.

Ongoing feedback from users would be an important input to make sure suitability errors are rectified in later releases.

ISTQB Glossary

interoperability: The capability of the software product to interact with one or more specified components or systems.

interoperability testing: The process of testing to determine the interoperability of a software product.

5.2.3 Interoperability

Interoperability is defined as the capability of the system to interact with one or more specified systems in all target environments (so there is a certain aspect of portability here, which we will discuss later).

Test analysts will be concerned with the end-to-end testing of scenarios using use cases, pairwise testing, classification trees, and other techniques. The facilitating mechanisms underneath that functionality is what technical test analysts are concerned about. Interoperability testing for technical test analysts will include ensuring the smooth functioning of data transfers between systems and checking that the interfaces between systems integrate smoothly. This testing is most visible during system integration and systems testing, but clearly the foundation will have to be built before that.

To the technical test analyst, all parts of the integration between systems are fair game, including hardware, middleware, firmware, operating systems, network configurations, and so forth. Ideally, by the time we get to system integration, the software between all of the involved systems should be able to work together. This, of course, will not happen by accident. The technical test analyst should expect to spend much time on preparation, researching the different systems and their underlying mechanisms and the environments they are expected to work in.

One measure of interoperability will be how easily the systems fit together. Once again, static testing of the plans for integration will be an important part of preparing for the integration process.

Some of the testing that we would expect to do might include the following items:

- Use of industry-wide communications protocols and standards, including XML, JDBC, RPCs[4], DCOM, etc.

4. XML: eXtensible markup language; JDBC: Java database connectivity; RPC: remote procedure calls

ISTQB Glossary

accessibility testing: Testing to determine the ease by which users with disabilities can use a component or system.

usability: The capability of the software to be understood, learned, used, and attractive to the user when used under specified conditions.

usability testing: Testing to determine the extent to which the software product is understood, easy to learn, easy to operate, and attractive to the users under specified conditions.

- Efficient data transfer between systems, with special emphasis on systems with different data representations and precisions
- The ability to self-configure, that is, for the system to detect and adapt itself to the needs of other systems
- Relative performance when working with other systems

The following list includes some of the interoperability testing we might be expected to do on the HELLOCARMS project:

- Ensuring that the communications protocols between all of the active players are compatible
- Testing to make sure data transfer between the Credit Bureau mainframe and the Scoring mainframe are correct
- Ensuring that each system can go offline and then be restored without data loss
- Testing to ensure that network slowdowns do not cause data loss or corruption
- Ensuring that changes to loan information made by LoDoPS are correctly propagated back to HELLOCARMS

5.2.4 Usability

Usability testing, naturally enough, focuses on the users. This is why many notable usability experts and usability test experts have a background in psychology rather than simply being technologists or domain experts. Knowledge of sociology and ergonomics is also helpful. An understanding of national standards related to accessibility can be important for applications subject to such standards.

Users can vary in terms of their skills, abilities, and disabilities. Something an expert technologist finds easy to understand can be absolutely mystifying to another experienced tester whose background is in psychology. Children tend to be remarkably clever in using technology. One day, Rex put an old laptop with a CD-ROM player in the bedroom of his eldest daughter, Emma. At the time she was five year old. He put some CD-ROMs in there, too, including an encyclopedia. Later that night, his wife was frightened by the sound of a man's voice coming from Emma's room. It turns out Emma had figured out how to enable the setting on the encyclopedia that reads the entries out loud and at random.

These kinds of settings and features—text-to-speech and speech-to-text—can be very useful to the disabled, especially those who have limited hand mobility or who are sight impaired. The hearing impaired or those with cognitive disabilities might need different types of assistive technologies. These all need to be tested.

Ultimately, a usable piece of software is one that is suitable for the users. Therefore, usability testing measures whether the users are effective, efficient, and satisfied with the software. Effectiveness implies that the software enables the users to achieve their goals accurately and completely under expected usage conditions. Efficiency implies that these goals can be achieved in some realistic, reasonable period of time. Satisfaction, in this context, is really the antonym of frustration; in other words, a satisfied user who has effectively and efficiently reach her goals with the system feels that the software was about as helpful as it could have been.

What attributes lead to a satisfied, effective, efficient user? One is understandability, the simplicity or difficulty of figuring out what the software does and why you might need to use it. Another is learnability, the simplicity or difficulty of figuring out how to make the software do what it does. Yet another is operability, the degree of simplicity or difficulty inherent in carrying out certain distinct tasks within the software's feature set. Finally, there is attractiveness, which is the extent to which the software is visibly pleasing, friendly, and inviting to the user.

If we are performing usability testing, as with most other testing, we can have as goals both the detection and removal of defects and the demonstration of conformance or nonconformance to requirements. In usability testing, the detection and removal of defects is sometimes referred to as *formative evalua-*

ISTQB Glossary

heuristic evaluation: A static usability test technique to determine the compliance of a user interface with recognized usability principles (the so-called heuristics).

tion, while the testing of requirements is sometimes referred to as *summative evaluation*.

In usability testing, we want to observe the effect of the actual system on real people, actual end users. (This is not to say that testers are not real people, but rather that we are not really the people who must use the system in our day-to-day lives.) To observe the effects, we need to monitor users interacting with the system under realistic conditions, possibly with video cameras, mock-up offices, and review panels.

Usability testing is sometimes seen as its own level, but it can also be integrated into functional system testing. Since usability testing has a different focus than standard functional testing, you can improve the consistency of the detection and reporting of usability bugs with usability guidelines. These guidelines should apply in all stages of the lifecycle, to encourage developers to build usable products in the first place.

There are three main techniques for usability testing.

The first is called inspection (also known as evaluation or review.). This involves considering the specification and designs from a usability point of view. Like all such reviews, it's an effective and efficient way to find bugs sooner rather than later. You can use actual users for this when you have artifacts like screen shots and mock-ups.

A form of review, a heuristic evaluation, provides for a systematic inspection of a user interface design for usability. It allows us to find usability problems in the design, then resolve them, and then reevaluate. That process continues until we are happy with the design from a usability point of view. Often, a small set of evaluators are selected to evaluate the interface, including evaluation with respect to known and recognized usability principles.

The second form of usability testing is validation of the actual implementation. This can involve running usability test scenarios. Unlike functional test scenarios, which look at the inputs, outputs, and results, usability test scenarios look at various usability attributes, such as speed of learning or operability.

ISTQB Glossary

Software Usability Measurement Inventory (SUMI): A questionnaire-based usability test technique for measuring software quality from the end user's point of view.

Usability test scenarios will often go beyond a typical functional test scenario in that they include pre- and posttest interviews for the users performing the tests. In the pretest interviews, the testers receive instructions and guidelines for running the sessions. The guidelines might include a description of how to run the test, time to allow for tests and even the test steps themselves, how to take notes and log results, and the interview and survey methods that will be used.

There also are syntax tests, which evaluate the interface, what it allows, and what it disallows. And there are semantic tests, which evaluate the meaningfulness of messages and outputs. As you might guess, some of the black-box techniques we've looked at, including use cases, can be helpful here.

A final form of usability tests is surveys and questionnaires. These can be used to gather observations of the users' behavior during interaction with the system in a usability test lab. There are standard and publicly available surveys like Software Usability Measurement Inventory (SUMI) and Website Analysis and Measure-Ment Inventory (WAMMI). Using a public standard allows you to benchmark against other organizations and software. Also, SUMI provides usability metrics, which can measure usability for completion or acceptance criteria.

Table 5–1 *Usability checklist instructions example*

Each item in this checklist pertains to a [usability characteristic] or quality of [the system under test] that influences how effective a very novice user will be in unpacking, assembling, powering on, and configuring software on [it]. This [checklist] is intended to predict an end user's experience with [the system].

This [checklist] consists of four major sections:
- Packaging and hardware (100 points)
- Software installation and configuration (100 points)
- Internet connection and online registration (50 points)
- Software discovery and usage (50 points)

For each section, complete the checklist by choosing the most appropriate answer to each question. To score the section, add up the points corresponding to the selected answers, and record the scores in the summary table at the end of the section.

As an example, in table 5-1 you see some introductory information from a document that described the usability test scenarios for the Internet appliance project we've referred to from time to time.

Notice that we define the goals of the test in the first paragraph.

The next paragraph describes the structure of the test set. Basically, it consists of four major scenarios. The scenarios are weighted, which corresponds to how important the test designer feels each one is. There are then some simple instructions on how to use the checklist.

The final paragraph explains to the user how to score the test.

5.2.5 Usability Test Exercise

Review the HELLOCARMS system requirements document, specifically the usability section. Analyze the risks and create an informal test design for usability testing. The following section contains our solution. Of course, your solution may differ based on your experience with usability testing.

5.2.6 Usability Test Exercise Debrief

Early in his career, Jamie had the chance to briefly work in a usability lab for a large, international corporation. It had specially built rooms with multiple cameras and two-way glass, and several observers watched every breath, twitch, and movement of test subjects while they navigated their software prototypes. After seeing the effort a prosperous company could put forth, Jamie always wondered what it would be like to test usability in a smaller, less-provisioned way. This is a good example.

There are several interesting requirements for HELLOCARMS under usability. We selected 030-020-020 under the learnability attribute and 030-010-020 under understandability.

Non-functional testing often forces us to be more creative in coming up with test designs than functional testing. It is not always simply coming up with input data, expected output data and behaviors, etc. In line with recommendations from ISO 9126, much of our non-functional testing will be static testing or trying to measure metrics after a project has occurred.

Starting with 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.*

Testing for this requirement would be straightforward static testing at first. It would consist of working through the wizard, one screen at a time, and comparing the information presented the users to the requirements and designs actually used. We would check for completeness, correctness, and order of presentation.

Once the system was delivered into beta testing, we would send out questionnaires to Telephone Bankers and our partners asking for information on their first week of using the system.

Specifically, we would target any errors, misunderstandings, or inefficiencies they run into, asking for feedback on upgrades or changes they might like to see in the wizard.

Before each modification project for HELLOCARMS, we would scrutinize all reported defect records looking for evidence of mistakes made by ignorance of the system and make sure our documentation covers those areas correctly.

While going through the wizard, we would keep in mind the understandability requirement, 030-010-020:

> *All screens, instructions, help, and error messages shall be understandable at an eighth grade level.*

We would make sure that little or no difficult domain-specific jargon was thrown in to confuse a Telephone Banker or partner.

Looking at the understandability requirement, there are several different ways to try to determine the grade level required to understand a document. One of the most common ways is the Flesch-Kincaid grade level readability formula (which just so happens to be built into MS Word). According to the Word help file, the formula is as follows:

FKRA = (0.39 x ASL) + (11.8 x ASW) - 15.59

In this formula, FKRA = Flesch-Kincaid reading age, ASL = Average sentence length, and ASW = average number of syllables per word.

To test the help documents, wizard, and other source documents that will be surfaced to the HELLOCARMS users, we would copy and paste them into MS Word (if they are not already there), perform a spell check, and then view the statistics. Any resulting value below 9.0 we would consider acceptable (9.0 being the upper boundary of ninth grade reading level).

Our questionnaires, mentioned earlier, would contain questions asking if the help files and the training wizard were clear and understandable. As before, we would also data mine the defect tracking tool before any SMLC (software maintenance life cycle) project started.

5.3 Quality Attributes for Technical Testing

Learning objectives

(K2) Characterize non-functional test types for technical testing by typical defects to be targeted (attacked), their typical application within the application lifecycle, and test techniques suited to be used for test design.

(K2) Understand and explain the stages in an application's lifecycle where security, reliability, and efficiency tests may be applied (including their corresponding ISO 9126 sub-attributes).

(K2) Distinguish between the types of faults found by security, reliability, and efficiency tests, (including their corresponding ISO 9126 subattributes).

(K2) Characterize testing approaches for security, reliability, and efficiency quality attributes and their corresponding ISO 9126 subattributes.

(K3) Specify test cases for security, reliability, and efficiency quality attributes and their corresponding ISO 9126 subattributes.

(K2) Understand and explain the reasons for including maintainability, portability, and accessibility tests in a testing strategy.

(K3) Specify test cases for maintainability and portability types of non-functional tests.

ISTQB Glossary

security: Attributes of a software product that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs and data.

security testing: Testing to determine the security of the software product.

For the remainder of this chapter, we will be discussing non-functional quality characteristics as defined by ISO 9126. Remember from the Foundation level that functional testing deals with what the system does while non-functional testing deals with how or how well the system does something.

Before we address non-functional testing, however, we are going to address technical security testing (*Advanced Software Testing, Vol. 1*, addresses functional security testing). Then we will work our way down the ISO 9126 standard discussing reliability, efficiency, maintainability, and portability. Usability was addressed earlier.

Table 5–2 *ISO 9126 categories and subcategories*

Functionality: Suitability, accuracy, interoperability, security, compliance
Reliability: Maturity (robustness), fault tolerance, recoverability, compliance
Usability: Understandability, learnability, operability, attractiveness, compliance
Efficiency: Time behavior, resource utilization, compliance
Maintainability: Analyzability, changeability, stability, testability, compliance
Portability: Adaptability, installability, coexistence, replaceability, compliance

Table 5-2 shows the elements of the ISO 9126 quality model that we saw before. Reliability, usability, efficiency, maintainability and portability are the quality attributes that technical test analysts are mostly interested in. First, however, we will take a look at the technical side of security testing.

5.3.1 Technical Security

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 have to put special emphasis on testing for them.

Typically, other types of failures have symptoms that we can find, either through manual testing or the use of tools. We know when a calculation fails; the erroneous value is patently obvious. Security issues often have no symp-

toms, 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 exits 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 because 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 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.

5.3.2 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, Vol. 1*, and we will not be duplicating those. Here are the security topics we will cover:

- 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

As we get into this section, 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. 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.

Piracy

There are a lot of ways an intruder may get unauthorized access to data.

SQL injection is a hacker technique that causes a system to run a Structured Query Language (SQL) query where it is not expected. Buffer overflow bugs, which we will discuss in the next section, may allow this, but so might taking 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 good targets for information theft. It is often not hard to guess them; 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, birthday, dog's name, or even just the word *password*. And when forced to come up with a hard-to-remember password, they write it down. It might be found underneath the keyboard or in the 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.[5] 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 system administrator's desk, or in the hallway are all locations we have seen, providing access to whoever happens to be passing by.

Temporary files can be a target if they are unencrypted. Even the data in EXE and DLL files can 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

5. http://microsoft-news.tmcnet.com/microsoft/articles/81726-microsoft-study-reveals-that-regular-password-changes-useless.htm

database management system (DBMS). Strong password control processes should be mandatory with testing regularly occurring to make sure they are enforced. All data in the binaries and all data files should be encrypted. Temporary files should be scrambled and deleted after use.

Buffer Overflow

It seems like every time anyone turns on their computer nowadays, they are reminded 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 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, called a stack frame, is allocated. Various forms of data are stored there, including all local 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 (malicious code insertion). 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.

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 on. 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.[6]

6. Just one example: http://defensetech.org/2008/08/13/cyber-war-2-0-russia-v-georgia/

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.

A variation on this is hacking a single HTTP request 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 latter type of attack. 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.

Anytime you have a data transfer between systems, this is 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[7]. 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.

Data Transfer Interception

The Internet consists of multiple computers transferring data to each other. Many of the IP packets that a computer is passing back and forth are not meant for that computer itself; they are 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. An unscrupulous organization might actually save off 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.

7. http://security.freebsd.org/advisories/FreeBSD-SA-00:53.catopen.asc

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 contains some kind of encryption. However, it is better and more secure than HTTP.

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 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 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 ever 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."[8]

Logic Bombs/Viruses/Worms

Finally, in this short list of possible security gotchas are 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 U.S, defense contractor General Dynamics was arrested for inserting into a system a logic bomb 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.[9]

8. A quote we have heard enough times to scare us!
9. http://www.gfi.com/blog/insidious-threats-logical-bomb/

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 and 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 suggest that the best strategy is to have good antivirus software installed on all systems and a strict policy of standards and guidelines for all users to prevent the possibility of infection.

To prevent logic bombs, all new and changed code should be subjected to some level of static review.

5.3.3 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 it 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.[10] 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.[11] Far too often, damage is done by hackers because no one thought of looking to see where the known vulnerabilities were.

---

10. http://www.sans.org/reading_room/whitepapers/windows/microsoft-windows-security-patches_273

11. Of course, this won't help your organization if you are targeted by someone who invents the hack.

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-327	Use of a Broken or Risky Cryptographic Algorithm
[25]	138	CWE-362	Race Condition

Figure 5–3 *List of top 25 security vulnerabilities from CVE website*

Figure 5-3 shows an example from the *Common Vulnerabilities and Exposures (CVE)* site.[12] This international website is free to use; it is a dictionary of publicly known security vulnerabilities. The goal of this website is to make it easier to share data about common software vulnerabilities that have been found.

The site contains a huge number of resources like the page shown in figure 5-3, 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.

A related website (seen in figure 5-4) is called *Common Attack Pattern Enumeration and Classification* (CAPEC).[13] It is designed to not only name common problems, but to give developers and testers information to detect and fight against different attacks. From the site's "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*

12. cve.mitre.org
13. capec.mitre.org

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 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.*[14]

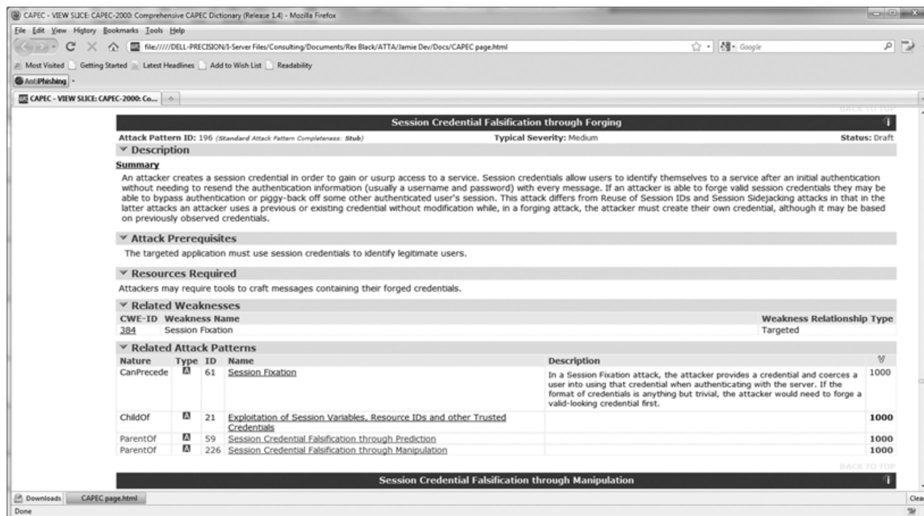


Figure 5–4 *Possible attack example from CAPEC site*

This site is sponsored by the Department of Homeland Security as part of the Software Assurance strategic initiative of the National Cyber Security Division of the U.S. government. 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 good website for the *Open Web Application Security Project* (OWASP).[15] This not-for-profit website is dedicated to improving the security

14. http://capec.mitre.org/about/index.html
15. OWASP.org

of application software. This is a wiki-type 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.

Okay, 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, with multiple reviews, walk-throughs, and inspections at each phase of the development lifecycle. 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 problems before going live.

What should you look for? Certain areas will be most at risk. Communication protocols are obvious targets, as are encryption methods. The configurations in which the system is going to be used may be germane.

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?

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 them 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?

ISTQB Glossary

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 the behavior of users when interacting with the component or system, and their probabilities of occurrence. A task is logical rather that physical and can be executed over several machines or be executed in noncontiguous time segments.

recoverability: The capability of the software product to reestablish a specified level of performance and recover the data directly affected in case of failure. [ISO 9126]

recoverability testing: The process of testing to determine the recoverability of a software product.

reliability**:** 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. [ISO 9126]

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.

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 but 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 comes only from eternal vigilance and constant testing.

5.3.4 Reliability

While 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 there are specific tests we can run to measure reliability, much of our information will come from evaluating metrics that we collect from other testing.

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 reliability growth model and other mathematical elements of reliability testing should be tuned with empirical data; otherwise, the results are meaningless.

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 lifecycle (SDLC); the more mature the system, the closer it is to being ready to move on to the next development phase.

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 must be added.

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 test error and exception handling with interface values that are erroneous, including out of range, poorly formed, or semantically incorrect. During integration test, we test incorrect inputs from user interface,

files, and devices. During system test, we might test incorrect inputs from OS, interoperating systems, devices, and user input. Valuable testing techniques include invalid boundary testing, exploratory testing, state transition testing (especially looking for invalid transitions), and attacks.

The last subcharacteristic 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 measuring the time 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.

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.

Finally, *availability* is not a formal subcharacteristic of reliability. It is closely related, however. It is defined as the capability of the system to be in a state to perform required functions at a given time and under stated conditions of use. This can be said to consist of a combination of maturity, fault tolerance, and recoverability (how long the system will be down after a failure).

It is interesting to compare 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.

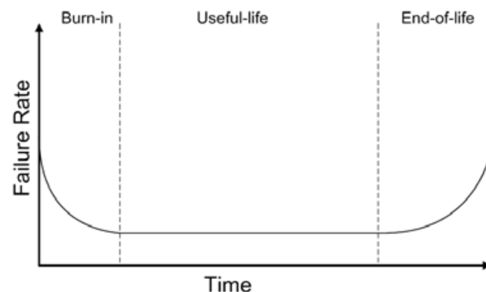


Figure 5–5 *Hardware reliability graph*

In figure 5-5, you see a graph that shows—in general—the reliability of hardware over time. Early in the development life cycle, there are a good number of failures that must be fixed. By the time the hardware is put into production, there are a low number of failures throughout its useful life. As the hardware starts approaching the end of its useful life, the failure rate starts to climb again. With hardware, many failures occur early during the burn-in period. If the hardware does not fail within the first *n* days, it often never will throughout its useful life.

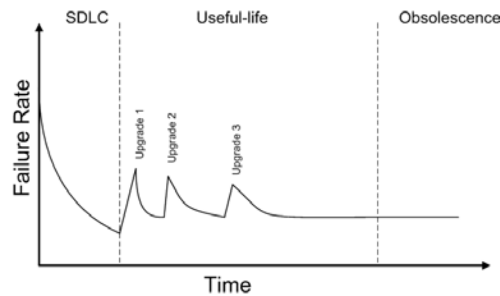


Figure 5–6 *Software reliability graph*

Compare that with the software reliability graph of figure 5-6. Software has a relatively high failure rate during the testing and debugging period; during the SDLC, the failure rate will trend downward (we hope!). While hardware tends to be stable over its useful life, however, software tends to start becoming obsolescent 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. Others might note that there are always features missing that require upgrades and enhancements. Whatever the truth of the argument, 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 5-6, as we get upgrades, complexity also tends to increase, which can lower reliability somewhat.

In reliability 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. During reliability testing, as we find bugs that are repaired, we'd expect the reliability to improve. Var-

ious mathematical models, called software reliability growth models, can be used to monitor this growth.

Defect density—how many defects per thousand lines of source code (KLOC) or per function point—is one metric that has been used for measuring reliability. Cyclomatic 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 number of failures.

Object-oriented development comes with 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.

While there are techniques for accelerated reliability testing for hardware—so-called Highly Accelerated Life Tests (or HALT testing)—software reliability tests usually involve extended duration testing. The tests can be a small set of prescripted 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, which is called stochastic testing. (For example, telephone switches are tested this way, as 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 at best.

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.

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.

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 in 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 test, we need to allow for the artificiality of our testing when trying to predict the future.

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)?

The National Aeronautics and Space Administration (NASA) is responsible for trying to keep mankind in space. When their 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 meets the following requirements:

- Defines the requirements for software controlled system fault/failure detection, isolation, and recovery
- Reviews the software development processes and products for software error prevention and/or reduced functionality states
- 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:

- Error seeding: 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.
- 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.

- 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.
- 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 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.

Static measures include the following:

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 number of problem reports.

The question that must be asked when discussing reliability testing is, Which faults are we going to be concerned with? A complex system can fail at hundreds or thousands of places. Since there is probably a great deal of cost involved, and as reliability needs go up, the costs escalate even faster, so an organization must plan carefully. While NASA can afford to pull out all stops when it comes to reliability testing, most other organizations are not so lucky.

Here are some events that are often of concern:

1. An external event that should occur does not, a device that should be online is not, an interface or process that the system needs is not available.
2. The network is slow or not available or suddenly crashes.

ISTQB Glossary

efficiency: The capability of the software product to provide appropriate performance, relative to the amount of resources used under stated conditions. [ISO 9126]

efficiency testing: The process of testing to determine the efficiency of a software product.

3. The operating system capabilities that the system relies on are not available or degraded.
4. User input is inappropriate, unexpected, or incorrect.

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.

5.3.5 Efficiency

ISO 9126 defines *efficiency* as the capability of the software product to provide appropriate performance relative to the amount of resources used and 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 the new system, the tester laughed. He claimed that on a good day, there 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.

ISO identifies two important subcharacteristics for efficiency: *time behavior* and *resource utilization*.

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-sales systems should have good time response to keep users happy.

In addition, for some 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.[16]

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 (or even 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 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 do performance testing only 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 lifecycle.

Last is 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 danger-

---

16. 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.

ously false. We will discuss all of the tasks that must be done for good performance testing before we ever turn on a tool.

5.3.6 Multiple Flavors of Efficiency Testing

There is an urban myth that the native Inuit peoples have more than 30 different names for snow, based on nuances in snow that they can see. While researching this story, we found that there is wide dispute as to whether this is myth or provable fact. 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, but 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, etc. Show them to an Inuit; they might fail to see any big distinction between them.

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. What follows is 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.[17] 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.*

17. *The Software Test Engineer's Handbook*

We really like Rex's definition better:

> *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 throughput. Some people distinguish between multi-user 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 spike testing is to verify a system's stability during a burst of concurrent user and/or system activity to varying degrees of load over varying time periods. Here are some examples of business situations that this type of test looks to verify a system against:
  - A fire alarm goes off in a major business center and all employees evacuate. The first 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.

And 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 previously mentioned tests 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.

No matter which of these we run, there is much more to creating a performance test that's meaningful than buying a really expensive tool with a 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 raft of bugs when you do test (usually the case), 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 failures. The latter is what we have mostly seen occur.

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 collection of units (builds or backbones) 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, the performance of the whole system in production should be demonstrated (after making sure it is going to work with earlier testing, right?).

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.

5.3.7 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 and 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?

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, there is not a powerful enough calculator made to get a meaningful extrapolation of what our testing is telling us.

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 (for example, 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 will 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.[18] This methodology consists of seven steps, as follows:

1. 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. Balanced against that is the cost. Replicating the environment

---

18. Performance Testing Guidance for Web Applications, http://msdn.microsoft.com/en-us/library/bb924375.aspx

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 in which we have worked, 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 and changes are made. As with risk analysis, which we discussed in chapter 3, we need to always be reevaluating the future based on what we discover during the process.

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's concern
   - Resource utilization: system concerns

   Identify system configurations that might result in the most desirable combinations of the preceding items. 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 manager 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 metric that is possible. Settle on a given set of measurements that are meaningful to proving success—or disproving it.

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 preceding designs into different models of system usage to be tested.
4. Configure the test environment. Prepare the test environment, tools, and resources needed to execute the models designed in step 3. 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.

   Create a schedule that outlines 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.
5. 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 custo-*

diet ipsos custodes?[19] 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 if the results make sense. Do not report the results of the smoke test as part of the official test results.

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 was 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.[20]

   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 factors that might be missed if tests are run only once. When you stop getting valuable information, you have run the tests enough.

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

---

19. *Satires of Juvenal.* Probably not talking about software, but still worth considering...

20. 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.

really cool; however, the stakeholders are more interested in whether the system will support their business goals.

Tune the system. Can you make changes to the system to positively affect its performance. 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 Microsoft 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."

In the real world, of course, you are likely to run out of time long before you get to this point. Remember that testing has to be aligned with the needs of the project. Don't let the perfect be the enemy of the good!

5.3.8 Efficiency Measurements

When we discussed reliability, some of the internal measurements were kind of soft. We could use math and statistics, and wave a magic wand, but the sad fact is many of the metrics were guesses.

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.

There are two main categories of metrics that are interesting in efficiency testing; not surprisingly, these are also the subcharacteristics that we mentioned earlier. One is time behavior, measurements that look at the amount of time it takes to do something. And the second is resource utilization, measurements of actual or projected resource usage that it takes to perform a task. And, in some cases, a third subcharacteristic may be important, efficiency compliance that references any applicable laws, standards or guidelines that the project must be concerned with.

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:[21]

- 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.
- Number of context switches per second.
- Length on queues (processor, disk, etc.) at any given time.
- Disk saturation and usage.
- Network errors, both inbound and outbound.
- Network packet round-trip time.
- Client data presentation time: How long does it take from the time a person clicks "go" until they see a result.

Remember, too, that many measurements waste time and resources, but too few and you don't learn what you need to know. Using metrics is both a science and an art, and they are perhaps the most frustrating thing we deal with when testing.

21. These come from a book by Ian Molyneaux, *The Art of Application Performance Testing.*

5.3.9 Examples of Efficiency Bugs

Good testing methodology tells us that when we are testing, we should have some kind of idea of what we are looking for. So in this section, we are going to discuss four separate scenarios that we may discover 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 5-7, 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.

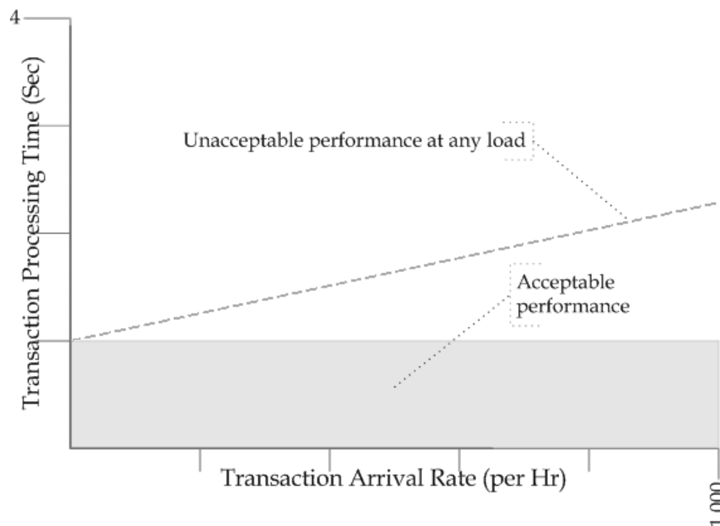


Figure 5–7 *Unacceptable performance at any load example*

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.

In figure 5-7, 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.

A bad database design and implementation where trying to access data just takes too long may be causing this. 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.

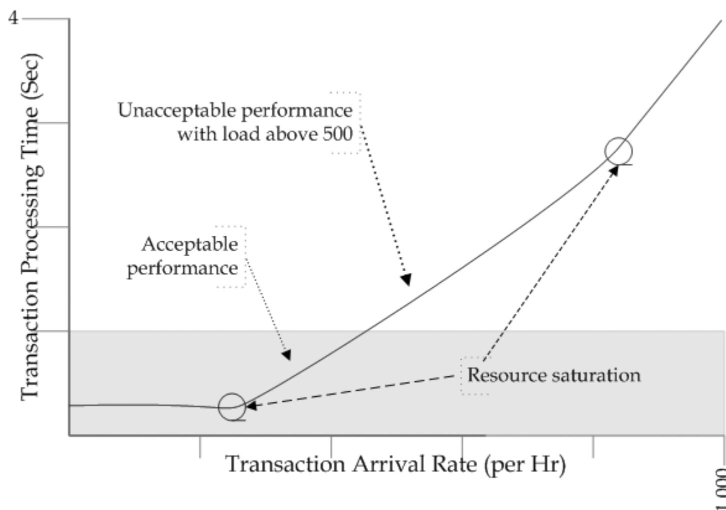


Figure 5–8 *Slow response under moderate loading example*

In figure 5-8, 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.

In figure 5-9, we show several curves. The first, solid green 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 that the previous run, and the third run was worse yet.

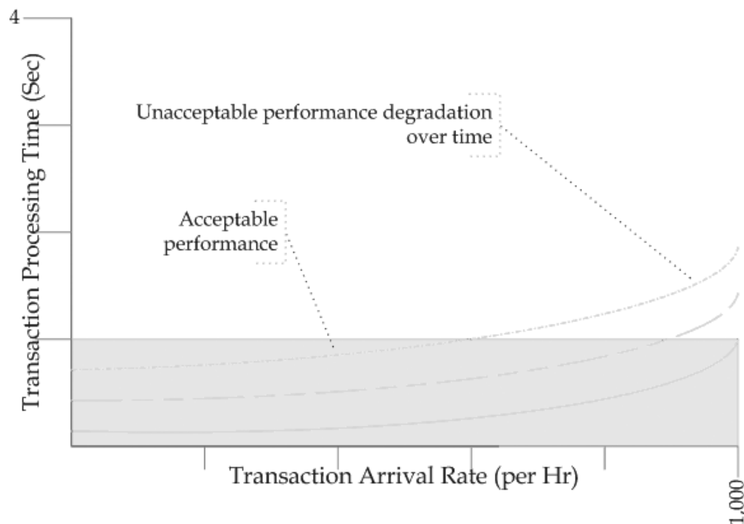


Figure 5–9 *Response that degrades over time example*

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.

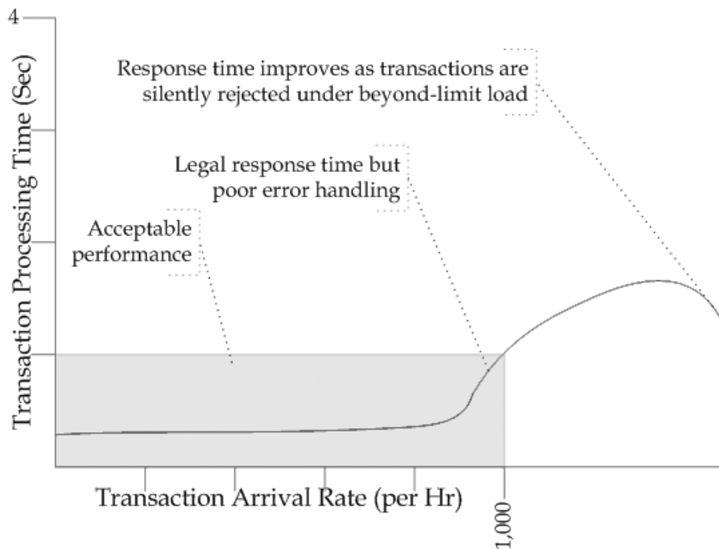


Figure 5–10 *Inadequate error handling example*

Finally, in figure 5-10, 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 were 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 the system 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 with 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 9.

5.3.10 Exercise: Security, Reliability, and Efficiency

Using the HELLOCARMS system requirements document, analyze the risks and create an informal test design for each of the following using one requirement for each:

- Security
- Reliability
- Efficiency

Our results are shown in the next section.

5.3.11 Exercise: Security, Reliability, and Efficiency Debrief

Security

For security, we picked 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), 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 we were active in static testing at every level as the design and code were being developed.

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
- Authentication and session management functions to make sure unauthorized users are not allowed to log in
- 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)
- Making sure all certificates are tested correctly to avoid corrupted or invalid certificate acceptance
- Testing any links on our pages to ensure that we only use trusted data in our links (to avoid getting a reputation for forwarding our customers to malware sites)

Reliability

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 per month in production be fewer 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 a dedicated workstation running for weeks).

This would depend on having automated tests available that exercise the GUI screens of the Telephone Banker. 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:

- 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 vs. 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 build's 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.

Efficiency

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-

ISTQB Glossary

maintainability: The ease with which a software product can be modified to correct defects, modified to meet new requirements, modified to make future maintenance easier, or adapted to a changed environment.

maintainability testing: The process of testing to determine the maintainability of a software product.

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 1-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, cancelling them out. Even so, because the times we are testing are relatively large (1 second and 2 seconds), we believe the testing would likely be valid.

5.3.12 Maintainability

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 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 Arnold Schwarzenegger said in *The Terminator*, "I'll be back."

In the ISTQB Foundation 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 lifecycle.

Okay, quick! Just off the top of your head come up with a dynamic maintainability test for HELLOCARMS. We'll wait. Hmmmmmmm.

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 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 the pig's ear is not going to be magically transformed into a silk purse by a new tool, a couple of scripted tests, or a reasonably attentive tester.

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 being made, 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 maintenance 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 of this, but also the procedures and infrastructure involved. For example, verifying that you can patch a system with a 100 MB patch is all well and good until you find you have forgotten that real users will have to download this patch through a dial-up 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, be serious, we don't have time to think about that. 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, in 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 be hardwired for this short-term/long-term calculation.

> *If I eat this cookie now, I intellectually know that I will have to work out—some time later—much harder that I like to. But what the heck—the cookie looks so good right now.*[22]

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 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 lowballed 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.

---

22. An amusing take on this, the "marshmallow test," can be found at http://www.newyorker.com/reporting/2009/05/18/090518fa_fact_lehrer?currentPage=1.

Lack of ownership is clearly a problem. Maintainability, as well as quality, should be owned by the team as a whole, but it 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 called that short-timer thinking and used to see it in the United States military during the '60s and '70s when the military draft was the norm rather than voluntary service.

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.

5.3.13 Subcharacteristics of Maintainability

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, and testability.

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 in the software 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. 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 understand-

able. 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 minicomputer (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, we were told, that when a programmer needed a particular class, he was likely to search through the library for fewer than 10 minutes before giving up and writing his own class. What made this confusing was that writing his own class might take several days, and at that point, he would have a class where the code was not yet debugged. Had he searched for a little more time, he likely would have found a completely debugged module that supplied the functionality he needed. Management termed this behavior the "not created here" problem.

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 document floating around. Whatever the reason, good documentation helps us understand and helps 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 her own style, unless she is told to follow some kind of standards. That might include the following standards:

- Indentation and other structure guidelines
- Naming conventions
- Modular guidelines (At the company mentioned earlier, the 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 super-classes, 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 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 she understands how we originally made it. Now, changing our code is likely to break her 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.

The second subcharacteristic for maintainability is called *changeability*. The definition in ISO 9126 is the capability of the software product to enable the implementation of a specified modification. 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 each other during execution. High coupling means that there are a lot of dependencies and shared code between modules. When that happens, 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: One module accesses local data in another.
- Common coupling: Two or more modules share global data.
- External coupling: Two or more modules share an external interface or device.
- Control coupling: One module tells another module what to do.
- Data-structure coupling: Multiple modules share a data structure, each using only part of it.
- Data coupling: Data is passed from one module to another, often via parameters.
- Message coupling: Modules are not dependent on each other and pass messages without data back and forth.
- No coupling: 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 generator. Using named constants is 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 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 de 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 capability 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 cost.

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. To make this system work, they had to throw out all of the rules. For this module, 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.

And finally, poor documentation will adversely affect changeability. When a maintenance programmer must guess at how to make changes, to infer as to what the original programmer was thinking, it can adversely affect changeability greatly. Documentation includes internal (comments in the code, self-documenting code, good naming conventions, etc.) and external documentation, including high- and low-level design documents.

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; 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, 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.

Last, there is the subcharacteristic of *testability.* This 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!" 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 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.

ISTQB Glossary

portability: The ease with which the software product can be transferred from one hardware or software environment to another.

portability testing: The process of testing to determine the portability of a software product.

5.3.14 Portability

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, cohabitating 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 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 de-installation are classic portability problems on the Windows platform.

Fortunately, portability defects are amenable to straightforward test design techniques like pairwise testing, classification trees,[23] 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

23. Pairwise testing and classification trees are discussed in *Advanced Software Testing, Vol. 1.*

compromises that were made to try to make the system portable if there was 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 type of hardware. 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: The capability to be adapted for different specified environments without applying actions other than those provided for that purpose.
- Replaceability: The capability to be used in place of another specified software product for the same purpose in the same environment.
- Installability: The capability to be installed in a specific environment. We will include uninstallability in this category.
- Coexistence: The capability to coexist with other independent software in a common environment sharing common resources.

As we mentioned earlier, 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. The catalog she ordered it from 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 huge—they thought of 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 program-

ming languages—such as Java—that you 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 anything for nothing. Adaptability comes at a price: more design work, more complexity, more code bloat, and with those, more defects. So, when your organization is looking into designing adaptability, make sure you know the targeted environments and the business case.

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 the procedures as well as the adaptability of the software.

Replaceability testing is 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 the local processor. 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 Microsoft Office 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 your 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 book about advanced test management, *Advanced Software Testing, Vol. 2*.

Installability is the capability of a system to be installed into a specific environment. Testers have to consider uninstallability at the same time.

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 gotchas 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 coinstalled 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. 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.
- 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.
- 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.

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, they also require some amount of functionality testing afterward to detect any problems that might have been silently introduced. At the end of the day, the most important question to ask is, When we are all done installing, will it work?

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? All the interoperating systems working okay?

At the beginning of discussing installability, we said it was a good news, bad news scenario, the good news being that install was conceptually straightforward. We lied. It's not. The best way we know to deal with install testing is to make sure 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 seen that done and don't believe it can work. 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. Until our tools and methodologies get a whole lot better, we think we will be doing 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 IBM AS/400 host module, a custom ODBC driver, and a full Windows application. They were responsible for testing everything that they 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! They 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.

Finally, we need to discuss *coexistence* testing, which is also called sociability or compatibility testing. Here, 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 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 siloed organizations 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.

By the way, the ISTQB Advanced syllabus mentions that compatibility testing is normally performed when system and user acceptance testing have been successfully completed. This is a good idea only if you don't care about nasty surprises at the end of a project. Seriously, coexistence testing should occur no later than system test.

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.

5.3.15 Maintainability and Portability Exercise

Using the HELLOCARMS system requirements document, analyze the risks and create an informal test design for each of the following using one requirement for each:

- Maintainability
- Portability

The debrief follows.

5.3.16 Maintainability and Portability Exercise 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, etc. 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[24] and our own internal checklists 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 we would 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 (changeability)? 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)?

Portability testing consists of adaptability, installability, coexistence, and replaceability subattributes. Because HELLOCARMS is surfaced on browsers,

24. Discussed in chapter 6.

we find the compelling attribute to be adaptability. Therefore, we have selected requirement 060-010-030 for discussion. It reads as follows:

> *HELLOCARMS shall be configured to work with all popular browsers that represent 5 percent 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 3, 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 need only the latest two versions.

This particular requirement is not enforced until release 3. 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 5 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. 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.[25] 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 browser/operating system speeds that represented most of our prospective users.

---

25. Interested readers can find this technique in *Advanced Software Testing, Vol. 1*, or at the website pairwise.org.

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.

5.4 Sample Exam Questions

1. You have been asked to research testing interoperability on the system currently under development. It consists of a number of COTS packages that will be used to process insurance payments through a number of existing systems. Which of the following capabilities are you likely to be trying to test?

   A. Validating the methods used to manipulate the data used by the system

   B. The computational accuracy of each individual stage

   C. Ability of the software to self-configure communications

   D. Ability of users to achieve specified goals using all of the modules

2. You are new to the organization and have been placed in a technical testing role. You're asked to investigate a number of complaints from customers who have made mistakes using the system in places that were not predicted. You have been tasked with trying to find a way of avoiding these kinds of errors in the future. Which of the following artifacts are you not liable to use while performing this task?

   A. Heuristic evaluation

   B. CAPEC

   C. SUMI

   D. WAMMI

3. Since releasing the latest version two weeks ago, your software system has been broken into at least 10 different times. So far no important customer data has been stolen, but it is only a matter of time. You have been tasked

with determining as quickly as possible if there are more vulnerabilities so a quick patch can be sent out. Which of the following test design techniques would most likely give you the information you need?

A. MC/DC

B. Failure-based taxonomy

C. Chartered exploratory testing

D. Software attacks

4. Non-functional 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

5. Non-functional 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. 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. For which of the following non-functional attributes would you most likely be interested in testing and collecting metrics?

   A. Adaptability

   B. Portability compliance

   C. Coexistence

   D. Stability

6. 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

7. 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 that 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

6 Reviews

"Anything becomes interesting if you look at it long enough."

Gustave Flaubert

The sixth 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. 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. The Advanced syllabus introduces additional types of reviews and covers strategies for effective and successful reviews. Chapter 6 of the Advanced syllabus has five sections.

1. Introduction
2. The Principles of Reviews
3. Types of Reviews
4. Introducing Reviews
5. Success Factors for Reviews

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

6.1 Introduction

Learning objectives

Recall of content only

Again, think back to the beginning of chapter 4. We introduced a taxonomy for tests, shown in figure 6-1. We mentioned 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 4 and 5, we talked about test techniques and quality characteristics, mostly from the point of view of dynamic testing.

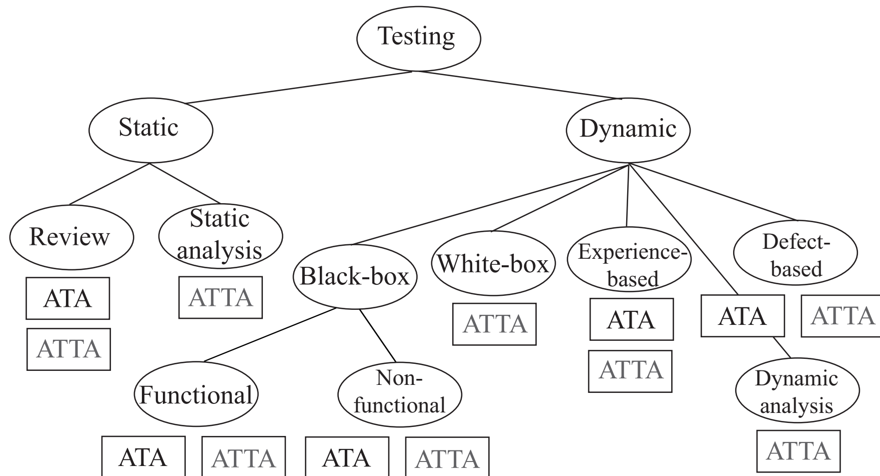


Figure 6–1 *Advanced syllabus testing techniques*

In this chapter, we cover static testing. In fact, we are going to focus on one branch of the static test tree, that of reviews. (We covered static analysis earlier in chapter 4.) Largely, the material in this chapter expands upon what was covered on reviews in the Foundation syllabus.

To have success with reviews, an organization must invest in and ensure good planning, 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). 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

ISTQB Glossary

review: An evaluation of a product or project status to ascertain discrepancies from planned results and to recommend improvements. Examples include management review, informal review, technical review, inspection, and walk-through.

reviewer: The person involved in the review that identifies and describes anomalies in the product or project under review. Reviewers can be chosen to represent different viewpoints and roles in the review process.

negative, confrontational, or belligerent participant can damage the entire process profoundly.

Reviews are easy to do poorly and hard to do well, so many organizations abandon them. However, done properly, reviews have one of the highest payoff rates of any quality-related activity.

1. Overview
   - Purpose, scope, conformance, organization, application
2. References
3. Definitions
4. Management reviews
   - Responsibilities, inputs/outputs, entry/exit criteria, procedures
5. Technical reviews
   - Responsibilities, inputs/outputs, entry/exit criteria, procedures
6. Inspections
   - Responsibilities, inputs/outputs, entry/exit criteria, procedures, data collection, process improvement
7. Walk-throughs
   - Responsibilities, inputs/outputs, entry/exit criteria, procedures, data collection, process improvement
8. Audits
   - Responsibilities, inputs/outputs, entry/exit criteria, procedures

Figure 6–2 *IEEE 1028 standard for software reviews*

Let's review the IEEE 1028 standard for reviews, shown in figure 6-2, which was introduced at the Foundation level. The first section of the standard is an overview. It covers the purpose of the standard, the scope of coverage, and guidelines for conformance with the standard, the organization of the standard, and how to apply the standard in an organization.

The second section is "References," which, as you might imagine, refers to other documents, standards, and so forth. The third section, "Definitions," defines terms used in the standard.

The fourth section addresses management reviews. Management reviews were out of scope at the Foundation level, you might recall. In this section, the standard talks about who has what responsibilities in a management review, the inputs to and outputs from a management review, the entry criteria to start such a review and the exit criteria to recognize when it's complete, and the procedures a management review should follow.

The fifth section addresses technical reviews. In this section, the standard talks about who has what responsibilities in a technical review, the inputs to and outputs from a technical review, the entry criteria to start such a review and the exit criteria to recognize when it's complete, and the procedures a technical review should follow.

The sixth section addresses inspections. As with the previous two sections, the standard talks about who has what responsibilities in an inspection, the inputs to and outputs from an inspection, the entry criteria to start an inspection and the exit criteria to recognize when it's complete, and the procedures an inspection should follow. However, because inspections are more formal than technical reviews, the standard also discusses collecting data from the inspection process and implementing improvements to the inspection process.

The seventh section addresses walk-throughs. As with the other sections, the standard talks about who has what responsibilities in a walk-through, the inputs to and outputs from a walk-through, the entry criteria to start a walk-through and the exit criteria to recognize when it's complete, and the procedures a walk-through should follow. Since the level of formality for walk-throughs is similar to that of inspections, the standard also discusses collecting data from the walk-through process and implementing improvements to the walk-through process.

Finally, the eighth section addresses audits. As with the other sections, the standard talks about who has what responsibilities in an audit, the inputs to and

ISTQB Glossary

audit: An independent evaluation of software products or processes to ascertain compliance to standards, guidelines, specifications, and/or procedures based on objective criteria, including documents that specify (1) the form or content of the products to be produced, (2) the process by which the products shall be produced, and (3) how compliance to standards or guidelines shall be measured.

inspection: A type of peer review that relies on visual examination of documents to detect defects, e.g., violations of development standards and nonconformance to higher-level documentation. The most formal review technique and therefore always based on a documented procedure.

management review: A systematic evaluation of software acquisition, supply, development, operation, or the maintenance process. The management review is performed by or on behalf of management and monitors progress, determines the status of plans and schedules, confirms requirements and their system allocation, and evaluates the effectiveness of management approaches to achieve fitness for purpose.

technical review: A peer group discussion activity that focuses on achieving consensus on the technical approach to be taken. See also *peer review*.

walk-through: A step-by-step presentation by the author of a document in order to gather information and to establish a common understanding of its content. See also *peer review*.

peer review: A review of a software work product by colleagues of the producer of the product for the purpose of identifying defects and improvements. Examples are inspection, technical review, and walk-through.

outputs from an audit, the entry criteria to start an audit and the exit criteria to recognize when it's complete, and the procedures an audit should follow.

6.2 The Principles of Reviews

Learning objectives

Recall of content only

Let's look at some review principles that were explained in the Foundation syllabus. First, as mentioned, a review is a type of static test. The object being reviewed is not executed or run during the review. Like any test activity, reviews can have various objectives. One common objective is finding defects. Others, typical of all testing, are building confidence that we can proceed with the item under review, reducing risks associated with the item under review, and generating information for 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. Some types of reviews also include suggesting improvements to the document as an objective.

Reviews usually precede dynamic tests. They 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.

Woody Allen, the New York film director, is reported to have once said that "80 percent of success is showing up". That might be true in the film business, but Woody Allen would not be a useful review participant. Reviews require adequate preparation. If you spend no time preparing for a review, expect to add little value during the review meeting.

In fact, you can easily remove value by asking dumb questions that you could have answered on your own had you read the document thoroughly before showing up. You might think that's a harsh statement, especially in light of the management platitude that "there are no dumb questions." Well, sorry, there are plenty of dumb questions. Any question that someone asks in a meeting because of their own failure to prepare, resulting in a whole roomful of people having to watch someone else spend their time educating the ill-prepared attendee on something he should have known when he came in the room, qualifies as a dumb question. In fact, to us, showing up for a review meeting unprepared qualifies as rude and unprofessional behavior, disrespectful of the time of the others in the room.

Because reviews are so effective when done properly, organizations should review all important documents. That includes test documents: test plans, test cases, quality risk analyses, bug reports, test status report, you name it. Our rule

ISTQB Glossary

informal review: A review not based on a formal (documented) procedure.

inspection leader (or moderator): The leader and main person responsible for an inspection or other review process.

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, 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.[1] 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.

During a formal review, there are some essential roles and responsibilities:

- The manager. The manager allocates resources, schedules reviews, and the like. However, the manager might not be allowed to attend based on the review type.
- The moderator or leader: This is the chair of the review meeting.

1. See Capers Jones's book *Software Assessments, Benchmarks, and Best Practices*.

- The author: This is the person who wrote the item under review. A review meeting, done properly, should not be a sad or humiliating experience for the author.
- The reviewers: These are the people who examine the item under review, possibly finding defects in it. Reviewers can play specialized roles based on their expertise or based on some type of defect they should target.
- The scribe or secretary or recorder: This is the person who writes down the findings.

Now, in some types of reviews, roles can be combined. For example, the author, moderator, and secretary can be the same person. In fact, as test manager, when Rex has had the test team review his test plans, he's often been the manager, the author, the moderator, and the secretary.

Some additional roles might be involved, depending on the review. We might involve decision makers or project stakeholders. This is especially true if an ancillary or even primary objective of the review is to build consensus or to disseminate information. In some cases, the stakeholders involved can be customer or user representatives. As an example, Rex did a review of mock-ups for the RBCS website with the marketing team, the outsource web development team, and the company executives. Since RBCS had hired the web development team, the executives were the customers and, for some features, the users.

Certain very formal review types also use a reader, who is the person responsible for reading, paraphrasing, and explaining the item under review.

You can use more than one review type in an organization. For some documents, time is more important than perfection. For example, on our test teams, we apply the "two pairs of eyes" rule to mean that a tester must read another tester's bug report before it can be filed. However, for more visible documents like test plans, we use a walk-through with the entire test team. For critical documents, you can use more than one review type on a single item. For example, when writing this book, Jamie and Rex did an informal review of each other's writing followed by a broader, more formal review as the final book came together.

6.3 Types of Reviews

Learning objectives

(K2) Compare review types with each other and show their relative strengths, weaknesses, and fields of use.

The Foundation syllabus discussed four types of reviews.

- At the lowest level of formality (and, usually, defect removal effectiveness), we find the informal review. This can be as simple as two people, the author and a colleague, discussing a design document over the phone.
- Technical reviews are more formalized, but still not highly formal.
- Walk-throughs are reviews where the author is the moderator and the item under review itself is the agenda. That is, the author leads the review, and in the review, the reviewers go section by section through the item under review.
- Inspections are the most formalized reviews. The roles are well defined. Managers may not attend. The author may be neither moderator nor secretary. A reader is typically involved.

As you can imagine, as the level of formality goes up, the rate of review—the number of pages per hour—goes down.

As a further note on terminology, Rex has heard technical reviews referred to as peer reviews, while Jamie has heard that term applied to any type of review, formal or informal. The ISTQB glossary and the Foundation syllabus say that the term *peer review* applies to more formal reviews only, with the Foundation syllabus adding the note that, in a peer review, the participants are "colleagues at the same organizational level." This terminological stew can constitute a confusing deviation between the ISTQB terminology and common usage; you have to be aware of this for any ISTQB exams.

You should remember that the IEEE 1028 standard and the Foundation syllabus are discussing idealized situations. In real-world practice, it is quite common to find organizations blending the parts they like from each type and discarding parts they don't like. It's also quite common to hear organizations talking about walk-throughs when the approach they use for the walk-through does not adhere to the IEEE 1028 rules.

While at the Foundation level we ignored management reviews and audits from the IEEE 1028 standard, let's fill that gap here. Let's start with management reviews. Common purposes of management reviews are to monitor progress, assess status, and make decisions about some project, system, ongoing activity, or process. (Of course, in some organizations, management reviews are organized for various political reasons too, but we can ignore that for the moment.)

Managers involved with the item being reviewed often perform management reviews. Various stakeholders and decision makers can assist them as well. The level of involvement of each of the participants can vary. In some cases, organizations will hire outside consultants to come in and do reviews. For example, a large portion of RBCS's business is doing test process and quality process assessments of various kinds for organizations. These are a hybrid between a management review and an audit, which we'll discuss in a moment. Test managers often drive these test assessments, in which case they are more like a management review. When outside test stakeholders drive these test assessments, they are more like an audit.

Part of a management review is often to assess how a project is doing in terms of plans, estimates, project risk management, and so forth. Another part of a management review is looking at the adequacy of various procedures and controls. Participants must prepare for these reviews, especially those who are going to deliver status information. We've done test assessments for organizations where people had so tenuous a grasp on what was going on in their testing group that our foremost recommendation was, "Get some metrics and tracking mechanisms in place immediately."

Typically, the outcome of a management review includes action items, recommendations, issues to be resolved, and the like. The decisions should be documented and the execution of action items and recommendations checked regularly. Unfortunately, it's not unusual for follow-up to be less than ideal.

Moving on to audits, these can be quite formal and, in some cases, quite adversarial. In an audit, there's a strong chance that the auditors are measuring people against a contract, a standard, an industry best practice, or the like. This can provoke defensiveness. The Advanced syllabus says that audits are least effective at revealing defects, but that really depends on the auditing and audited organization. When we do testing audits for our clients, we are very, very good at finding defects, both project defects and process defects.

One essential element of an audit is the independent evaluation. As with a management review, we can measure a process, a project, an ongoing activity, or a system. However, another essential element of an audit is the idea of being in or out of compliance. Audits can be done by a lead auditor with an auditor team or by a single auditor. The auditors collect evidence through interviews, witnessing, examining documents, analyzing metrics, and so forth. We find that the interviews can be particularly interesting, especially when an audit has become high stakes for some of the participants. Attempts to spin, mislead, misdirect, convince, and stall the auditor occur in such situations.

As with the management review, the outcome of an audit can include action items, recommendations, issues to be resolved, and the like. However, it also includes an assessment of compliance (or noncompliance). This is often measured against a multidimensional scale or checklist, so even if 99 items are in compliance, if 1 is out of compliance, the organization might fail the audit. If noncompliance is the finding, then corrective actions for the item or items that failed would be typical. Again, the decisions should be documented and the execution of action items, recommendations, and corrective actions should be checked regularly, along with periodic reassessment of compliance. In regulated industries or for legally mandated audits, follow-up on audit results are usually excellent, but in nonregulated industries, follow-up is often less than ideal.

In addition to the types of reviews laid out in the IEEE 1028 standard, we can classify reviews in terms of the work products or activities subject to review. A contractual review, naturally enough, corresponds to some sort of project milestone, often one linked to payment or continuation of a contract. It could also be a management review for a safety-critical or safety-related system. The review could involve managers, vendors, customers, and technical staff. When a project is going well, these are routine. When a project starts to go poorly, particularly if there are multiple vendors involved, expect massive amounts of time and energy to be spent by each vendor trying to obscure who is responsible for the problems.

We can consider requirements reviews. A requirements review can be informal, it can be a walk-through, it can be a technical review, or it can be an inspection. The scope of a requirements review should be whatever it needs to be. If we are building a safety-critical system, a review or part of a review should consider safety. If we are building a high-availability system, a review or part of

a review should consider availability and reliability. For all systems, requirements and the requirements reviews should address both functional and nonfunctional requirements. We can include in a requirements review acceptance criteria and test conditions.

We can also consider design reviews. A design review can range from informal to technical reviews and inspections. Like a requirements review, they can involve technical staff and customers and stakeholders, though you'd expect that as the level of detail becomes more intense, the participants would become more technical. In some organizations, there is a concept of preliminary design review and a critical design review. The preliminary design review is a technical review (also a peer review due to the attendees) where technical peers propose the initial approach to deal with technical issues related to system or test designs. The critical design review covers the proposed design solutions, which can include test cases and procedures in some cases.

The operational readiness review, acceptance review, or qualification review is a combination of technical and managerial review. This is sort of a final safety net. We want to review all the data to make a final decision on readiness for production. As important as this is, Rex has seen situations where the project was so intensive and exhausting that by the time the operational readiness review—or exit meeting or project launch meeting or whatever it was called—occurred, everyone rubber-stamped a decision to go ahead with production even though there were lots of good reasons to say, "Wait, don't do this." In one case, that decision lead to months of extremely poor system performance in production.

As discussed in the Foundation syllabus, there are six phases for a formal review:[2]

1. Planning, including for each work product to be reviewed and for all reviews to occur on a project
2. Kickoff, again for each work product and for all reviews
3. Individual preparation for each work product; reading the document and noting problems
4. The review meeting itself, for each work product

---

2. At the time of writing this book, due to a formatting error, the Foundation 2010 syllabus showed the process as having 12 steps. This problem has been rectified in the 2011 version of the Foundation syllabus.

5. Any rework necessary based on the changes required by the review results
6. Finally, follow-up both for individual work products if needed and for the overall reviews done on the project

Now, remember that good process is important, but the right participants are essential. The participants must match the work product to be reviewed. Inviting the wrong people to reviews guarantees ineffectual reviews, even if you follow the process to the letter.

Capers Jones, in his studies of thousands of projects across hundreds of clients, has found some interesting data on reviews, their applications, and the effectiveness of various types of reviews. Jones mentions that the informal reviews are the least effective, reviews that have some but not all elements of formality are about average, and the most effective are the highly formalized inspections. Of course, to be effective at any level of formality, you have to do the reviews well and you have to have organizational support for the process.[3]

As you can see in table 6-1, both the level of formality and the type of item to which the review is applied has a strong influence on the percentage of defects found and removed. If you think of the reviews as a series of filters—which is a good way to think of all quality assurance and testing activities—here's a quick mathematical demonstration of how effective reviews can be.

First, imagine that you started with 1,000 defects. You follow worst practices in reviews, but at least you review all types of items. In this case, you would enter testing with about 166 defects. Now, imagine that you started with 1,000 defects again. However, this time you follow best practices (and again you review all types of items). This time, you go into testing with 3 defects.

Table 6–1

	Least	Average	Most
Requirements review	20%	30%	50%
High-level design review	30%	40%	60%
Functional design review	30%	45%	65%
Detailed design review	35%	55%	75%
Code review	35%	60%	85%

3. These figures and table 6-1 are derived from *Software Assessments, Benchmarks, and Best Practices* by Capers Jones.

6.4 Introducing Reviews

Learning objectives

(K2) Compare review types with each other and show their relative strengths, weaknesses, and fields of use.

The following steps are useful in successfully introducing reviews:

- Secure management support: Reviews are not expensive from a budget point of view, as test automation is, but they do require a time commitment, especially when time is tight.
- Educate managers: You need to have an honest conversation about the business case for reviews, including the costs, benefits, and potential issues. Avoid exaggerating the benefits because, if your exaggeration is detected later, reviews might be cancelled.
- Put structure in place: Have documented review procedures for the various types of reviews you'll use. Have templates and forms available. Establish an infrastructure such as the reviews metrics database. If you intend to do geographically distributed reviews, make sure you have the tools in place for that.
- Train: Educate the participants on review techniques and procedures.
- Obtain participant support: Make sure those who will do the reviews and those whose work will be reviewed are comfortable and supportive.
- Do some pilot reviews: Expect to make some mistakes—and plan to learn from them.
- Demonstrate the benefit: You have a defined business case, right? Now show management that you achieved what you promised!
- Apply reviews to all (or at least the most important) documents: Requirements, contracts, project plans, test plans, quality risk analyses, and similar high-visibility documents are obvious targets. However, we have found simply ensuring informal reviews of bug reports to be amazingly valuable.

You won't necessarily need to do every step in every organization, and you don't need to do these steps in perfect, sequential order, but you should think long and hard about why it's okay to skip a step if you think it is.

Your organization will invest time and money in reviews. Managers will expect a return on that investment. To demonstrate a return on the review investment, you can use metrics like the reduced or avoided cost of fixing defects or dealing with failures. What does a defect cost in system test? How about after release? A simple spreadsheet can show the benefits of reviews and evaluate the success of the reviews after their implementation.

Don't forget to measure the return in terms of saved time too. Money is not always the biggest concern for managers. In fact, time to market is usually a bigger issue. So, if you can document that a defect takes 5 hours to resolve when found in a review and 25 hours when found in system test, you have a solid business case for how time investment in reviews during the early stages of a project reduces the likelihood of project delay at the end of the project.

Having established metrics, it's important to continue to monitor them. It's easy for review processes to become ritualistic and stuck, and then the value goes down. If you see the benefit dropping off, ask yourself why? In fact, the benefit should constantly be going up. You should be looking for metrics-based, measurable ways to improve the review processes. Make sure that you—and your managers—see reviews and review process improvement as a long-term investment.

6.5 Success Factors for Reviews

Learning objectives

(K4) Outline a review checklist in order to find typical defects to be found with code and architecture review.

A number of factors influence the success—or, if absent, the failure—of reviews. The Advanced syllabus classifies those into three groups. Let's start with the technical factors.

Ensure that you are following the defined process correctly. This can be particularly tricky for formal types of reviews like inspection. Now, that doesn't mean you can't tailor these processes, but it's usually a good idea to master the defined processes, as described in standards or authoritative texts, first.

We mentioned the importance of the business case. To support your business case, you have to record the costs of reviews (particularly in terms of effort) and the benefits that the organization obtains. A problem with reviews is that the benefits accrue long after the cost was incurred. That's true for all testing, of course, but it's especially acute for reviews, particularly if you forget to measure the value.

You don't have to wait until a document is done before you start reviewing it. You can and should review early drafts or partial documents when you're dealing with something critical. This can help to identify and prevent patterns of defects before they are built into the whole document.

That said, make sure you have some rules about what it means for something to be ready for review. You can waste people's time by sending them materials that aren't sufficiently mature to be reviewed. You can also waste people's time and frustrate them by sending them stuff to review that's still changing. People who are frustrated because they are wasting their time on some activity tend to find ways to stop wasting their time on that activity, which means that the review process can wither away. So have some entry criteria. These should also include the simple rule that everyone has to show up prepared.

Checklists are helpful for reviews. It's too easy to forget important areas without them. Have some checklists. You can start with checklists from reputable industry experts, such as the ones included in this book, but make sure to extend those to be organization specific. They should address common defects based on what you find. Also, have different checklists for different kinds of documents, such as requirements, use cases, and designs. Finally, have different checklists for different review processes.

The appropriate level of formality varies. So be ready to use more than one type of review. Consider your objectives. Is the idea to do a quick document cleanup before sending to a client? To improve some technical design decisions? To educate stakeholders? To generate information for management?

We've mentioned the rule of "two pairs of eyes," and we try hard not to violate that rule. Sometimes, deadlines intervene. However, you should review—or, better yet, inspect—all documents that are vitally important. If a document is involved in making an important decision, such as signing a contract, be sure to inspect the proposal, contract, or high-level requirements specification first. If a

major expenditure is being contemplated, have a management review to authorize it.

For large documents, you can use a sampling of a limited subset to estimate the number of defects in the entire document. This can help to determine if a re-review is needed. Keep in mind that this sampling approach won't work for a document cleanup or edit.

Watch out for distractions. It's easy to find a bunch of minor format, spelling, and grammar errors. Focus on finding the most important defects, based on content not format.

Finally, as we mentioned earlier, continuously improve the review process.

Now, some organizational factors.

Make sure managers will plan and estimate for adequate time, especially under deadline pressures. It is a false economy to think that if you skip highly efficient bug removal activities early in the process, somehow the schedule end date will be accelerated, but that kind of thinking is rampant in software engineering.

Be careful with the metrics. For one thing, remember that some reviews will find many defects per person-hour invested, while others won't. There are some mathematical models for predicting defect density, which are beyond the scope of this book. Be careful not to use simplistic models. Most importantly, never ever let review defect metrics be used for individual performance evaluations. That introduces a level of defensiveness that will kill the process.

Make sure to allow time for rework of defects. It's a classic testing worst practice to assume that a test activity will conclude without finding any defects.

Make sure the process involves the right participants. A study by Motorola in 2003 showed that the right participants were the strongest indicator of review success.[4] This includes technical or subject matter expertise, of course. It also includes the issue of balance, making sure the review team has representatives from all key groups. And, it includes understanding the review process, usually through training, especially for formal types of reviews. The second-strongest indicator, they found, was having the right number of participants, so make sure to think carefully about who and how many.

---

4. Jeff Holmes, "Identifying Code-Inspection Improvements Using Statistical Black Belt Techniques," *Software Quality Professional*, December 2003.

If you are in a medium-to-large organization that is using reviews, have a review forum to allow people to share their experience and ideas. This can be reserved for moderators or leaders.

There's no point in having people at a review meeting if they don't contribute. So ensure that the participants participate. Part of this is ensuring proper preparation. Another part is to draw less-vocal participants into the meeting. Just because someone doesn't have a forceful personality doesn't mean they don't have good ideas.

Again, when dealing with critical documents, apply the strongest, most formal techniques. Remember Jones's figures on review effectiveness. What percentage of defects can you afford to leave in each kind of document?

Make sure to have a process in place for review process improvement. If this isn't supported by metrics, it's likely to point you in the wrong direction. Make sure the process for improving the review process includes a mechanism to recognize and celebrate the improvements gained.

Finally, some people issues.

As with managers, educate all stakeholders and participants to expect defects. Make sure that's not an unpleasant surprise to them. Make sure they have allowed for rework and re-review time. People tend to overbook themselves in today's workplace. If they do so, being confronted with a list of issues to resolve in their document is likely to be a traumatic experience because it means overtime.

The review leader is not Torquemada, the Grand Inquisitor of the Spanish Inquisition. The rack, the iron maiden, and waterboarding are not review tools or techniques. Reviews should be a positive experience for authors, where they learn how to do their job better from respected peers. Both Rex and Jamie can still remember review sessions with two or three mentors early in their careers that helped them grow significantly. That said, if authors have had bad experiences, be careful with forcing an author to consent to a review. It's best if management handles this.

Given how efficient defect location and removal is during reviews, we should be happy, not unhappy, when we find defects. Make sure people see that as an opportunity to save time and money. Don't look to point fingers or assign blame when defects are found.

Monitor the dialog in the room. We want constructive, helpful, thoughtful, and objective discussion. Make sure that people are thinking about the deeper issues, including how the document under review fits into the broader picture of the project.[5]

6.5.1 Deutsch's Design Review Checklist

You can and should apply reviews to designs, not just requirements and use cases. 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." They were 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.

We can create a checklist based on Deutsch's observations. The idea of a checklist is to force you to think. When 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 that 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? Why would it need to be so fast anyway? In today's world, you might be

5. For a discussion of reviews from a formal perspective, see Tom Gilb and Dorothy Graham's book *Software Inspection*.

doing some distributed processing. Running web services from...anywhere. RPC 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? These are all things to consider.

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 twenty-first century. As we travel around the country teaching classes, it surprises us to see how many people are still on dial-up connections. So when our fancy distributed application is downloading five megabytes of company logos to our customer's computer through that 40k dial-up, that could create some performance issues.
4. **The network is secure.** If you truly believe that, you might want to go back and review the technical security section in chapter 5. 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, 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 app is in-house or worldwide; the only constant in networks is change. It may not change today, or tomorrow. 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 administrator of the system. Tools will be needed to solve issues with deployment. Where will they get the tools, the training, 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, then the copper or fiber, through to 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?

As you can see, much of your design and implementation for any system you might want to build would require discussion of all these points. Hence a checklist.

6.5.2 Marick's Code Review Checklist

Technical test analysts are likely to be invited to code reviews. 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.[6]

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 operations on data items, Marick says we should ask the following questions:

- Are all strings NULL terminated? If we 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?

---

6. This is drawn from Brian Marick's *The Craft of Software Testing*.

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.

6.5.3 The OpenLaszlo 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.[7]

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?

7. The complete list can be found at http://wiki.openlaszlo.org/Code_Review_Checklist.

Finally, here are the documentation questions included in the OpenLaszlo 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?

Jamie and Rex 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.

6.6 Code Review Exercise

In this exercise, you apply Marick's and the OpenLaszlo code review checklists to the following code shown following the instructions.

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 you 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 present 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.    }
```

6.7 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:

1. No comments
2. 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 explanation for each change.

3. No reasonable naming conventions are followed. I would prefer Hungarian notation so we can discern the data type automatically.[8]
4. 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 too 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?

---

8. *Hungarian notation* is a term that originated at Microsoft, thanks to 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.

- 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 since there is no storage allocated. This is a violation of Marick's allocation question, Is too little (or too 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 we must name the local variables the same as the parameters (considering the way they are used, it makes a little sense), then we should 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 one. 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 one. I would want to know if the user can scale at 5.3 (or any other real number) or if they could use only integers. If they could input only 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 ("/nScale 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, OpenLaszlo'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! I have 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 in the user documentation? No documentation.
  - Does the implementation match the documentation? No documentation.

6.8 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 because 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. 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.

6.9 Deutsch Checklist Review Exercise Debrief

Senior RBCS Associate Jose Mata reviewed an early version of the HELLO-CARMS system requirements document (which did not include the non-functional sections). Jamie Mitchell reviewed the latest version of the document. Both used Deutsch's 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 6-1 does not include any redundancy as in failover servers, backup switches, 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, 040-010-060, 040-010-070 and 040-010-080). Some system-wide latency has been addressed (040-010-040 and 040-010-050). 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), web server (040-020-020) and 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, etc.

  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 marked as 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 under-defined.

- 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.

6.10 Sample Exam Questions

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

4 During the project, your manager schedules a private interview between you and a person from an international accounting firm. This person asks you some questions about the processes used to design, implement, and test the system. What process is this interview most likely part of?

A. Management review

B. Process inspection

C. Regulatory audit

D. Project walk-through

7 Incident Management

"Tell your testers they're finding too many defects."

A real quote from a real project manager, shared with us
by Susan Herrick.

The seventh chapter of the Advanced syllabus is concerned with incident management. As was discussed in the Foundation syllabus, an incident has occurred anytime the actual results of a test and the expected results of that test differ. The Advanced syllabus uses the IEEE 1044 standard to focus on incident lifecycles and the information testers should gather for incident reports. Chapter 7 of the Advanced syllabus has six sections.

1. Introduction
2. When Can a Defect Be Detected?
3. Defect Lifecycle
4. Defect Fields
5. Metrics and Incident Management
6. Communicating Incidents

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

7.1 Introduction

Learning objectives

Recall of content only

Incident management is an essential skill for all testers. Test managers are more concerned with the process. There must be a smooth, timely flow from recognition to investigation to action to disposition. Testers—both technical test analysts and test analysts—are mostly concerned with accurately recording incidents and then carrying out the proper confirmation testing and regression testing during the disposition part of the process.

Testers have a somewhat different emphasis depending on their role. Test analysts compare actual and expected behavior in terms of business and user needs. Technical test analysts evaluate behavior of the software itself and might need to apply further technical insight.

7.2 When Can a Defect Be Detected?

Learning objectives

Recall of content only

We can detect defects through static testing, which can start as soon as we have a draft requirements specification. We can detect failures, being the symptoms of defects, through dynamic testing, which can start as soon as we have an executable unit.

Testing is a filtering activity, so to achieve the highest possible quality, we should have static and dynamic test activities pervasive in the software lifecycle. In addition to filtering out defects, if we have lots of earlier filters like requirements reviews, design reviews, code reviews, code analysis, and the like, we will have early defect detection and removal. That reduces overall costs and reduces the risk of schedule slips.

When we see a failure, we should not automatically assume that this indicates a defect in the system under test. Defects can exist in tests too.

ISTQB Glossary

defect (or bug or fault or problem)**:** A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g., an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system.

error (or mistake)**:** A human action that produces an incorrect result.

failure: Deviation of the component or system from its expected delivery, service, or result.

incident (or deviation)**:** Any event occurring that requires investigation.

incident logging: Recording the details of any incident that occurred, e.g., during testing.

incident report (or deviation report): A document reporting on any event that occurred (e.g., during the testing) that requires investigation.

root cause analysis: An analysis technique aimed at identifying the root causes of defects. By directing corrective measures at root causes, it is hoped that the likelihood of defect recurrence will be minimized.

root cause: A source of a defect such that if it is removed, the occurrence of the defect type is decreased or removed.

7.3 Defect Lifecycle

Learning objectives

(K4) Analyze, classify, and describe functional and non-functional defects in understandable defect reports.

In figure 7-1, you see a diagram that shows the IEEE 1044 incident management lifecycle, including a mapping from IEEE 1044 that shows how typical incident report states in an incident tracking system would fit into this lifecycle. Let's look at this lifecycle.

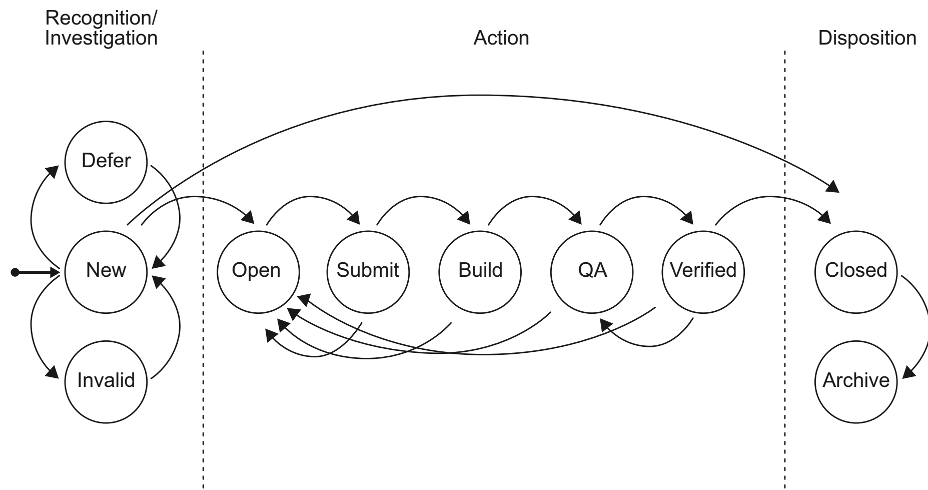


Figure 7–1 *IEEE 1044 incident management lifecycle*

We assume that all incidents will follow some sequence of states in their lifecycle, from initial recognition to ultimate disposition. Not all incidents will travel through the exact same sequence of states, as you can see from figure 7-1. The IEEE 1044 defect lifecycle consists of four steps:

Step 1: Recognition. Recognition occurs when we observe an anomaly, that observation being an incident, which is a potential defect. This can occur in any phase of the software lifecycle.

Step 2: Investigation. After recognition, investigation of the incident occurs. Investigation can reveal related issues. Investigation can propose solutions. One solution is to conclude that the incident does not arise from an actual defect; e.g., it might be a problem in the test data.

Step 3: Action. The results of the investigation trigger the action step. We might decide to resolve the defect. We might want to take action indicated to prevent future similar defects. If the defect is resolved, regression testing and confirmation testing must occur. Any tests that were blocked by the defect can now progress.

Step 4: Disposition. With action concluded, the incident moves to the disposition step. Here we are principally interested in capturing further information and moving the incident into a terminal state.

Of course, what's driving the incidents from one state to another, and thus from one step in the lifecycle to another, is what we learn about the incident. We need the states since defects are handed off from one owner to another owner, so we must capture that learning. Therefore, within each step—and indeed, embedded in each state—are three information capture activities:

- Recording
- Classifying
- Identifying impact

The way this works is shown in Table 7-1. This process of continually refining our understanding of the incident through classification and data gathering is the essence of the IEEE 1044 standard.

Table 7–1 *IEEE 1044 classification process*

Step	Activities		
	Record...	*Classify...*	*Identify impact...*
1. Recognition	Include supporting data	Based on important attributes	Based on perceived impact
2. Investigation	Update and add supporting data	Update and add classification on important attributes	Update based on investigation
3. Action	Add data based on action taken	Add data based on the action taken	Update based on action
4. Disposition	Add data based on disposition	Based on disposition	Update based on disposition

During the recognition step, we will record supporting data. We will classify based on important attributes that we have observed. We will identify impact based on perceived impact, which might differ from the final impact assessment.

During the investigation step, we will update and record more supporting data. We will update and add classification information on importance based on attributes uncovered during the investigation. We will update the impact based on investigation too.

During the action step, we will record new supporting data based on the action taken. We will also add classification data based on the action taken. We will update the impact based on the action too.

ISTQB Glossary

anomaly: Any condition that deviates from expectation based on requirements specifications, design documents, user documents, standards, etc. or from someone's perception or experience. Anomalies may be found during, but not limited to, reviewing, testing, analysis, compilation, or use of software products or applicable documentation. See also *bug*, *defect*, *deviation*, *error*, *fault*, *failure*, *incident*, *problem*.

Finally, during the disposition step, we will record final data based on disposition. The classifications will be adjusted and finalized based on the disposition. The final impact assessment will be captured.

Notice we've been talking about data and classifications. The IEEE 1044 standard includes mandatory and optional supporting data and classifications for each activity in each step. We'll review these in the next few sections. By the way, when we say "mandatory supporting data and classifications," we mean mandatory for IEEE 1044 standard compliance.

Each of these data items and classifications is associated with a step or activity. The IEEE has assigned a two-character code in the standard: RR (recognition), IV (investigation), AC (action), IM (impact identification), and DP (disposition).

We'll go through these data items and classifications. As we do so, don't get lost in the trees and fail to see the forest. The important thing to think about is not—usually—"Is my incident management system IEEE 1044 compliant?" but rather "Might this data or classification be useful to capture?"

The following are the recognition step classifications:

- Project Activity (RR1*nn*): What were you doing when the incident was observed? This is a mandatory field for IEEE 1044 compliance.
- Project Phase (RR2*nn*): What phase was the project in (mandatory)? This will have to be tailored to your lifecycle.
- Suspected Cause (RR3*nn*): What do you think might be the cause (optional)? We've found that, in many cases, capturing this data can be useful for the developers, especially if you have very competent technical testers.

- Repeatability (RR4*nn*): Could you make the incident happen more than once (optional)? We have a problem with IEEE 1044 calling this optional because we think that reproducibility is an absolute must in terms of an incident report.
- Symptom (RR5*nn*): How did the incident manifest itself (mandatory)?
- Product Status (RR6*nn*): What is the usefulness of the product if the incident isn't resolved (optional)? We disagree with the optional category for this one too.

The *nn* characters at the end of the codes indicate that these are hierarchies. Subclassifications exist within each one. For example, each of the project activity choices has a specific code like RR110, RR120, etc. IEEE 1044 defines choices for these as well, but we're not going to review down to that level of detail here.

The following are the recognition step data:

- What environment were you working in when you saw the incident? You should capture product hardware, product software, database, test support software, platform, firmware, and other useful information.
- What origination details can be captured (including who was the tester)? You should capture your name, the date the incident was observed, the code or functional area, the distribution (what is the version of the test object), and contact information like e-mail address, address, phone number, and company ID.
- At what time did you see the incident? This is operating time (i.e., time since last reboot or total uptime), wall clock time, system time, and CPU time.
- What, if any, vendor information applies? This includes company, contact name, vendor ID, expected resolution, and expected resolution date.

The following are the investigation step classifications:

- Actual Cause (IV1*nn*): What really caused the incident (mandatory)?
- Source (IV2*nn*): What was the incident's origin (mandatory)? This question involves the underlying mistake that was made.
- Type (IV3*nn*): What type of defect caused the failure (mandatory)? This is a question of defect taxonomy.

Remember that classifications from previous steps can be updated during this step.

The following are the investigation step data:

- What acknowledgement information can you capture? What data was received, what report number was assigned, who is the investigator, what are the estimated start and end dates of the investigation, when (subsequently) did the actual start and end dates of the investigation occur, how many person-hours were spent, on what date did you receive this acknowledgment, and what documents were used in the investigation?
- What verification information can you capture? What was the source of anomaly (or incident) and how did you verify the data from the recognition process?

Remember that data from previous steps can be updated during this step.

The following are the action step classifications:

- Resolution (AC1*nn*): When and how should the incident be resolved (mandatory)?
- Corrective Action (AC2*nn*): What can be done to prevent such incidents in the future (optional)?

Remember that classifications from previous steps can be updated during this step.

The following are the action step data:

- What resolution identification information can you capture? What test item is to be fixed, what specific component within the item is to be fixed, how can you describe (in text) the fix, when is the planned date for action completion, who is the person assigned, what is the planned date of fix completion, or, if the fix is deferred, where is your reference or authority for that?

ISTQB Glossary

priority: The level of (business) importance assigned to an item, e.g., defect.

severity: The degree of impact that a defect has on the development or operation of a component or system.

- What resolution action information can you capture? What is the date on which it was completed, which organization is assigned to verify resolution, and which person is assigned to verify resolution?

Remember that data from previous steps can be updated during this step.

The following are the disposition step classifications and supporting data:

- Disposition (DP1*nn*): How was the problem finally resolved (mandatory)?
- What anomaly (or incident) disposition information should you capture? What action was implemented, on what date was the report closed, on what date was document updating complete, when was the customer notified, and what reference document numbers might exist?
- What verification information should you capture? What is the name of the person doing the verification, on what date did the verification occur, what version and revision levels were verified, what method did you use to verify, and what is the test case you used to verify?

Remember that classifications and data from previous steps can be updated during this step.

Now, throughout the lifecycle, impact classifications are made and revised. Let's look at some of those impact classifications:

- Severity (IM1*nn*): What is the impact on the system (mandatory)?
- Priority (IM2*nn*): What is the relative importance of the incident (optional)? We disagree with this being optional. In fact, we'd say it's more important than severity in many cases.
- Customer Value (IM3*nn*): How does this incident affect customer(s) or market value (optional)? Again, this strikes us as essential information.
- Mission Safety (IM4*nn*): How does this affect mission objectives or safety (optional)? This would apply only to certain systems, of course.

- Project Schedule (IM5*nn*): How will resolving this incident affect the project schedule (mandatory)?
- Project Cost (IM6*nn*): How will resolving this incident affect the project cost (mandatory)?
- Project Risk (IM7*nn*): What is the project risk associated with fixing this incident (optional)? This seems like another one that should be required.
- Project Quality/Reliability (IM8*nn*): What is the project quality/reliability impact associated with fixing this incident (optional)? Yet another important variable.
- Societal (IM9*nn*): What are the societal issues associated with fixing this incident (optional)? This would apply only to certain systems, e.g., nuclear power plant control software.

The following are the impact data:

- What is the cost impact of this incident? That includes cost to analyze, estimated cost if the fix is done, estimated cost if the fix is not done, and other costs of resolution.
- What is the time impact of this incident? That includes estimated time required if the fix is done, estimated verification time if the fix is done, estimated time if the fix is not done, and actual implementation time if the fix is done.
- What is the risk of this incident? This is a text description.
- What is the schedule impact? This includes assuming the incident is resolved, assuming it's not resolved, and if it is resolved, what the actual schedule impact was.
- What is the contract change, if any?

Again, remember that these data items can be changed later in the lifecycle if required.

7.4 Defect Fields

Learning objectives

(K4) Analyze, classify, and describe functional and non-functional defects in understandable defect reports.

Having looked at the fields defined in IEEE 1044, let's look at how to apply IEEE 1044, particularly how to make its lifecycle and fields map to your situation. As you saw in the previous section, IEEE 1044 specifies a set of mandatory and optional classifications and data fields. People involved with an incident report set and update those fields at various points in the incident report's lifecycle. Typically, that occurs as part of a state transition, when the report moves from one state to another. Remember that figure 7-1 showed how IEEE 1044 maps the lifecycle onto a typical incident report state-transition diagram.

The companion standard to IEEE 1044, IEEE 1044.1, is about how to implement an IEEE 1044–compliant incident management system in your organization. The authors of that standard understand that different organizations have different names for incident classifications and data. So IEEE 1044.1 defines a process for mapping the IEEE terms for fields and data to the names used at a particular organization. Your organization can be compliant with the IEEE 1044 standard without having to rename classifications and data and without having to rework your incident lifecycle.

Why bother with IEEE conformance? Well, it certainly makes sense to use a consistent incident management process within your company across projects. Rex did an assessment for a client once where they tried to compare the quality of the software in a current project with similar software from past projects. Because the incident management processes were not the same—not just the tools, which he could have handled, but the meanings of the classifications and the data gathered—it was not possible to do this comparison. If you have IEEE conformance throughout your organization, then you can compare not only from one project to another, but also with other organizations that are also IEEE compliant.

The following is the process for applying the IEEE 1044 standard to your organization:

Step 1. Map your current classifications to IEEE 1044 classifications. This will give you an idea of the size of the job.

Step 2. Determine the need to conform to IEEE 1044, based on the size of the effort required. Keep in mind that conformance need not be an all-or-nothing proposition. You can decide to achieve conformance in certain areas, for certain classifications and data elements, but not for others.

Step 3. Review the IEEE 1044 classifications, considering especially those classifications that are easy to gather, currently useful, or worth analyzing in the future. You'll want to have those in your system.

Step 4. Select essential classifications for implementation.

Step 5. Define how to use these classifications. For example, what type of analysis do you intend to do on incident data, and when? It's important to know this because your incident management process must be set up to collect the data in such a way as to be useful for these analyses and in such a way as to provide the data in time.

Step 6. Document the categories associated with the classifications. For example, which one is a recognition category, which one is an impact category, and so forth? The less you have diverged from the IEEE 1044 standard in terms of naming, of course, the easier this will be.

Step 7. Document the classifications and their use.

Step 8. If you care about IEEE compliance, this is the point to document conformance or nonconformance with IEEE 1044. What value is this? Well, probably none if your incident management system is for your own use. However, if you are a software or testing services company and you intend to connect your incident management system with your clients', then IEEE compliance might provide a common point of reference.

Step 9. Define the supporting data to collect at each step.

Step 10. Document the supporting data to be collected. (Now, personally, while the IEEE 1044.1 standard puts steps 9 and 10 down here, we'd actually do these in parallel with steps 1 and 5.)

Step 11. Map the classifications and data to the states in your current incident, bug, issue, or defect tracking system. Yes, the correct name, from IEEE's point of view, is incident because you don't know if a behavior is a bug or defect until after investigation. Really, though, call it whatever you'd prefer.

Step 12. Determine and document the process for gathering classifications and data in the incident tracking system. In other words, during what state is a classification or piece of data initially input? During what states may it be updated? During what states must it be updated? Who may or must input? Who may update? Who must update?

Step 13. Plan for use of the information. Again, to us this is backwards in the order of things. We'd do this in parallel with step 2.

Step 14. Provide training to users and management. Users need to know how to input and update classifications and data. Managers need to know how to use the metrics and other information from it.

So, these last couple of sections have given you some ideas on how to expand your incident management system based on IEEE 1044 compliance. However, IEEE 1044 compliance is just a means to an end. What are we trying to accomplish?

Remember that we capture all of this information in the interest of doing something with it. What we want to do immediately is to take action to resolve the incident. Incident reports should capture actionable information. An actionable incident report has the following properties:

- It is complete. It is not missing any important details.
- It is concise. It does not drone on and on about unimportant matters.
- It is accurate. It does not misdirect or misinform the reader.
- It is objective. It is based on facts, as much as possible, and it is not an attack on the developers.

In addition to taking action for the individual incident report, remember that many test metrics are derived from aggregate analysis of incident reports. This was discussed in the Foundation syllabus in the chapter about managing testing. So it's important for incident reports to capture accurate classification, risk analysis, and process improvement information.

Figure 7-2 shows an example of using classification information to learn something interesting about a project. This Pareto chart analyzes the number and percentage of bugs associated with each major subsystem—system, really—in a large complex project. This project, called the NOP[1] project, tied together 10 systems via a wide area network, a local area network, and the phone system to implement a large distributed entertainment application.

As you can see in figure 7-2, the interactive voice response (or IVR) application is responsible for about half of the bugs. The customer service application (or CSA) adds about 30 percent more. The rest of the applications are relatively solid. The content management (or CM) application is less than 10 percent of the bugs. The interactive voice response server's telephony and OS/hardware layers each are around 5 percent, with the remaining applications and infrastructure accounting for the other 4 percent.

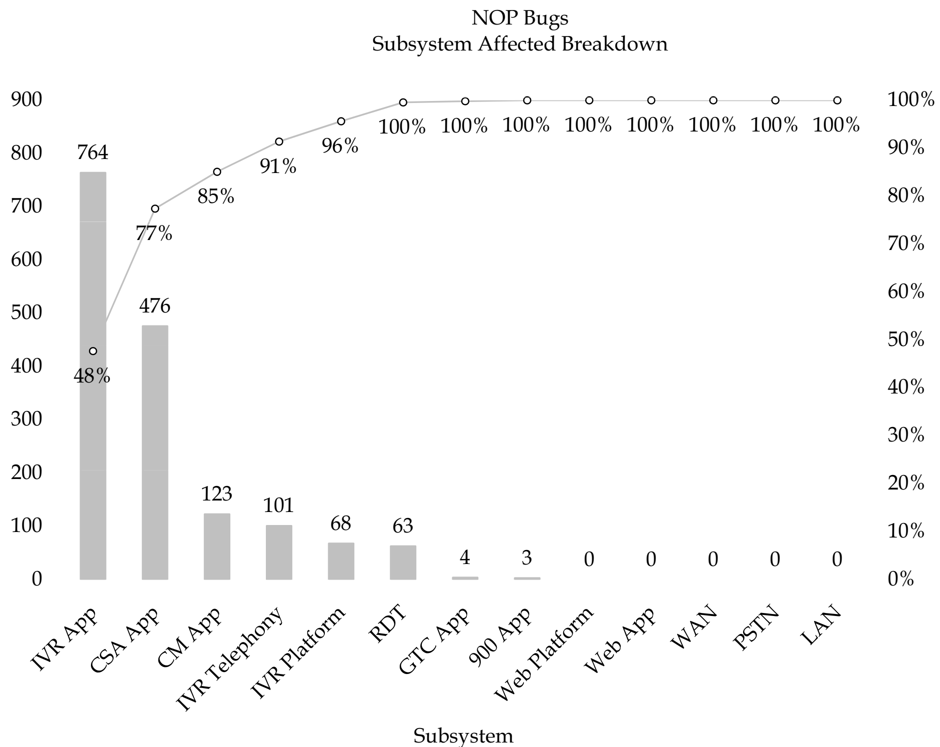


Figure 7–2 *Classification*

1. New operating paradigm

7.5 Metrics and Incident Management

Learning objectives

Recall of content only

The use of the textual descriptive information in the incident reports is usually obvious to technical test analysts, but it's easy to get confused about the use of the classifications. Rex had one client tell him about spending hundreds of thousands of dollars on consulting to improve their incident tracking system to use the latest in classification schemes, orthogonal defect classification. However, they tried to save money on the project by not training people in how to use the fields, so all of the classification information was worthless!

Incident classification information needs to be seen from the immediate point of view of the project and from the long-term point of view of organizational and process improvement. From the project point of view, incident classifications should support test progress monitoring as discussed in the Foundation syllabus. Various metrics like bug cluster analysis, defect density analysis, and convergence (also called open/closed charts) are used during a project to manage defect trends and check readiness for release.

From the organization and process point of view, we want to assess how we're doing and figure out how to do better. So, incident classifications should support process improvement initiatives. We should be able to assess phase containment, which is the percentage of defects that are detected and removed in the same phase they were introduced. We should be able to assess root causes so we can reduce the total number of defects. And, we should be able to assess defect trends across projects to see where best—and worst—practices exist.

7.6 Communicating Incidents

Learning objectives

Recall of content only

Bad incident reports are a major cause of friction and poor relationships in project teams. We see it all the time. To maintain good relations in the team, keep the following in mind.

It is usually not the tester's job to apportion blame or affix fault. Avoid any statements that could be construed as accusations or blaming. Avoid comments that someone could take personally.

A good incident report should provide objective information. Stick to the facts. If you are going to make an assumption or state a theory, state your reasons for doing so. If you do decide to make such assumptions or theories, be sure to remember the first rule about not getting personal.

An incident report is usually an assertion that something is wrong. So, when you are saying that a problem exists, it helps to be right. Strive for utmost accuracy.

Finally—and this is more of a mindset but it's really an important one—start to see incident reports as a service you provide not just to managers but also to developers. Ask developers what information you can include in your reports to help them out. You'd be surprised what a difference this can make.

Some testers get frustrated when "their" bugs don't get fixed. When we see that during an assessment, our first thought is that something is broken in the incident management process. Ideally, a bug triage or incident triage meeting occurs, involving a cross-functional group of stakeholders, to prioritize incidents. It's seldom good to rely on just developer or tester opinions about what should be fixed or deferred. That's not to say that developers' and testers' opinions and input don't count, but rather that good incident management requires careful consideration of the options for handling an incident. Few projects have the luxury of fixing every single incident that comes along.

To sum it all up, good communication and relations within the team, good defect tracking tools, and good defect triage are all important for a good inci-

ISTQB Glossary

configuration control board (or change control board or bug triage committee or incident triage committee)**:** A group of people responsible for evaluating and approving or disapproving proposed changes to configuration items and for ensuring implementation of approved changes.

configuration item: An aggregation of hardware and software, or both, that is designated for configuration management and treated as a single entity in the configuration management process.

configuration management: A discipline applying technical and administrative direction and surveillance to identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements.

dent management process. Incident management is a testing fundamental that all test analysts should master.

7.7 Incident Management Exercise

Assume that a select group of Telephone Bankers will participate in HELLOCARMS testing as a beta test. The bankers will enter live applications from customers, but they will also capture the information and enter it into the current system afterward to ensure that no HELLOCARMS defects affect the customers.

The bankers are not trained testers and are unwilling to spend time to learn testing fundamentals. So, to avoid having the bankers enter poorly written incident reports and introduce noise into the incident report metrics, management has decided that, when a banker finds a problem, he or she will send an e-mail to a test analyst to enter the report.

You receive the following e-mail from a banker describing a problem:

> I was entering a home equity loan application for a customer with good credit. She owns a high-value house, though the loan amount is not very large.
>
> At the proper screen, HELLOCARMS popped up the “escalate to Senior Telephone Banker” message. However, I clicked continue and it allowed me to proceed, even though no Senior Telephone Bank Authorization Code had been entered.
>
> From that point forward in this customer’s application, everything behaved normally.
>
> I had another customer with a similar application—high-value house, medium-sized loan amount—call in later that day. Again, it would let me proceed without entering the authorization code.

The exercise consists of two parts:

1. What IEEE 1044 recognition and recognition impact classification fields and data are available from this e-mail?
2. What steps would you take to clarify this report?

The solutions are shown on the following pages.

7.8 Incident Management Exercise Debrief

Rex did the solution to this exercise, so he’ll describe it here.

First, I evaluated each of the pertinent recognition and recognition impact classifications and data fields to see if this e-mail or other information I assume I have is presented. My analysis is shown in table 7-2.

Table 7–2 *Incident report IEEE 1044 coverage*

IEEE Information	*Available?*
Project Activity	Presumably, we know this for all such beta tests.
Project Phase	Presumably, we know this for all such beta tests.
Suspected Cause	Not available.
Repeatability	Available, but more isolation and replication of this issue is needed.
Symptom	Available.
Product Status	Not available, but we can presume that it's unacceptable for the product to allow the Telephone Bankers to bypass a risk management policy like this.
Environment	Presumably, we know this for all such beta tests.
Originator	Presumably given in the sender information for the e-mail.
Time	Not available.
Vendor	Some of the vendor information we can presume to know, while the other information, such as about when they will supply a fix, is not applicable at this point.
Severity	Available.
Priority	Not available, but again we can presume this is a high priority.
Customer Value	Not available, but inferable.
Mission Safety	Not applicable.
Project Schedule	Not applicable at this point because investigation is required.
Project Cost	Not applicable at this point because investigation is required.
Project Risk	Not applicable at this point because investigation is required.
Project Quality/ Reliability	Not applicable at this point because investigation is required.
Societal	Not applicable at this point because investigation is required.
Cost	Not applicable at this point because investigation is required.
Time	Not applicable at this point because investigation is required.
Risk	Not available, but again we can make some inferences and describe the risk associated with letting Telephone Bankers bypass bank risk management policies.
Schedule	Not applicable at this point because investigation is required.
Contract Change	Not applicable at this point because investigation is required.

Next, I have annotated the report with some steps I'd take to clarify it before putting it into the system. The original information is shown in italic, while my text is shown in regular font.

I was entering a home equity loan application for a customer with good credit.

I would want to find out her exact data, including income, debts, assets, credit score, etc.

She owns a high-value house, though the loan amount is not very large.

I would want to find out the exact value of the house and the loan amount.

I would test various combinations of values and loan amounts to see if I could find a pattern.

At the proper screen, HELLOCARMS popped up the "escalate to Senior Telephone Banker" message. However, I clicked continue and it allowed me to proceed, even though no Senior Telephone Bank Authorization Code had been entered.

I would want to find out if the banker entered anything at all into that field.

I would test leaving it empty, input blanks, input valid characters that were not valid authorization codes, and conduct some other checks to see whether it is ignoring the field completely.

From that point forward in this customer's application, everything behaved normally.

I would test to see whether such applications are transferred to LoDoPS or are silently discarded. If they are transferred to LoDoPS, does LoDoPS proceed or does it catch the fact that this step was missed?

I had another customer with a similar application—high-value house, medium-sized loan amount—call in later that day. Again, it would let me proceed without entering the authorization code.

Here also I would want to find out the exact details on this applicant, the property value, and the loan amount.

7.9 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 last two questions are examples of scenario questions.

1. Which of the following is a section included in an IEEE 829–compliant incident report?
    A. Test items
    B. Procedure steps
    C. Location
    D. Steps to reproduce
2. Which of the following shows the steps of IEEE 1044–compliant incident management in proper order?
    A. Recognition, investigation, action, disposition
    B. Recognition, action, investigation, disposition
    C. Investigation, recognition, action, disposition
    D. Recognition, investigation, removal, disposition
3. Which of the following is a classification that you would make for the first time during the investigation step of an IEEE 1044–compliant incident management process?
    A. Suspected cause
    B. Source
    C. Resolution
    D. Disposition

Scenario

Assume you are a test analyst working on a banking project to upgrade an existing automated teller machine system to allow customers to obtain cash advances from supported credit cards. The requirements specification contains the following paragraph:

> The system shall allow cash advances from 20 dollars to 500 dollars, inclusively, for all supported credit cards. The correct list of supported credit cards is American Express, Visa, Japan Credit Bank, Eurocard, and MasterCard.

You are reviewing an incident report written by one of your peers. The *steps to reproduce* section of the report contains the following statement:

1. Inserted an American Express card into the ATM.
2. Properly authenticated a test account with $1,000 available cash advance balance.
3. Attempted to withdraw $20 from the account.
4. Error message "Amount requested exceeds available funds" appeared.
5. Reproduced this failure with two other accounts that also had sufficient available credit to cover a $20 withdrawal.
6. Verified that the ATM itself had sufficient cash to service the request.
7. Problem did not occur with Visa, Japan Credit Bank, Eurocard, or MasterCard.

4. Assume the defect report is currently in a *new* state, indicating it needs a review. Relying on the information given in this scenario and assuming that your organization follows an IEEE 1044–compliant lifecycle, which of the following statements best describes what should happen next to this report?
   A. Move it to an *invalid* state because it does not describe a valid defect.
   B. Move it to a *defer* state because it does not describe an important defect.
   C. Move it to an *open* state for prioritization by project stakeholders.
   D. Move it to a *build* state so that the tester will check the fix.

5. Assume the defect report is currently in a *new* state, indicating it needs a review. Also assume that your organization follows an IEEE 1044–compliant incident-classification scheme. Rely on the information given in this scenario. Which of the following IEEE 1044 classification fields cannot yet be classified?
   A. Suspected Cause
   B. Actual Cause
   C. Repeatability
   D. Symptom

8 Standards and Test Process Improvement

"Create constancy of purpose toward improvement of product and service, with the aim to become competitive and to stay in business, and to provide jobs."

W. Edwards Deming, in point 1 of his famous 14 points for management.

The eighth chapter of the Advanced syllabus is concerned with standards and test process improvement. The concepts in this chapter apply primarily for test managers. There are no learning objectives at any level defined for test managers in this chapter. However, as a test analyst or technical test analyst working on a test team that might be subject to standards or test process improvement efforts, it's good to be familiar with the main concepts and terms of their work. In addition, if you're studying for the ISTQB Advanced Level Technical Test Analyst exam, remember that certain concepts related to standards are covered in the Foundation syllabus and thus are examinable. So you should read chapter 8 of the Advanced syllabus for familiarity and recall only. You should also review the standards discussed in the Foundation syllabus, ensuring that you have mastered the learning objectives related to them.

ISTQB Glossary

Capability Maturity Model (CMM): A five-level staged framework that describes the key elements of an effective software process. The Capability Maturity Model covers best practices for planning, engineering, and managing software development and maintenance. See also *Capability Maturity Model Integration (CMMI)*.

Capability Maturity Model Integration (CMMI): A framework that describes the key elements of an effective product development and maintenance process. The Capability Maturity Model Integration covers best practices for planning, engineering, and managing product development and maintenance. CMMI is the designated successor of the CMM. See also *Capability Maturity Model (CMM)*.

Critical Testing Processes (CTP): A content-based model for test process improvement built around 12 critical processes. These include highly visible processes, by which peers and management judge competence, and mission-critical processes, in which performance affects the company's profits and reputation.

Test Maturity Model (TMM): A five-level staged framework for test process improvement, related to the Capability Maturity Model (CMM), that describes the key elements of an effective test process.

Test Maturity Model Integration (TMMi): A five-level staged framework for test process improvement, related to the Capability Maturity Model Integration (CMMI), that describes the key elements of an effective test process.

Test Process Improvement (TPI): A continuous framework for test process improvement that describes the key elements of an effective test process, especially targeted at system testing and acceptance testing.

9 Test Techniques

"A worker may be the hammer's master, but the hammer still prevails. A tool knows exactly how it is meant to be handled, while the user of the tool can only have an approximate idea."

Milan Kundera

The ninth chapter of the Advanced syllabus is concerned with test tools, automation, and performance testing. While the Foundation syllabus covers this topic as well, the Advanced syllabus goes beyond the Foundation material to provide a solid conceptual background for using and administering tools. In addition, the Advanced syllabus elaborates on the categorization of tools introduced in the Foundation syllabus. Chapter 9 of the Advanced syllabus has three sections.

1. Introduction
2. Test Tool Concepts
3. Test Tool Categories

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

9.1 Introduction

Learning objectives

Recall of content only

In this chapter, we'll expand on some basic tool ideas described in the Foundation syllabus. We first address general tool concepts and then specific tools. Then we will spend some time discussing both automation and performance testing.

All testers need a basic grasp of the test tools available and what they can—and can't—do. Too often, organizations and individuals bring unrealistic expectations to the use of test tools, commonly the expectation that the choice of the right (often expensive) tool will solve most of their testing problems. Many of the tools-related failures that we have seen can be laid on the altar of unrealistic and uninformed expectations.

The Advanced syllabus groups the tools according to role; i.e., those for test managers, those for test analysts, and those for technical test analysts. Of course, some tools have broader use, across multiple roles.

9.2 Test Tool Concepts

Learning objectives

(K2) Compare the elements and aspects within each of the test tool concepts: benefits and risks, test tool strategies, tool integration, automation languages, test oracles, tool deployment, open-source tools, tool development, and tool classification.

Test tools can be very useful, and, indeed, some are essential. For example, it's hard to imagine a test project that involves more than two or three people getting along without some kind of incident tracking system. Generally, test tools can improve efficiency and accuracy of testing. However, you must select and implement tools carefully to receive the benefits.

While we often think of test automation in terms 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 test tools involves not only careful selection and implementation, but also careful ongoing management. 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.

When automating test execution, you should plan to create a proper framework[1] for your automation system. 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.

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 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 it often means that you'll want to have some documentation in place about how it works, why it is like it is, and so forth. 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.

Finally, remember to plan for expansion and maintenance. Failure to think ahead, particularly in terms of how the tests can be maintained, will reduce the scalability of the automated system; that will reduce the possibility of getting a positive return on your investment.

9.2.1 The Business Case for Automation

We will examine many of these issues in more depth, starting with the business case for automated testing. Remember, test automation should occur only when

---

1. We will use two related terms in this book, *architecture* and *framework*. These two terms are often used interchangeably, but we will not. An architecture is a conceptual way of building an automated system. Architectures we will discuss include record/replay, simple framework, data-driven, and keyword-driven architectures. A framework is a specific set of techniques, modules, tools, etc. that are molded together to create a solution using a particular architecture. We will discuss these differences in detail later in the chapter.

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.

When we talk about automation benefits, notice that we are referring to benefits involving duration, effort, or coverage we wouldn't have with manual testing. The return on investment has to be considered in terms of comparison to other alternatives. In addition, those alternatives must be alternatives that actually would be pursued. In other words, if we automate a large number of tests, but those are tests that 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 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. Let's look first at some examples of initial costs:

- Evaluating and selecting the right tool. Many companies try to shortcut this and pay the price later, so don't succumb to the temptation.
- Purchasing the tool, or adapting an open-source tool, or developing your own tool.
- Learning the tool and how to use it properly. This includes all costs of intra-organizational knowledge transfer and knowledge building, including designing and documenting the test automation architecture.
- Integrating the tool with your existing test process, other test tools, and your team. Your test processes will have to change. If they don't change, then what benefit are you getting from the tool?

Here are some examples of recurring costs:

- 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 the other way, to maximize test durability.
- Ongoing license fees.
- Support fees for the tool.
- Ongoing training costs; e.g., for new staff that come on-board or tool upgrades.

- Porting the tests to new platforms.
- 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 means 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 they had 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 case will often miss important recurring costs, especially maintenance.

We can also think of costs in terms of fixed costs and variable costs. Fixed costs are those that we incur no matter how many test cases we want to automate. Tool purchase, training, and licenses are primarily fixed costs. Variable costs are those that vary depending on the number of tests we have. Test script development, test data development, and the like are primarily variable costs.

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 get a positive return on our investment. The scalability of the system will determine how much investment of time and resources are needed to add and maintain more tests.

When determining the business case, we must also consider risks. In the Foundation syllabus, we discussed these 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 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, open-source tools becoming orphans, and the inability to adapt to new platforms.

In this advanced 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. If 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. We will discuss this to some depth in an upcoming section.
- You see an overall drop in defect detection effectiveness because everyone is fixated with running the scripted, invariable, no-human-in-the-loop automated tests. Automated tests are great at building confidence, managing regression risks, and repeating tests the same way, every time. However, the natural exploration that occurs when people run test cases

manually doesn't happen with scripts. 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 and with little effort. Because the cost and duration are low, we can run them at will, pushing up overall coverage and thus confidence upon release.

Given the size of the initial investment, you have to remember that the benefits will take many months if not years to equal the initial costs. Understand, in most cases, there is no shortcut. If you try to reduce the initial costs of 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?

For one thing, we have better predictability of test execution time. If we can start the automated test set, leave for the night, come back in the morning, and find that the tests have all run, that's a very nice feeling, and management loves that kind of thing.

For another thing, notice that the ability to quickly run regression and confirmation tests creates a byproduct benefit. Since we can manage the risk associated with changes to the product better and faster, we can allow changes later in a project than we otherwise 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 change inherent in certain lifecycle models, especially in agile and iterative lifecycles, the ability to manage regression risk without ever-increasing effort can be a bonus when using automation.

Finally, there are certain test types that cannot be covered manually in any meaningful way where automation is a must. These include performance and reliability testing. With the right automation in place, we can reduce risk by testing these.

9.2.2 General Test Automation Strategies

In a later section, we will discuss different automation architectures. In this section, we want to talk about some general strategies for succeeding with test execution automation. 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 administrative assistant. 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.

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.

The most common question he is asked when teaching a class or speaking to a group of people is, Which tool do you recommend? We think most people are actually questioning, 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 right 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. 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, pragmatic automation using any tool can work for a short 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, the more time will be needed to solve those problems. There are so many problems inherent with automation, a fully drawn-out, strategic plan forward 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 thought-out strategy. 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 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.

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; i.e., 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 that 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.

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. 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. For a good example, 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.[2]

That said, be careful with your test tool portfolio, both freeware and commercial. It's easy to have that get out of control, especially if everyone downloads their own favorite freeware test tool. Have a careful process for evaluating and selecting test tools, and don't deviate from that process unless there is a good business reason to do so.

Finally, to enable the reuse and consistency of your automated testing, make sure to provide guidelines for how to get tools, how to select tests to automate, how to write maintainable scripts, and other similar tasks. This should entail a well-thought-out, well-engineered, and well-understood process.

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, an incident 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. In such

2. See "Quality Goes Bananas" on the RBCS articles page, www.rbcs-us.com.

situations, try to integrate various test tools and get them to exchange information.

Just because you bought a single vendor's test tool suite don't necessarily mean that it will integrate with your other tools. You should consider not buying ones that don't.

If you can't get a fully integrated set of tools, you might have to integrate them yourselves. The extent of effort you put into doing this should be balanced against the costs and risks associated with moving the information around manually.

Lately, there have been many advances in integrated development environments. We testers can hope that this presages similar integration for test tools in the future.

Most test automation tools—at least those for execution—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 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.

Scripting languages and their capabilities vary widely. Some scripting languages are like general-purpose programming languages. Others are domain specific, like TTCN-3. Some are not domain specific but have features that have made them popular in certain domains, like TCL in the telephony and embedded systems worlds. Many modern tools support widely understood programming languages (for example, Java, Ruby, and VBA) rather than the custom, special-purpose languages of the early tools (for example, TSL, SQA Basic, and 4Test).

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 running them establishes nothing more than basic platform compatibility.

In addition to quality considerations, with open-source tools that have certain types of licenses, such as the Creative Commons and GNU General 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, or if you test on a nonstandard platform, you can always build your own tool. Plenty of people 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. Also, since one or two people often develop 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. This could preclude the use of custom and open-source tools for testing such systems, unless you are ready to perform the certification yourself.

Finally a few words about 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 knew. Historically, automation tools have a very high "shelf-ware index." Many is the time that we have found multiple automation tool sets with valid licenses sitting in the back of the lab closet. Closely related to that is try to understand the possible extensibility of

the tool. In other words, the tool currently does not have the sought-after capability, but with a little programming, configuration, and/or extension, it could.

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.

Certain tools create software (automation and performance tools come immediately to mind). When a tool creates software, the output needs to be managed the way other software is managed. 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.

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 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 (so-called) tests that had no way of matching expected with actual results. The assumption was made that if the test did not blow up, it must have passed.

Earning value with automation is rarely easy and never accidental. Before deploying a tool, it is essential to understand that.

9.2.3 An Integrated Test System Example

Figure 9-1 is an example of an integrated test system using automation built for an insurance company. The system under test—or, more properly, the system of systems under test—is shown in the middle.

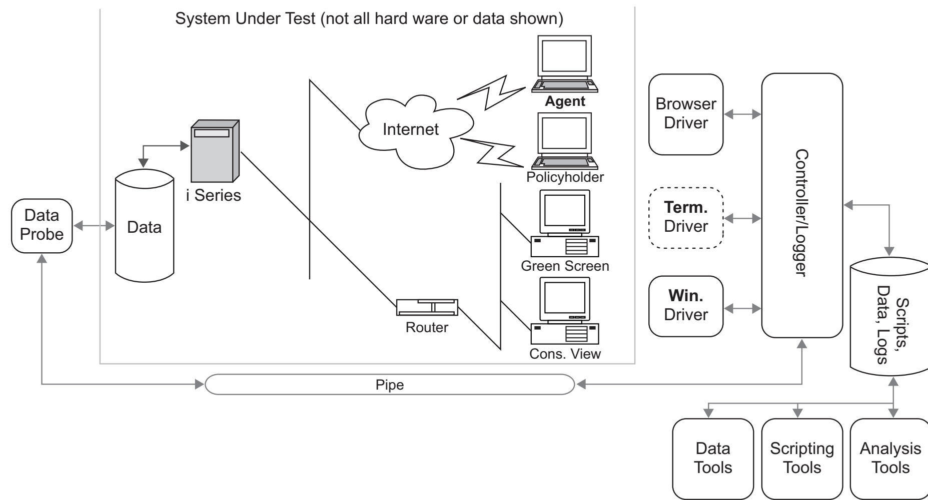


Figure 9–1 *Integrated test system example*

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 payable and receivable, and the like.

On the right side of the figure, you see the main elements of the test automation system. For each of the three interface types, we need a driver that will allow us to submit inputs and observe responses. The terminal driver is shown in a dotted line because there was some question initially about whether that would be needed. The controller/logger piece uses the drivers to make tests 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, then find tools that can support it rather than let

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. We need a way to catch those kinds of bugs, and this design does that for us.

In all likelihood, the tool or tools used to implement the right-hand side of this figure would be one or two commercial or freeware tools, integrated together. The data probe and pipe would probably be custom developed.

9.3 Test Tool Categories

Learning objectives

(K2) Summarize the test tool categories by objectives, intended use, strengths, risks, and examples.

(K2) Map the tools of the tool categories to different levels and types of testing.

Throughout this book, we've been looking at taxonomies of various kinds. In this section, we'll go through tools, organized by taxonomies. Let's begin by looking at various ways we could classify tools.

ISTQB Glossary

test management tool: A tool that provides support to the test management and control part of a test process. It often has several capabilities, such as testware management, scheduling of tests, logging of results, progress tracking, incident management, and test reporting.

In the Foundation syllabus, we grouped tools by the test activity they supported. We can also classify tools other ways:

- By the level of testing they support; e.g., unit test tools, integration test tools, system test tools. This strikes us as quite weak, given the potential for reuse.
- By the types of defects we are looking for with them.
- By the type of test techniques they support; e.g., orthogonal arrays and classification tree tools are useful when pairwise testing is needed.
- By the purpose of tool.
- By the application domain they are used to test, which is most useful for domain-specific rather than general-purpose test tools.
- Based on how they are applied.
- Based on the user who is expected to use the tool.

This last method is how we'll classify tools in the following sections. In the discussion about tools and their classifications, keep in mind that we are augmenting the tools categories already introduced in the Foundation syllabus, along with introducing new tools categories. You'll need to refer to the Foundation syllabus as well as the Advanced syllabus for general information concerning the other tools categories not included in this section.

9.3.1 Test Management Tools

Test management tools are used to help manage the artifacts and processes of testing. Since testing generates information, there's a lot of information to manage. Good test management tools facilitate the storage and dissemination of information.

Some test management tools have built-in requirements modules, others allow programmatic/API connection to full requirements management tools. Since most test management tools also include the ability to act as a central

repository to store analysis, design, and implementation artifacts and link them all together, these tools can facilitate full traceability between the test basis and the artifacts we create to test it.

Some tools allow the test team to organize conditions for testing in different environments. Together with the storage of test environment data for different environments and the instructions for building and initializing different environments, these tools can help us manage very complex testing environments.

Most test management tools allow the tracking of concurrent test execution, including when tests are running in different test environments at multiple sites. Some tools automatically collect results from automated tests and supply a simplified interface for manual testers to record their results.

Some of these tools contain modules for tracking incident/defect records. Some have built-in defect workflow management; others simply interface to stand-alone defect tracking tools.

By storing all of the artifacts in one place, these tools facilitate the automatic generation of various test-related metrics, including these:

- Number and current status of requirements
- Time metrics, including the time needed for preparing and executing test cases, test suites, regression test sets, and other test-process-describing metrics
- The number of test cases, test scripts, test environments, and so forth
- The current state for all test cases, including passed, failed, skipped, blocked (and the blocking conditions), queued, and in process
- Trends in various metrics like bug find/fix rates
- Logging and failure information

Test management tools are used by test managers, test analysts, and technical test analysts. These tools are useful throughout the project lifecycle.

9.3.2 Test Execution Tools

Used properly, test execution tools should reduce costs, increase coverage, and/or make tests more repeatable. Because of the large amount of effort and tedium, test execution tools are often used to automate regression tests.

Most test execution tools are also called capture/replay (or sometimes record/replay) tools; they work by executing a set of instructions written in a

ISTQB Glossary

test execution tool: A type of test tool that is able to execute other software using an automated test script, e.g., capture/replay.

scripting language, which is just a programming language, customized for the tool. The tool usually gives precise ability to drive key presses and mouse actions, along with inspection of the graphical user interface or some other interface.

The scripts can be recorded using capture (record) facilities; in many of these tools, you can also program the scripts as you can a real application. Capture/replay tools can be useful for tracing the path of exploratory or other non-scripted testing, but the resultant scripts and expected results are very difficult to maintain. These tools lie at the heart of an effective automation architecture; we'll discuss that later.

Recording is usually performed by intercepting and reading the messages sent to the operating system queue. Every time you press a key or mouse button or move the mouse, messages are generated by the operating system and sent to a central queue. From there, they are dispatched to the GUI elements that are to react to them.

For example, if we move the mouse cursor to a button on the screen and press the left mouse button, the following chain of events may occur:[3]

- A series of mouse movement messages are generated. These start at the screen coordinates where the mouse cursor was and continue to where you stopped moving the mouse. These messages are used by the operating system to successively redraw the cursor across the screen giving the user feedback on current location.
- When the left mouse button is pressed, several messages are generated, showing that the button was depressed at a given location and then released at a given location.
- The application currently under the cursor is dispatched those messages, giving the application focus and making it the current application.

3. Different operating systems and applications may behave differently. This chain of events is describing MS Windows and a standard Win32 application.

- The application window under the cursor is forwarded the messages and is given the focus. That causes the window to redraw if it was not in focus before. If it was partially obscured by a different window, it is moved to the top and becomes fully visible.
- The control group (if any) under the mouse cursor is forwarded the messages, telling it of the mouse click. Sometimes control groups are fairly complex; the messages keep getting passed inward, container to container, until they actually reach the button that the user clicked.
- At this point, the button accepts the mouse click (assuming it is enabled). It calls the code functions that are supposed to execute when the button is depressed and released.

Test record tools watch over the queue and capture the messages that are generated. Early versions of these tools simply captured the raw information; e.g., a mouse click occurred at location X =137, Y = 567 on the screen. These would generate the following line of code in the script:

```
MouseClick Left 137,567
```

Later on when playing back this script, this line would simply re-create the set of messages to move the mouse cursor to that location and generate a left button click there. If the screen was exactly in the same state and the application being run was exactly in the same state, this would usually work correctly. However, any changes to the screen and the applications running on it would likely cause this action (and hence the test being run) to fail. For example, if the button was no longer at that same spot (137, 567) on the screen, whatever was currently there would receive the click.

Subsequent generations of test execution tools were refined to capture some context around the event. Many of the current tools are designed to understand what the mouse click means. In the tool scripting language, they might generate the following set of lines to record the event:

```
SetActive Application XYZ
SetActive MainWindow
PressButton EnterKey
```

These lines would be able to re-create the button press no matter where the window was, no matter what the screen looked like. While these changes

have made the tools more robust, most tools are still a long way from being foolproof.

One continuing problem with these tools is the identification of GUI objects on the screen. As human beings, we see a collection of widgets on the screen, organized to do a particular task; each is a metaphor for the particular task we might want to do. We see a button, a list box, a window. There are only a handful of these widgets that we recognize by their shape and context when we use the operating system. The current window the author is looking at contains a rich edit box, several toolbars, a menu bar, a status bar, a ruler, vertical and horizontal scroll bars. These are drawn specifically so users can recognize them.

When people learn how to use, say, MS Windows, they learn to mouse click radio buttons, check boxes, icons and push buttons; type into and read from text fields; drag scroll bars; etc. Once we learn to use these screen metaphors, we can run pretty much any application.

Unfortunately, the capture/replay tool does not see the same screen metaphors that human users do. If you programmatically tell the tool to click a button, it cannot see it; it has to ask the operating system to put a mouse click on this specific object defined by the following properties.

The properties that are used to define an object may be fixed by the tool or they may be defined by the user. Some of the properties that may be used to uniquely identify an object are subject to change. For example, length, width, X-location (in window), Y-location, Z-location,[4] tab order, associated text, etc. may all change from build to build and make very poor object identifiers when using a capture/replay-type tool.

A common problem when using these tools is the occurrence of test failures caused by change. If we set up the tool to use certain properties to uniquely identify objects with which we interact and those properties change, our tests using those objects will fail. Good automators will have a variety of techniques to deal with this (and other issues) when they automate tests. Specifics on dealing with object identification, because each tool is different, are out of scope for this book.

4. The Z-order is used to determine which windows are on top of other windows.

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.

fault seeding: The process of intentionally adding known defects to those already in the component or system for the purpose of monitoring the rate of detection and removal and estimating the number of remaining defects.

fault seeding tool: A tool for seeding (i.e., intentionally inserting) faults in a component or system.

Test execution tools can use a comparator to compare the expected results—which may have been captured during some previous test run—with the actual results. Usually these comparators can be instructed to avoid comparing fields that will vary, like dates and times.

Poor skills in programming and bad design of automation architecture can cause failure of test automation. There's also a need for careful management. And, since the test execution scripts are programs, we need to remember to test them.

These tools can be used at any level of testing. At the unit test level they are used by developers, perhaps doing test-first or test-driven development. At other levels they are usually used by technical test analysts to create tests and may be used in the run configuration by any tester. Jamie's credo has always been that automated tests need to be so easy to run that anyone on the project should be able to kick them off and read the results.

9.3.3 Debugging, Troubleshooting, Fault Seeding, and Injection Tools

Debugging and troubleshooting tools can help us narrow down the area where a bug lives. In some cases, as with user interface bugs, the location of the bug is obvious, but in other cases the bug can be a long way from the symptom. Debugging tools can include logs, traces, and simulated environments.

Debuggers have the ability to allow a programmer or technical test analyst to execute programs line by line, watching for unexpected control or data flows. They can halt the program at any program statement if the operator has a hunch

ISTQB Glossary

dynamic analysis tool: A tool that provides runtime information on the state of the software code. These tools are most commonly used to identify unassigned pointers; check pointer arithmetic; monitor the allocation, use and deallocation of memory; and flag memory leaks.

static analysis: Analysis of software 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.

static analyzer: A tool that carries out static analysis.

about where the bug lives or wants to check some variables at the point. Debuggers can set flags on and examine program variables.

As was mentioned in the Foundation syllabus, debugging is related to testing but is not testing. Similarly, debugging tools are related to testing but are, strictly speaking, not testing tools.

Debugging and troubleshooting tools are used mostly by programmers and technical test analysts. Technical test analysts can use these tools at any point in the lifecycle once code exists.

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 is also sometimes called mutation testing. As you saw earlier, NASA uses this technique to help in reliability testing.

Fault injection is usually about injecting bad data or events at an interface. For example, Rex has a tool that allows him to randomly corrupt file contents. Notice that this is something that Whittaker's attack technique discusses.

Fault seeding and fault injection are mainly used by programmers and technical test analysts. Technical test analysts can use these tools at any point in the lifecycle once code exists.

9.3.4 Static and Dynamic Analysis Tools

Static analysis tools, which automate some parts of the static testing process, can be useful throughout the lifecycle. They provide warnings about potential

problems with code, requirements, etc. For example, a code analysis tool will flag dangerous or insecure constructs. Running a spelling and grammar checker on a requirements specification can reveal a difficulty level that's too high. We discussed static testing in chapter 4.

The usual problem we've had when using these tools for clients is the number of false positives. In this case, a false positive is a potential problem that does not actually cause any damage. The number of false positives on an existing code base can be huge, as many as one for every 5 or 10 lines of code.

There are various strategies for working around this, like using the tool on only new and changed modules of code. Fortunately, vendors recognize this problem and are working to fix it.

Static analysis tools are mainly used by programmers and technical test analysts. They can use static analysis tools at any point in the lifecycle once the work product to be analyzed exists.

Let's look at an example of static analysis and text execution tools in action.

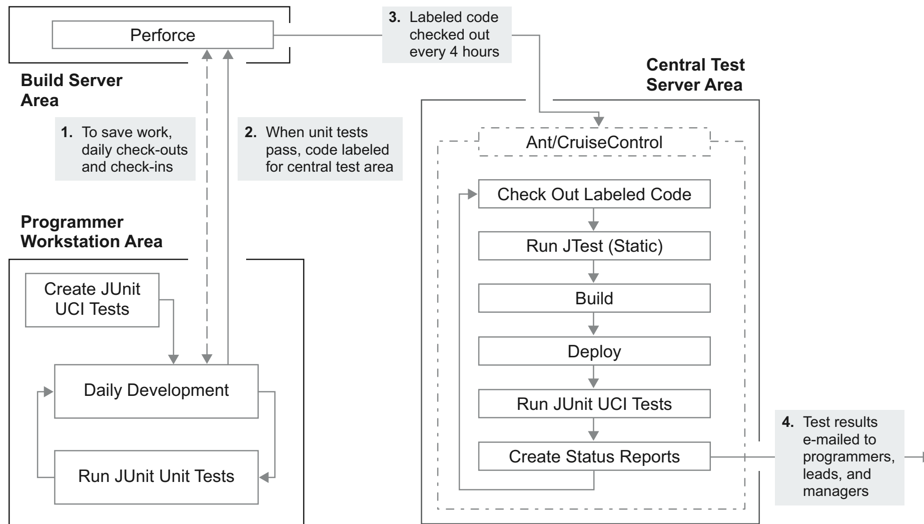


Figure 9–2 *Static analysis and unit testing example*

Figure 9-2 shows a testing framework RBCS built for a client. This tool provided automated static analysis and unit, component, and integration testing, using both commercial and open-source tools.

The best way to read this figure is clockwise starting at the bottom left. Let's see how this worked.

The individual programmers worked on their own areas, creating unit, component, and integration tests for their code as they built it. They ran the unit tests locally using JUnit. At the end of each day, they checked their work into Perforce, the configuration management system, but that code was labeled as "not ready for the build" until it was approved.

Once the unit tests passed, the programmer would have his code and unit tests reviewed by the lead programmer in his group. (Yes, that's a bit more informal of a review process than we would have preferred, but it was all we could convince them to do.) If the review was a success, the code and tests were then checked into Perforce labeled as "ready for the build."

Now, the central test server had a script running on it that checked for new "ready for the build" labeled code every four hours. If it found some, it would initiate a new test run. That test run consisted of two parts: first, a static test using the JTest tool from Parasoft; next, a full dynamic test running all the unit, component, and integration tests in the repository. Once the test run completed, the results were e-mailed and posted on the intranet.

We think this approach is clever and one just about every development organization should try to adopt. Notice that the very activity that increases regression risk—checking in new or changed code—also triggers the actions that will reduce that regression risk.

Dynamic analysis tools provide runtime information on the state of the executing software. They can be used to pinpoint a number of problems that are hard to find in static analysis and hard to isolate in dynamic testing. These tools include evaluating pointer use, but perhaps memory leak detection is the most common example. Memory leaks are particularly likely in programming languages like C and C++ where programmers manage memory directly—since they sometimes mismanage it!

These tools are most often used by technical test analysts, but they can be used by any tester. Since dynamic analysis tools tend to sit in memory quietly until a failure occurs, it is often useful for testers to run it during some of their testing at the system test level. We discussed these tools extensively in chapter 4.

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.

9.3.5 Performance Testing Tools

Performance test tools typically consist of two major elements. One is a load generator. The other is a measurement and analysis component.

The load generator executes a script, which implements an operational profile. You should remember the concept from chapter 5. Sometimes these scripts are captured, though our experience is that they more typically are programmatically created. The script needs to be able to throw at the system under test whatever kind of data the system needs to accept.

When you run a mix of scripts under most performance testing tools, a complex mixture of simulated or virtual users can be pounding on the system simultaneously. In many cases, the tools are not pounding directly on the user interface, but rather on a communication interface such as HTTP or HTTPS.

While this is happening, the measurement component is gathering metrics. including these typical metrics:

- Number of simulated users
- Number and type of transactions generated by the simulated users
- Response times to particular transaction requests made by the users

Based on these, various reports can be created, including graphs of load against response times. Performance testing is a complicated activity with a number of important factors to consider:

- First, do you have sufficient hardware and network bandwidth on the load-generator host required to generate the loads? We have seen load generators saturate before the system under test did, which defeats the purpose.
- Second, is the tool you intend to use compatible with the communications protocol used by the system under test? Can the tool simulate everything you need to simulate?

- Third, does the tool have sufficient flexibility and capability to allow you to create and run the different operational profiles?
- Finally, are the monitoring, analysis, and reporting facilities that you need available?

While simple load generators for reliability testing are commonly built in-house, performance test tools are typically purchased or open-source versions used. The real tricky part—and where most of the work will be should you decide to build your own performance testing tool—is in the measurement and analysis piece. When we and our associates have had to build performance testing tools, that was usually the hardest part.

Let us mention something at this point that you should keep in mind. Many performance-related defects are design problems. We have seen late discovery of serious performance problems doom a project. So, when performance is a key quality risk, be sure to use modeling and unit testing to test critical components rather than waiting for system tests.

Performance test tools are typically used by technical test analysts. They can use these tools during any test level as part of test execution for that level, but it happens most typically during system and acceptance test.

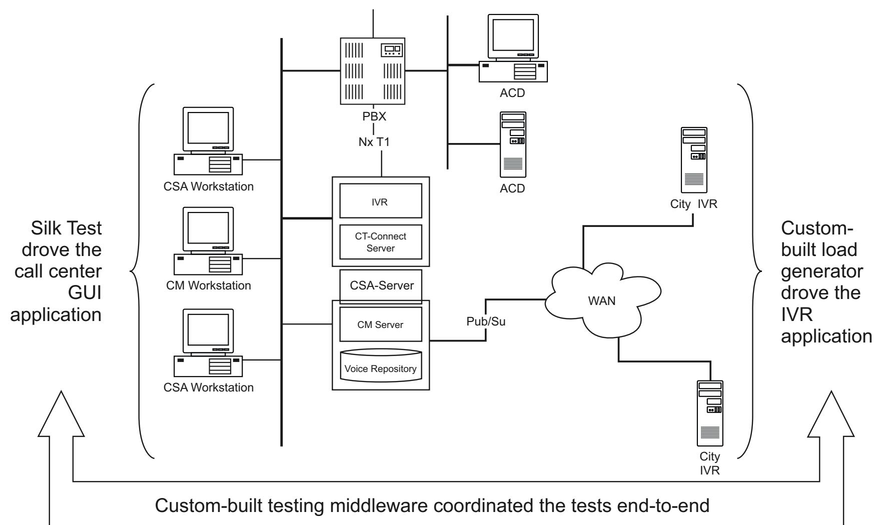


Figure 9–3 *Performance testing with integrated tools*

Let's look at an example here, this time for performance testing using integrated tools. We've mentioned the wide area network IVR server project earlier in this book. In Figure 9-3 you see the architecture of that system. We have the wide area network of IVR servers to the right side, the support content management and customer service application servers in the center, and the customer service agent desktops at the left.

If someone pressed 0 while on the IVR, they were supposed to be transferred via VoIP (voice over IP) through the network to a waiting agent. At the same time the agent answered the phone, that agent was supposed to see, popping up on his screen, all the profile information in the system about the user he was about to talk with. That had to work reliably, and it had to work even if the server was loaded down.

Therefore, we needed a way to do end-to-end performance testing. We had a load generator that could create calls on the IVR side. The load generator was scripted, so we could include 0 in some of those scripts. The load generator was also able to coordinate, through custom middleware, with QA Partner (now called SilkTest), which was driving and watching the screens. We could actually time the transaction, from pressing 0 on the IVR to seeing the screen pop up on the customer service agent's desktop. We would run a bunch of these transactions, capture the transit time, and log that information for later analysis.

9.3.6 Monitoring Tools

Closely related to performance tools are monitoring tools, which are used to observe a particular system or subsystem while it is exercised (often using a performance tool). Monitoring tools are designed to give visibility to the internal workings of a network, database, server, or other subsystem in a complex environment. Some of these are intrusive, changing the way a system works, but many of them are essentially passive without materially affecting the system being observed.

Consider the complex path of a transaction that occurs in an Internet-/browser-based environment. A user or virtual user types in a URL. That request traverses the IP stack and is encoded into packets of information. It is placed on the LAN. It travels through the router and the router switch, to the proxy server, through the firewall and on to the Internet. The packets weave

their way through various routers and switches to the destination. At the destination, the packets go through the firewall and get unpacked by traversing through the IP stack and to the web server. There, the web server may interact with the database server, the application server, and any variety of other players. Then, the requested information weaves its way back through a similar path.

At any point through this path, a system may slow down or stop the progress of the action. That means we may need any number of different places to observe the traffic. That is where monitoring tools come in. A network analyzer (or sniffer) can intercept and log the traffic passing through the network. It can decode what is in the packet, allowing near real-time information when there are problems on the network. This allows the sniffer to gather and collect network statistics.

Other tools can monitor database usage, various server usage, router and switch usage, and firewall activities and gather measurements just about every step of the way. While these tools are often used in production to ensure optimum performance, they are particularly useful when performance testing. When a system is put together, there are often inefficiencies and bottlenecks. Typically, when performance testing, the tester will arrange to have various experts (from the network, server, back-end and other areas) to be available, monitoring the testing. When a slowdown occurs, the experts manning the monitors try to figure out why. Often the problem is the result of one or more settings rather than a complete failure. All of these people work together to tune the system for peak performance as well as look for bugs.

These tools can be used by technical test analysts; they are often used by domain experts in production.

9.3.7 Web Testing Tools

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 will also provide a graph of the link tree, the size and speed of downloads, the number of hits, and other metrics. Some tools will do a form of static analysis on the HTML to check for conformance to standards.

ISTQB Glossary

hyperlink test tool: A tool used to check that no broken hyperlinks are present on a website.

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 capture/replay tool for browser-based testing.

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 lifecycle once the website to be analyzed exists.

ISTQB Glossary

emulator: A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system. See also *simulator.*

simulator: A device, computer program, or system used during testing that behaves or operates like a given system when provided with a set of controlled inputs. See also *emulator.*

9.3.8 Simulators and Emulators

Simulators, as those of you who have watched any movies about space flight know, provide a way to test in an artificial environment. We might want to do this because some of the code or some other part of the system is unavailable. We might want to do this because the real system is too expensive to use in testing. We might want to do this because testing in the real system is unsafe. For example, aircraft, spacecraft, and nuclear control software is usually tested in simulators before being deployed. You could say that the deployment constitutes the first test in the real environment.

Some simulators can be sophisticated, able to inject faults, produce reproducible pseudo-random data streams, and the like. Our experience with testing in simulators has been that no matter how good they were, there were always things we found when we went onto the real hardware. Timing problems and resource constraints and dependencies in particular are tricky to simulate.

An emulator is a type of simulator in which software mimics hardware. While working in the early ’90s, Jamie used an emulator for testing the rewrite of the operating system for a midrange computer; the CPU that was destined to be the heart of the system had not yet been fabricated. It was slow, but it did allow them to unit test the code and shake out many of the bugs before the hardware existed. Interestingly enough, this emulator was called “the piranha simulator.” It is our experience that, in real life, simulators and emulators are often mistaken for each other.

Test analysts and technical test analysts, depending on the type of emulation required, use these tools. They can use these tools during any test level, as part of test execution for that level, but these tools are most typically used during early test levels when the item simulated or emulated is unavailable.

ISTQB Glossary

data-driven testing: A scripting technique that stores test input and expected results in a table or spreadsheet so 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/replay tools. See also *keyword-driven testing*.

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. See also *data-driven testing*.

test oracle: A source to determine expected results to compare with the actual results of the software under test. An oracle may be the existing system (for a benchmark), other software, a user manual, or an individual's specialized knowledge, but it should not be the code.

9.4 Keyword-Driven Test Automation

Learning objectives

(K3) Create keyword/action word tables using the keyword selection algorithm to be used by a test execution tool.

(K3) Record tests with capture/replay tools in order to make regression testing possible with high quality, and many test cases covered in a short timeframe.

Many years ago, Jamie attended a workshop on automation that was attended by many of the most experienced automators in the country. While he was talking with several of the attendees, they 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, we 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 the same type of solutions.

In testing, everyone trying to automate had a common problem. Automation of testing was a meme complex[5] that had spread like wildfire in the early 1990s, and there were a lot of tools being developed for it. Unfortunately, the basic capture/replay process just did not work. The idea 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. A lot of these solutions looked very similar. Looking 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 return on investment (ROI), or the need to show value. Most of the tools (all vendor tools back then) were very expensive. A positive return on investment required that we be able to create large numbers of tests that could be run whenever needed. However, while we could create large numbers of tests, we could almost never run them all successfully.

We needed a solution. The solution, as you will see, was a logical progression of architectures. Most automators went from capture/replay to the framework architecture 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, automation is still evolving. Many are the times 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

---

5. Definition of *meme*: 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.

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 he knows what he is doing. Too often, we have seen people with resumes claiming expertise at automation when the only thing they know is a single capture/replay tool.

In his book *Outliers (Little, Brown and Company, 2008)*, 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; i.e., 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).

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 wrappers 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/replay 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. The script had triggered the failure, so when they came in the next morning, they had a good record of it due to the script's running. 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 you want a stable, long-term testing solution that works every time you 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 9-4 you see a recorded script from one of the all-time popular capture/replay tools, WinRunner from Mercury-Interactive (currently owned by HP). We have removed the spaces to save room, but this is pretty much what is 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_MRnumber", "MRE5418");
3.    obj_type ("edt_MRnumber","<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_MRnumber", "BC3456 ");
11.   obi_type ("edt_MRnumber","<kTab>");
12.   password_edit_set("edt_Password","dzctmzgtdzbs");
13.   obi_type ("edt_Password","<kReturn>");
14.   set_window ("FREDAPP97 Msg", 5);
15.   button_press ("Yes");
```

Figure 9–4 *WinRunner recorded script (partial)*

First of all, it is a little difficult to read. This script exemplifies why we have code guidelines and standards. But since it is meant to execute directly, maybe we aren't suppose 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), she sits down, looks at the screen, and interacts with what she sees 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. If the object we want to deal with is not active, we click it 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 incident.

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.

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 is completely wrong in how it models the human tester. 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 that is not a valid model. Manual testers—at least those who know what they are doing—add 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 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 9-4. 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 and then continue on? Is it an unrelated failure that we must stop for? The automation tool has no such capability.

If the scripter puts in a check for a particular thing and that thing is incorrect, the tool will find it. But anything else will not 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 a 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 it took a long time showing up. Not on the screen? A tester will scroll it. Renamed? A tester will find it.

The truth is that some modern 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 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 captured 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 test two 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? Oops! We'll see you Monday morning.

9.4.1 Capture/Replay Exercise

Referring to the recorded WinRunner script back in figure 9-4:

1. The script has been changed to pseudocode in figure 9-5 so it is more understandable. Analyze what the tool is doing and identify the likely failures that could occur when a script like this is run.
2. What changes might mitigate the issues identified in question 1?

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 "FREDAPP97 Msg/5" the active window
15. Press the Yes button

Figure 9–5 *WinRunner script translated to pseudocode*

9.4.2 Capture/Replay Exercise Debrief

The original script is one that Jamie recorded. This debrief was performed by him. Analyzing the pseudocode, there are a number of global failures possible;

that is, they could happen in any one of the steps. They include the following items:

- Timing. At any given time, a GUI object might not be available within the time limit established by the tool (typically 3 seconds). In line 7, we are waiting for a pop-up window to appear. The previous step pressed a Return key inside an edit. That means the default button would be pressed, usually resulting in a pop-up. Same in lines 9 and 14. In addition, there is an assumption that all of these objects are always available. My recollection is that tabbing out of certain controls caused other controls to become enabled or disabled. That created race conditions when going to the next control that caused many failures.
- The naming conventions of the different windows (FREDAPP followed by an integer) likely means that the GUI mapping was not correctly created. Windows should be named with a meaningful name. We have no idea what was used for identification of these windows, which means that any change is likely to break the script.
- Same for all controls that are dealt with. Any change in the system under test is going to hit this script like a bowling ball.
- Tabbing from control to control is often a bad idea since tab order tends to change with usability modifications.
- The user ID and password are coded directly in. This limits the usefulness of the script over time.
- Likewise, all data is hard-coded. This application was used in a medical setting where the data changed rapidly. The values being entered could literally be changed that afternoon.
- Lines 7 and 8 represent a window that did not always pop up. It depended on other settings, both on the server and in the client's control. Approximately 40 percent of the time it did not pop up, causing the script to fail.
- No error handling; when a problem occurs in this script, the entire suite dies.

I could actually come up with about a dozen more problems with this script, but you get the picture.

For the second part of the exercise, the answer is pretty simple. Programming! All of the issues enumerated, and many of the others not mentioned, can be solved the same way. By using the programming language of the tool (in this case TSL), good software engineering techniques could be used to solve almost any issue.

Notice, though, that programming raises the cost of the script and the cost of the automation and adds many extra tasks that did not exist before. *There ain't no such thing as a free lunch!*

9.4.3 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. (While we were writing this book, Rex had a number of programmers and testers make the exact same complaint about tests created with 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 it 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 (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 has no such problem.

Timing changes. Boom, automation is likely to fail. A human has no such problem.

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 cleverer 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 automation killer!

Let's work through a theoretical situation, one that every automator who has made the jump from capture/replay to the next step has made.

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 test case in turn fails. You analyze the problem and there you find that, for no particular reason, 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 that phrase 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 File Open. You slowly count to 10 in three languages under your breath to avoid saying something unpleasant.

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 ROI.

9.4.4 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.

The automation tool the tester is using has a programming language; programming is the solution to the problem. 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 file name.

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 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 investment costs, much less get positive ROI. If you cannot reliably run lots of tests, automation will never pay off.

Notice now, however, that this is no longer the capture/replay architecture. It is partially recording, partially programming. And there are a lot more failure points than just trying to open the 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 incident record. We can put multiple tasks in a single function, giving us aggregate func-

tionality. 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 her imagination. The more often a function is going to be used, the more value there is to making it elegant.

This leads to what we call the simple framework architecture. Other people use other names, and there does not appear to be any standard name 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, Land Cruiser, etc. Each is seen as an automobile (as compared to, say, a truck or an airplane).

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 a positive return on the investment.

Functionality that is used a lot gets programmed with functions. The more failure prone the functionality, the more time we spend carefully programming in automatic error handling. Some stuff that is rarely done might still be recorded. 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: a programming tester that many people just call an automator. 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 investing in a long-term project, building that 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 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 test 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 between tester and automator as disjoint. Not every tester wants to be a programmer, and that may 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 they are supposed to be mitigating to the system?

We believe the best organizational design is to have a test subgroup made up of automation specialists. This would include, as with 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 positive ROI. 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.

9.4.5 Data-Driven Architecture

The number of scripts 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 investment. 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. You are testing a critical GUI screen with 25 separate fields. There are a lot of different scenarios that you want to test. You also need to test the error handling to make sure you are not entering bad data into the database. Manual testing this is likely to be ugly, mind-numbing, brain-deadening testing of the worst sort. Enter all the values. Press OK. Make sure it is accepted. Go back. Enter all the data. Press OK. Go back. Repeat until you want to find a bridge to jump off. This is the exact reason automation was invented, right? But you could easily have 100 different scripts, one for each different (but similar) test case. That is a lot of maintenance.

To automate this in our 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, we press 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 who has 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 variable (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 almost identically to all of our framework scripts except for the parameterization) represents any number of actual tests. It has 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 that 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 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 is 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. 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 all 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, about finding failures and mitigating risk. We do not want them worrying about how the automation works. Need a completely new scenario? The tester needs to give the automator enough information that he can script it—a good solid manual test procedure is optimal. If it is something that he needs 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 becomes nonlinear where one script fixed may fix dozens or hundreds of tests.

But we are not yet done. We still have scripts.

9.4.6 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 (or sometimes action word) testing. The official definition of this is a meta-language 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 9-1 contains a partial manual test procedure; not a full blown IEEE 829 test procedure specification, but a nice minimalist test procedure.

Table 9–1 *A minimal manual test procedure*

Task	*Data to use*	*Expected Result*
Start system under test	C:\...\SUT.exe	Starts up correctly
Login	User ID/Password	Logs in correctly
Create a new record	Pointer to spreadsheet row of data	Record creation notification
Edit a record	Record key: a pointer to spreadsheet row of data	Expect change notification

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 in the system under test. It's 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 training monkeys, right?

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. Starts up correctly, log in correctly, record created correctly, 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? 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. Certainly column two is not abstract; that is the concrete data of the test case. With good testers, 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, etc. 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 he wanted to do in the order he wanted, and to pick up the data he 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 meta-language (in this case, *meta* means high level) that has, as a grammar, the tasks that a tester wants to execute. It does not have the normal structures (loops, conditionals, etc.) that a normal, procedural programming language has. It is actually a lot like Structured Query Language (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 that 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 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 meta-language to use, the lower we can drive our variables costs, like training and support.

Therefore, perhaps the most important part to make this architecture work is that we need an intelligent front end that can lead a tester through the process of building the test. It should have drop-down lists that are intelligently loaded with keywords so the user does not need to remember the grammar. Such a front end can be built in Excel (on the low end) or just about any rapid application development (RAD) language that the automator is comfortable with (Jamie tends to use Delphi) on the high end. It is our belief that the better we build this, 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 completely blank desktop. 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 front end would have a drop-down list which would include the keyword <StartApp>. If the user clicks <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, 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 which 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.

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.

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 isn't likely to always be the case. Errors might occur or the GUI could be changed. This is where the 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 move 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.

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 must be able to execute a task, as we discussed for our manual test procedure. The testers arrange 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 reasonable-

ness 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 script. This will handle all of the logging tasks, exception handling, failed test handling, and execution of the keywords. Essentially, this virtual machine is the framework. When started, this script likely sets up the environment, initializes the log, picks up the first test to be run, picks the first keyword, executes it, picks up the next keyword, etc.

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 her 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 the tester could immediately start using it in her test cases. Then the automator would need to create the functionality behind the keyword to make it executable before the tests are to be run.

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 automator must be a good developer. 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.

But consider the implications of this system. We have effectively built a truly scalable 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. Ten or twenty or a hundred testers could all use the keyword system. There is no limit to the number of testers, nor is there a limit to the number of tests. One thousand or 100,000 test cases; we still only 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.

Reviewing where we are now:

- Domain experts (i.e., testers) 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, most of these that Jamie has built, he has included the execution module in the front end. By allowing anyone to graphically choose what and when to run, we simplify that aspect also. Some of these also had log browsers built in also.

There is a huge benefit to the keyword architecture that we have not yet addressed. Throughout this entire section on 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 hard-coded into the procedure. As long as we do the same tasks, most 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. Medical, mission-critical, and similar software testing 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 DOS, another way for Word for Windows 3.1. Each different version of Word changed the way we opened a file. 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 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 ideas change 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 not change. The number of keywords tends to be relatively stable, so the maintenance will be relatively minimal (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. To record a script, the system has to be well into system test so a recording can be successfully made. The earlier you record, 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, although 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. Certainly the functionality of the keywords will not be there yet. However, the automators could conceivably have the functionality close to ready on first delivery of the code 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). Now, with keyword testing, it is possible to push automation further up in the schedule. Conceivably, a suite of automated tests could be applied to functionality the first time it comes into test if the automators were supplied with early code releases to write their functions.

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.

There are a number of options available for keyword-driven tools. There are several open-source frameworks available as well as several commercial tools. 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. Over his career, Jamie has created a number of these, starting back in the mid-1990s. While they take a lot of time and effort, he has generally found it to be well worth the investment.

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.

Keywords are generally useful at the system and acceptance test levels.

9.4.7 Keyword Exercise

Refer to the pseudocode recorded script in figure 9-6. This is the same code you saw in figure 9-5.

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 "FREDAPP97 Msg/5" the active window
15. Press the Yes button

Figure 9–6 *WinRunner script translated to pseudocode*

9.4.8 Keyword 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 earlier started the application, 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 (edt_MRnumber), 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 key 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 of the preceding statements 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:

UserID, Password, and DomainName

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 which the front end would likely show as a checkbox.

The next step would appear to be redundant, but it was part of the security on the multilevel system. Lines 9–13 are the same actions as 1–6 on a different window. This is another layer of security allowing the doctor to get into the section of the system that allows her to prescribe 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, UserID, 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 the doctor will click Yes or No. In this case, the message is a nag (informational) message that we want to ignore—hence the Yes.

Our keyword script (so far) would look like this:

	Keyword	*Arg1*	*Arg2*	*Arg3*
1	LoginDomain	"MRE5418"	"kzisnyixmynxfy"	"VN00417"
2	DoTreatment	No		
3	LoginSection	"BC3456"	"dzctmzgtdzbs"	Yes

Finally, notice that we could conceivably pass four arguments to <LoginDomain>, with the fourth being whether to answer Yes or No to the screen that pops up after logging 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.

9.5 Performance Testing

Learning objectives

(K3) Design a performance test using performance test tools including planning and measurements on system characteristics

This section is going to deal with performance tools.

Before performance tools were readily available, we used to simulate performance testing using a technique we 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 we could to see what it would do. Sometimes we could cause it to crash; usually we couldn't. Occasionally, if we could get it to crash once, we could not crash it a second time.

This nonscientific attempt at testing was pretty weak. We could not get meaningful measurements; it was essentially binary: failed | didn't fail. If we did

get a failure, it was rarely repeatable, so we often did not understand what it told us.

Luckily we don't have to do this anymore; in today's world, real performance test tools are prevalent and relatively cheap compared to even 10 years ago. There are now open-source and leased tools as well some incredible vendor tools available.

We talked about the testing aspects of performance testing back in chapter 5. In this chapter, we want to discuss the tool aspect. For the purpose 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.

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 laws that prevent us from using certain data. For example, in the United States, testing with any data that is covered by HIPAA (Health Insurance Portability and Accountability Act) 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.

Generally, production data is much bigger than the available space in the test environment. That raises many questions about how the production data (assuming we can 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).

Luckily 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 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 for 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. This includes the following data:

- User credentials (user IDs and passwords): Reusing credentials during the test may invalidate some of the findings (due to cache 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, etc. 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 and 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 files 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. Too small, and some of the testing will not be meaningful (timing 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 production.

Jamie's experience in performance testing is that the test data is usually smaller than production. Rex has seen many times where the amount of data is similar to production. Clearly, either case is possible. If you have lots of data, great. If not; well, we do the best we can with what we have.

Note that the back-end 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.

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, 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 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 was placed in the script. That data needs to be pulled out and a small amount of programming must be done so that in each place the script used a constant value, it now picks the data up from the data store.

Once the script has been written, it must be tested. Try running it by itself; try running multiple instances of it to make sure the data parameterization was done correctly.

By the time we get to this point, we are almost ready. 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 6, 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 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: Collect and report 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, each one can be found and extensive knowledge discerned by performing a web search.

When dealing with these measurements, remember: we are still testing. That means that expected results are important to compare against the actual values returned.

So now we are all set. We can start our performance test, right? Well, maybe not quite yet. We need to try everything together; 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 stubs its toe and falls over, they can do immediate trouble-shooting to find out why. Often it is just a setting on the server or a tweak to the database that is needed and you are 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 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:

- We can use the big bang, where we just dump everyone onto the system at once (a spike test scenario).

- We may ramp up and down slowly, throwing all of the different transactions into the mix.
- We may 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 an article in *Wired* magazine that discussed how many 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.

9.5.1 Performance Testing Exercise

Given the efficiency requirements in the HELLOCARM system requirements document, determine the actual points of measurement and 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.

9.5.2 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, 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 (2.5 seconds), it would likely be close enough.

The second, 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 that in this case, 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 we get a return on the wire. 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, etc. 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 HELLOCARMS. 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 purpose 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.

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 per 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 runs, we would graph out the metrics that had been captured from the Database Server. Had the server CPU reached 80% 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 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 making sure we are comparing apples to apples and oranges to oranges. We fear that without more information, we might just be measuring an entire fruit salad in this test.

9.6 Sample Exam Questions

1. 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 do you think 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.

2. You 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 staff 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. Which of the following points of information about your organization would tend to make keyword-driven automation a desirable automation method rather than 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.

4. 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. Ensured that the test environment is identical to the production environment
   B. Modeled the system to learn how it's actually used
   C. Purchased or rented enough virtual user licenses to match peak usage
   D. Brought in experienced performance testers to train all of the participants

5. 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 needed by your project. You have very little budget for tools or people, and the timeframe for the project is about to be shortened. 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

10 People Skills and Team Composition

> *"Decimation: punishment in the Roman army. Of every ten soldiers, one was executed.… After a very serious offense (e.g., mutiny or having panicked), the commander of a legion would take the decision, and an officer would go to the subunit that was to be punished. By lot, he chose one in ten men for capital punishment. The surviving nine men were ordered to club the man to death."*
>
> Motivational management techniques of the Roman army, explained by Jona Lendering's article in Livius: Articles on Ancient History (www.livius.org).

The 10th chapter of the Advanced syllabus is concerned with people skills and test team composition. The chapter starts with the skills of the individual tester, then moves to internal and external test team dynamics. It concludes with discussions of motivating testers and test teams and with communicating testing results. Chapter 10 of the Advanced syllabus has six sections.

1. Introduction
2. Individual Skills
3. Test Team Dynamics
4. Fitting Testing within an Organization
5. Motivation
6. Communication

Most of these sections are primarily the domain of test managers. However, let's look at each section, particularly the section on communication, and how it relates to technical test analysis.

10.1 Introduction

Learning objectives

Recall of content only

This chapter is focused primarily on test management topics related to managing a test team. Thus, it is mainly the purview of Advanced Test Manager exam candidates. Since this book is for technical test analysts, most of our coverage in this chapter is for simple recall.

However, it is important for all testers to be mindful of their relationships while doing test work. How effectively you communicate with your peers will influence your success as a tester. Of course, you should also be sure to improve your hard skills over time so that you become a better tester.

10.2 Individual Skills

Learning objectives

Recall of content only

The concepts in this section apply primarily for test managers. There are no learning objectives defined for technical test analysts in this section. In the course of studying for the exam, read this section in chapter 10 of the Advanced syllabus for general recall and familiarity only.

10.3 Test Team Dynamics

Learning objectives

Recall of content only

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

ISTQB Glossary

independence of testing: Separation of responsibilities, which encourages the accomplishment of objective testing.

course of studying for the exam, read this section in chapter 10 of the Advanced syllabus for general recall and familiarity only.

10.4 Fitting Testing within an Organization

Learning objectives

Recall of content only

The concepts in this section apply primarily for test managers. There are no learning objectives defined for technical test analysts in this section. In the course of studying for the exam, read this section in chapter 10 of the Advanced syllabus for general recall and familiarity only.

10.5 Motivation

Learning objectives

Recall of content only

The concepts in this section apply primarily for test managers. There are no learning objectives defined for technical test analysts in this section. In the course of studying for the exam, read this section in chapter 10 of the Advanced syllabus for general recall and familiarity only.

10.6 Communication

Learning objectives

(K2) Describe by example professional, objective, and effective communication in a project from the tester perspective, considering risks and opportunities.

There are three levels of communication for most test teams:

- First, we communicate, mostly internally but also with others, about the documentation of test products. This includes discussions of test strategies, test plans, test cases, test summary reports, and defect reports.
- Second, we communicate feedback on reviewed documents, typically on a peer level both inside and outside the test group. This includes discussions about requirements, functional specifications, use cases, and unit test documentation.
- Third, we communicate as part of information gathering and dissemination. This can include not just peer level communications, but communications to managers, users, and other project stakeholders. It can be sensitive, as when test results are not encouraging for project success.

It's important to remember that both internal and external communications affect the professionalism of technical test analysts.

Effective communication assists you in achieving your objectives as a technical test analyst, while ineffective communication will hinder you. It's important to be professional, objective, and effective. You want each communication you have, both inside and outside the test team, to build and maintain respect for the test team. When communicating about test results, giving feedback on issues with documents, or delivering any other potentially touchy news, make sure to use diplomacy.

It's easy to get caught up in emotions at work, especially during test execution when things are often stressful. Remember to focus on achieving test objectives. Remember also that you want to see the quality of products and processes improved. Don't engage in communication that is contrary to those goals.

It's also easy to communicate as if you were communicating with yourself or someone like you. In other words, we testers often speak in a sort of shorthand about very fine-grained details of our work and findings, and with a certain degree of skepticism. When talking to fellow testers, this is fine. However, you have to remember to tailor communication for the target audience. When talking to users, project team members, management, external test groups, and customers, you need to think carefully about how you are communicating, what you are communicating, and whether your communications support your goals.

As test managers, we have seen a single thoughtless e-mail, bug report, or hallway conversation do a great deal of damage to a test team's reputation and credibility. So even with all the other work you have to do, remember to think about your communications.[1]

Figure 10-1 shows an example of test communication. This is an excerpt of an e-mail to a vendor about the results of acceptance testing of the RBCS website. The first paragraph is to communicate that this is a carefully thought-out analysis, not just one of the dozens of "fired-off" e-mails someone is likely to get. The message is, "Pay attention to this e-mail, please, because I did." This paragraph also refers the reader to further details in the attached document.

The second paragraph—including the bulleted list and closing sentence—summarizes what needs to be done to complete the acceptance testing and move into deployment. The third paragraph clarifies the meaning of the deferral of certain bugs. Rex wanted to make sure RBCS was not waiving any legal rights, so RBCS had to insist on these problems being fixed later. The final paragraph is a subtle hint that RBCS staff members were disappointed to be still finding problems.

1. You can find an excellent discussion of people issues in testing in Judy McKay's *Managing the Test People* ***(Rocky Nook, 2007)***

I have spent a couple of hours reviewing the current status of the site and the acceptance test. Please see attaches [documents] with deferred bug reports and [test status]…for pass 2.

The following issues are must-fix to move forward with Deployment:

- Consistency of meeages and UI (for examples, see bug 85, 91, 92, 95, 97)
- Newsletter link…not in place as agreed (see bug 98)
- Identification and resolution of internal dead links (see bug 103)

While some of these issues might strike the casual reader as picayune, please understand that our target customers…are sensitive to any errors…

In the interests of moving forward and having this critical marketing collateral in place…I have agreed to defer a number of bugs from pass 1 that either failed verification testing or which related advertised product features [the vendor] retroactively and unilaterally withdrew… Please note that deferral of these bugs does not indicate acceptance by RBCS of the disposition of those bugs for all time.

Finally, please note that there were eighteen (18) new bug reports filed during the second pass.

Figure 10–1 *Acceptance test status e-mail*

Now, this type of e-mail is appropriate for a customer to send to a vendor, explaining test results. Would you send it to your development colleagues? Probably not. The important point here is that every word and every sentence of that e-mail had a communication objective.

10.7 Sample Exam Questions

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

1. Assume you are a technical test analyst working on a banking project to upgrade an existing automated teller machine system to allow customers to obtain cash advances from supported credit cards. You are unable to obtain information about the minimum and maximum throughput of the connections between the automated teller machine and the payment processing networks, which is not included in the design specification. Which of the following is an example of a good way to communicate that problem in an e-mail?

A “Until I receive a complete specification of throughput between ATMs and the network, no progress on test design can occur.”

B “When will it be possible for us to know the minimum and maximum throughput of the ATM/network connections? Test design is impeded by a lack of clarity here.”

C “Here we go again. The design specification is incomplete and ambiguous.”

D Do not communicate the problem; just log the delaying effect of the information problem and be ready to explain the delays to management when they ask.

11 Preparing for the Exam

"Because I am hard, you will not like me. But the more that you hate me, the more you will learn."

Senior Drill Instructor to U.S. Marine boot camp attendees, in Stanley Kubrick's Vietnam film, Full Metal Jacket.

The 11th chapter of this book is concerned with topics that you need to know to prepare for the ISTQB Advanced Level Technical Test Analyst exam. The chapter starts with a discussion of the ISTQB Advanced Level Technical Test Analyst learning objectives, which are the basis of the exams.

Chapter 11 of this book has two sections.

1. Learning objectives
2. ISTQB Advanced exams

If you are not interested in taking the ISTQB Advanced Level Technical Test Analyst exam, this chapter might not be pertinent for you.

11.1 Learning Objectives

Each of the Advanced syllabus exams is based on learning objectives. A learning objective states what you should be able to do prior to taking an Advanced exam. Each Advanced exam has its own set of learning objectives. 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: remembering, understanding, application, and analysis. Exam questions will be structured so that you must have achieved these learning objectives to determine the

correct answers for the questions. The exam will cover the more basic levels of remembrance and understanding implicitly as part of the more sophisticated levels of application and analysis. For example, to answer a question about how to create a test plan, you will have to remember and understand the IEEE 829 test plan template. So, unlike the Foundation exam, where simple remembrance and understanding often suffice to determine the correct answer, on an Advanced exam, you will have to apply or analyze the facts that you remember and understand in order to determine the correct answer.

Let's take a closer look at the four levels of learning objectives you will encounter on the Advanced exam. The tags K1, K2, K3, and K4 are used to indicate these four levels, so remember those tags as you review the Advanced syllabus.

11.1.1 Level 1: Remember (K1)

At this lowest level of learning, the exam will expect that you can 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:

- Nondelivery of service to an end user or any other stakeholder
- 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 syllabus and also standards like ISO 9126 and IEEE 829 that are referenced in the Advanced syllabus. Expect this level of learning to be required for questions focused on higher levels of learning like K3 and K4.

11.1.2 Level 2: Understand (K2)

At this second level of learning, the exam will expect that you can select the reasons or explanations for statements related to the topic and can 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 under-

stand a test procedure, such as explaining the sequence of tasks. Watch for keywords such as *summarize*, *classify*, *compare*, *map*, *contrast*, *exemplify*, *interpret*, *translate*, *represent*, *infer*, *conclude*, and *categorize*.

For example, you should be able to explain the reason 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 non-functional 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 syllabus and the proper use of standards like ISO 9126 and IEEE 829 that are referenced in the Advanced syllabus. Expect this level of learning to be required for questions focused on higher levels of learning like K3 and K4.

11.1.3 Level 3: Apply (K3)

At this third level of learning, the exam will expect that you can 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 expect to evaluate a software application or create a testing model for a given software application. If the syllabus gives a model, the coverage requirements for that model, and the procedural steps to create test cases from a model in the Advanced syllabus, then you are dealing with a K3 learning objective. Watch for keywords such as *implement*, *execute*, *use*, *follow a procedure*, and *apply a procedure*.

For example, you should be able to do the following:

- Identify boundary values for valid and invalid equivalence partitions.
- Use the generic procedure for test case creation to select the test cases from a given state transition diagram (and a set of test cases) in order to cover all transitions.

This means that you should be able to apply the techniques described in the ISTQB Advanced syllabus to specific exam questions. Expect this level of learning to include lower levels of learning like K1 and K2.

11.1.4 Level 4: Analyze (K4)

At this fourth level of learning, the exam will expect that you can separate information related to a procedure or technique into its constituent parts for better understanding and can distinguish between facts and inferences. A typical exam question at this level will require you to analyze a document, software, or project situation and propose appropriate actions to solve a problem or complete a task. Watch for keywords such as *analyze*, *differentiate*, *select*, *structure*, *focus*, *attribute*, *deconstruct*, *evaluate*, *judge*, *monitor*, *coordinate*, *create*, *synthesize*, *generate*, *hypothesize*, *plan*, *design*, *construct*, and *produce*.

For example, you should be able to do the following:

- Analyze product risks and propose preventive and corrective mitigation activities.
- Describe which portions of an incident report are factual and which are inferred from results.

This means that you should be able to analyze the techniques and concepts described in the ISTQB Advanced syllabus to answer specific exam questions. Expect this level of learning to include lower levels of learning like K1, K2, and perhaps even K3.

11.1.5 Where Did These Levels of Learning 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 fairly standard educational fare, though you probably haven't encountered it unless you've been involved in teaching training courses.

As a practical matter, we recommend thinking 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 projects.

As you can see, there is an upward progression of ability that adheres to each increasing level of learning. Much of the focus at the Advanced level is on application and analysis.

11.2 ISTQB Advanced Exams

Like the Foundation exam, the Advanced exams are multiple-choice exams. Multiple-choice questions consist of three main parts. The first part is the stem, which is the body of the question. The stem may include a figure or table as well as text. The second part is the distracters, which are choices that are wrong. If you don't have a full understanding of the learning objectives that the question covers, you might find the distracters reasonable choices. The third part is the answer or answers, which are 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 only covered implicitly within the higher-level questions.

For example, the Foundation exam might typically include a question like this:

Which of the following is a major section of an IEEE 829 compliant test plan?

A. Test items
B. Probe effect
C. Purpose
D. Expected results

The answer is A, while B, C, and D are distracters. 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 first learning a subject. However, the ability to recall facts does not make you an expert, any more than our ability to recall song lyrics from the 1970s qualifies us to work as singers 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 an incident report describing the time and date at which you first noticed the missing test release
B. Creating a test procedure specification that describes how to install a test release
C. Sending an SMS text to the configuration management team manager
D. Send an e-mail to the project manager and the configuration management team manager

The answer is C. A, B, and D are distracters. A is wrong because it is not that the release didn't install, it's that it didn't even arrive. B is wrong because, while such a test procedure might be useful for installation testing, it has nothing to do with the escalation process described in the test plan. C is consistent with the test plan. D is not consistent with the test plan because the spirit of the one-hour delay described in the test plan excerpt is that the configuration management team manager should have a chance to resolve the problem before the project manager is engaged. In addition, when time is of the essence, e-mail is not a good escalation technique.

As you can see, this kind of question requires analysis of a situation. Yes, it helps to know what the IEEE 829 templates such as the test plan, incident report, test item transmittal report, and test procedure specification contain. In fact, you'll probably get lost in the terminology if you don't know the standard. However, simply knowing the IEEE 829 standard will not allow you to get the right answer on this question except by chance.

11.2.1 Scenario-Based Questions

Further complicating this situation is the fact that many exam questions will actually consider a scenario. In scenario-based questions, the exam will describe a set of circumstances. It will then present you with a sequence of two, three, or even more questions.

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 mes-

sage is presented on the database server in Cyrillic script, though the chosen language is U.S. English. At that point, the database tables are corrupted and any attempt to use the application under test results in various database connection error messages (which are at least presented in U.S. English).

Consider the following possible actions:

I. Notifying the configuration management team manager
II. Notifying the project manager
III. Filing an incident 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 distracters. A is wrong because there is no incident report filed, which is required by the test plan when the installation fails. C is wrong because meaningful testing cannot continue against the corrupted database, because the project manager is notified before the configuration management team manager, and because the incident report is filed before an attempt to reproduce the failure has occurred. D is wrong because the project manager is notified before the configuration management team manager and because no incident report is filed.

As you can see, with a scenario-based question it's very important that you study the scenario carefully before trying to answer the questions that relate to it. If you misunderstand the scenario—perhaps due to a rushed reading of it—you can anticipate missing most if not all of the questions related to it.

Let 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 questions. There's an added complication with K1 learning objectives: They are not explic-

itly defined. The entire syllabus, including glossary terms used and standards referenced, is implicitly covered by K1 learning objectives. Here is an excerpt from the Advanced syllabus:

> "This syllabus' content, terms and the major elements (purposes) of all standards listed shall at least be remembered (K1), even if not explicitly mentioned in the learning objectives."

So, you'll want to read the Advanced syllabus carefully, a number of times, as you are studying for the Advanced exam.

Not only should you read the Advanced syllabus, but you'll need to go back and refresh yourself on the Foundation syllabus. Again, as excerpt from the Advanced syllabus:

> "All Advanced Certificate examinations must be based on this syllabus and on the Foundation Level syllabus. Answers to examination questions may require the use of material based on more than one section of this and the Foundation syllabus. All sections of this and the Foundation syllabus are examinable."

Notice that the second sentence in the preceding paragraph means that a question can conceivably cross-reference two or three sections of the Advanced syllabus or cross-reference a section of the Advanced syllabus with the Foundation syllabus. So, it would be smart to take a sample Foundation exam and reread the Foundation syllabus as part of studying for the Advanced exam.

11.2.2 On the Evolution of the Exams

The structure of the Advanced exams continues to evolve. Further, note the following somewhat insidious paragraph tucked away in the Advanced syllabus:

> "The format of the examination is defined by the Advanced Exam Guidelines of the ISTQB. Individual Member Boards may adopt other examination schemes if desired."

We have written this chapter based on the ISTQB Advanced Exam Guidelines. We assume that most ISTQB national boards and exam boards will choose to follow those. However, based on this paragraph in the Advanced syllabus, exams that differ from the ISTQB Exam Guidelines and thus what is described

in this chapter can be created by some boards. You'll want to check with the national board or exam board providing your exam to be sure.

Okay, having read this, you might be panicking. Don't! Remember, the exam is meant to test your achievement of the learning objectives in the Advanced syllabus. This book contains solid features to help you do that:

- Did you work through all the exercises in the book? If so, you have a solid grasp on the most difficult learning objectives, the K3 and K4 objectives. If not, go back and do so now.
- Did you work through all the sample exam questions in the book? If so, you have tried a sample exam question for most of the learning objectives in the syllabus. If not, go back and do so now.
- Did you read the ISTQB glossary term definitions where they occurred in the chapters? If so, you are familiar with these terms. If not, return to the ISTQB glossary now and review those terms.
- Did you read every chapter of this book and the entire ISTQB Advanced syllabus? If so, you know the material in the ISTQB Advanced syllabus. If not, review the ISTQB Advanced 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 syllabus, you will be in good shape for the exam.

Good luck to you when you take the exam, and the best of success when you apply the ideas in the Advanced syllabus to your next testing project.

Appendix A

Bibliography

Advanced Syllabus Referenced Standards

British Computer Society. BS 7925-2 (1998), Software Component Testing.

Institute of Electrical and Electronics Engineers. IEEE Standard 829 (1998/2007), IEEE Standard for Software Test Documentation.

Institute of Electrical and Electronics Engineers. IEEE Standard 1028 (1997), IEEE Standard for Software Reviews.

Institute of Electrical and Electronics Engineers. IEEE Standard 1044 (1993), IEEE Standard Classification for Software Anomalies.

International Standards Organization. ISO/IEC 9126-1:2001, Software Engineering – Software Product Quality.

International Standards Organization. ISO/IEC 9126-2:2003, Software Engineering – Product Quality Part 2: External Metrics.

International Standards Organization. ISO/IEC 9126- 3:2003, Software Engineering – Product Quality Part 3: Internal Metrics.

International Software Testing Qualifications Board. ISTQB Glossary (2007), ISTQB Glossary of Terms Used in Software Testing, Version 2.0.

US Federal Aviation Administration. DO-178B/ED-12B, Software Considerations in Airborne Systems and Equipment Certification.

Advanced Syllabus Referenced Books

Bath, Graham, and Judy McKay. *The Software Test Engineer's Handbook*. Rocky Nook, 2008.

Beizer, Boris. *Software Testing Techniques*. ITP, 1990.

Beizer, Boris. *Black-Box Testing*. Wiley, 1995.

Black, Rex. *Managing the Testing Process (Second Edition)*. Wiley, 2002.

Black, Rex. *Critical Testing Processes*. Addison-Wesley, 2003.

Black, Rex. *Pragmatic Software Testing*. Wiley, 2007.

Black, Rex. *Advanced Software Testing, Vol. 1*. Rocky Nook, 2009.

Buwalda, Hans. *Integrated Test Design and Automation*. Addison-Wesley, 2001.

Burnstein, Ilene. *Practical Software Testing*. Springer, 2003.

Copeland, Lee. *A Practitioner's Guide to Software Test Design*. Artech House, 2003.

Craig, Rick, and Stefan Jaskiel. *Systematic Software Testing*. Artech House, 2002.

Dustin, Elfriede. *Effective Software Testing*. Addison Wesley, 2003.

Gerrard, Paul, and Neil Thompson. *Risk-Based e-Business Testing*. Artech House, 2002.

Gilb, Tom, and Dorothy Graham. *Software Inspection*. Addison-Wesley, 1993.

Graham, Dorothy, Erik van Veenendaal, Isabel Evans, and Rex Black. *Foundations of Software Testing*. Thomson Learning, 2007.

Grochmann, M. "Test Case Design Using Classification Trees." *Conference Proceedings of STAR*, 1994.

Jorgensen, Paul. *Software Testing: A Craftsman's Approach (Second Edition)*. CRC Press, 2002.

Kaner, Cem, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing*. Wiley, 2002.

Koomen, Tim, and Martin Pol. *Test Process Improvement*. Addison-Wesley, 1999.

Marick, Brian. *The Craft of Software Testing*. Prentice Hall, 1995.

Molyneaux, Ian. *The Art of Application Performance Testing*. O'Reilly, 2009.

Myers, Glenford. *The Art of Software Testing*. Wiley, 1979.

Pol, Martin, Ruud Teunissen, and Erik van Veenendaal. *Software Testing: A Guide to the T-map Approach*. Addison-Wesley, 2002.

Splaine, Steven, and Stefan Jaskiel. *The Web-Testing Handbook*. STQE Publishing, 2001.

Stamatis, D. H. *Failure Mode and Effect Analysis*. ASQ Press, 1995.

van Veenendaal, Erik, ed. *The Testing Practitioner*. UTN Publishing, 2002.

Whittaker, James. *How to Break Software*. Addison-Wesley, 2003.

Whittaker, James, and Herbert Thompson. *How to Break Software Security*. Addison-Wesley, 2004.

Wiegers, Karl. *Software Requirements (Second Edition)*. Microsoft Press, 2003.[1]

Other Referenced Books

Beizer, Boris. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, 1984.

Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley Professional, 2000.

Koomen, Tim, et al. *TMap Next*. UTN Publishers, 2006.

McKay, Judy. *Managing the Test People.* Rocky Nook, 2007.

Nielsen, Jakob. *Usability Engineering*. Academic Press, 1993.

Tufte, Edward. *The Graphical Display of Quantitative Information, 2e*. Graphics Press, 2001.

Tufte, Edward. *Visual Explanations*. Graphics Press, 1997.

Tufte, Edward. *Envisioning Information*. Graphics Press, 1990.

White, Gregory, et al. *Security + Certification*. Osborne, 2003.

Other References

Badlaney, Janvi, Ghatol, Rohit , and Jadhwani, Romit. "An Introduction to Data-flow Testing." Dept. of Computer Science, North Carolina State University.

1. In an omission, this book is not included in the Advanced syllabus bibliography but it is referenced in the Advanced syllabus text. Therefore, we have included it here.

Holmes, Jeff. "Identifying Code-Inspection Improvements Using Statistical Black Belt Techniques." *Software Quality Professional*, December 2003, Volume 6, Number 1.

Black, Rex, and Greg Kubaczkowski. "Mission Made Possible." *Software Testing and Quality Engineering*, July/August 2002, Volume 4, Issue 4.

"Structural Coverage Metrics." IPL (available at IPL.com).

dictionary.com, for standard English words.

Appendix B

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, authorized RBCS course attendees, and readers of this book.

I Table of Contents

I Table of Contents 551
II Versioning 553
III Glossary 555
000 Introduction 557
001 Informal Use Case 558
003 Scope 559
004 System Business Benefits 561
010 Functional System Requirements 562
020 Reliability System Requirements 566
030 Usability System Requirements 567
040 Efficiency System Requirements 568
050 Maintainability System Requirements 570
060 Portability System Requirements 571
A Acknowledgement 573

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	

III Glossary

Term[1]	Definition
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 *home equity loans*, *home equity lines of credit*, and *reverse mortgages* (see definitions below).
Secured Loan	Any loan for which the borrower uses an asset as collateral. The loan is secured by the collateral in that the borrower can make a legal claim on the collateral if the borrower fails to repay the loan.
Home Equity Loan	A lump sum of money disbursed at the initiation of the loan and lent to the homeowner at interest. A home equity loan is a secured loan, secured by the equity in the borrower's home.
Home Equity Line of Credit	A variable amount of money with a prearranged maximum amount available for withdrawal by the homeowner on an as-needed basis and lent to the homeowner at interest. A home equity line of credit allows the homeowner to take out, as needed, a secured loan, secured by the equity in the borrower's home.
Mortgage	A legal agreement by which a sum of money is lent for the purpose of buying property and against which property the loan is secured.
Reverse Mortgage	A mortgage in which a homeowner borrows money in the form of regular payments which are charged against the equity of the home, typically with the goal of using the equity in the home as a form of retirement fund. A reverse mortgage results in the homeowner taking out a regularly increasing secured loan, secured by the equity in the borrower's home.

1. These definitions are adapted from www.dictionary.com.

000 Introduction

The Home Equity Loan, Line-of-Credit, and Reverse Mortgage System (HELLOCARMS), as to be deployed in the first release, allows Globobank Telephone Bankers in the Globobank Fairbanks call center to accept applications for home equity products (loans, lines of credit, and reverse mortgages) from customers. The second release will allow applications over the Internet, including from Globobank business partners as well as customers themselves.

At a high level, the system is configured as shown in Figure A-1. The HELLOCARMS application itself is a group of Java programs and assorted interfacing glue that run on the Web server. The Database server provides storage as the application is processed, while the Application server offloads gateway activities to the clients from the Web server.

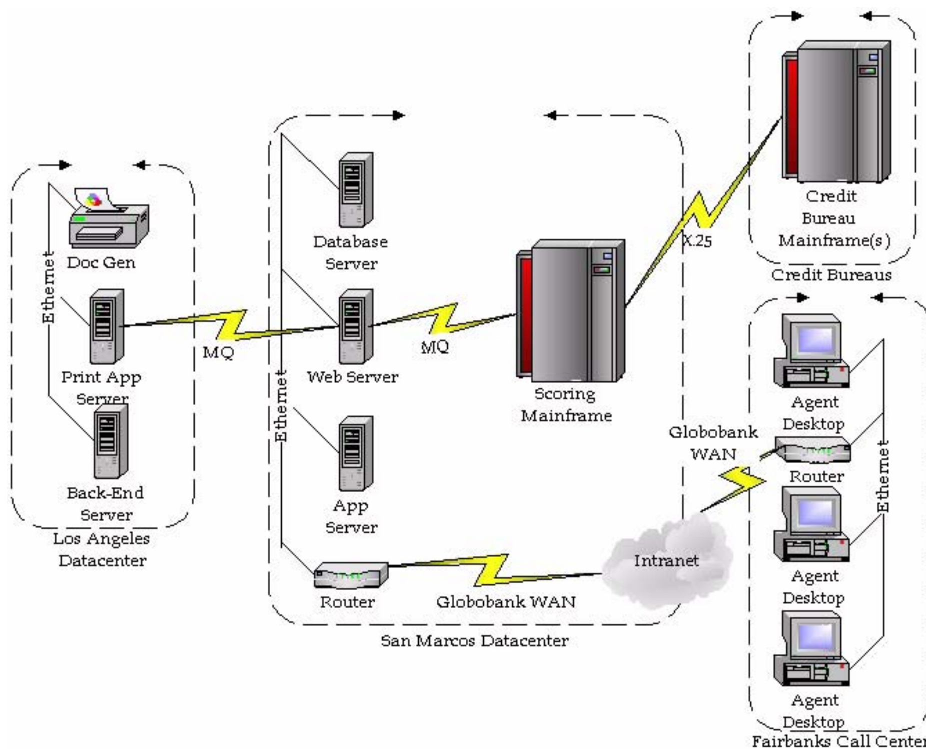


Figure A-1 HELLOCARMS system (first release)

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 when the following events occur:
    a. LoDoPS sends documents to customer.
    b. Globobank Loan Servicing Center receives signed documents from customer.
    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, HELLO-CARMS 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 the following:

- 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 provide access from only a Globobank call center (specifically Fairbanks).
- Developing an interface to Globobank's existing Scoring Mainframe for scoring a customer based on their loan application and HELLOCARMS features.
- Developing an interface to use Globobank's existing underwriting and origination system, Loan Document Printing System (LoDoPS), for document preparation. This interface allows the HELLOCARMS System, after assisting the customer with product selection and providing preliminary approval to the customer, to forward the preapproved application (for a loan, line of credit, or reverse mortgage) to the LoDoPS and to subsequently track the application's movement through to the servicing system.
- Receiving customer-related data from the Globobank Rainmaker Borrower Qualification Winnow (GloboRainBQW) system to generate outbound offers to potential (but not current) Globobank customers via phone, e-mail, and paper mail.

004 System Business Benefits

The business benefits associated with the HELLOCARMS are as follows:

- Automating a currently manual process, and allowing loan inquiries and applications from the Internet and via call center personnel (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 the 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[2]
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 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

2. Priorities are:
   1 Very high
   2 High
   3 Medium
   4 Low

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 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 preapproved 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

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	Add a loan fee based on property type: • 1.5% for rental properties (duplex, apartment, and vacation) • 2.5% for commercial properties • 3.5% for condominiums or cooperatives • 4.5% for undeveloped property Do not add a loan fee for the other supported property type, residential single family dwelling.	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	*Interoperability*	
010-030-010	If the customer provides a “How Did You Hear About Us” identifier code during the application process, retrieve customer information from GloboRainBQW.	2
010-030-020	Accept joint applications (e.g., partners, spouses, relatives, etc.) and score all applicants using the Scoring Mainframe.	1
010-030-030	Direct Scoring Mainframe to remove duplicate credit information from joint applicant credit reports.	2
010-030-040	Allow user to indicate on a separate screen which, if any, are existing debts that the customer will retire using the funds for which the customer is applying. Allow user the option to exclude specific debts and to include specific debts. For debts to be retired, send a stipulation to LoDoPS that specifies which debts 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 the customer must pay with loan proceeds.	3

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	*Security*	
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	*Compliance (functionality standards/laws/regs)*	
	[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	*Maturity*	
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 fewer than five (5) failures per month in production.	2
020-010-030	The system shall average fewer than one (1) failure per month in production.	4
020-020	*Fault-tolerance*	
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	*Recoverability*	
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	*Compliance (reliability standards/laws/regs)*	
	[To be determined in a subsequent revision]	

030 Usability System Requirements

The capability of the system to be understood, learned, used, and attractive to the user and the call center agents when used under specified conditions.

ID	Description	Priority
030-010	*Understandability*	
030-010-010	Support the submission of applications via the Internet, including the capability for untrained users to properly enter applications.	2
030-010-020	All screens, instructions, help, and error messages shall be understandable at an eighth 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	*Learnability*	
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 for an average user to become proficient in the use of HELLOCARMS within 8 hours of training.	3
030-030	*Operability*	
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-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	*Attractiveness*	
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 the US and countries outside the US where the system is used.	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	*Time behavior*	
040-010-010	Provide the user with screen-to-screen response time of 1 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 1 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 they were 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
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]	
040-020	*Resource utilization*	
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
040-030	*Compliance (performance standards)*	
	[To be determined in a subsequent revision]	

050 Maintainability System Requirements

The capability of the system to be modified. Modifications may include corrections, improvement, or adaptations of the software changes in environments and in requirements and functional specifications.

ID	Description	Priority
050-010	*Analyzability*	
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	HELLOCARMS 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	*Changeability*	
050-020-010	Modification complexity (see ISO 9126-2) for all changes to HELLOCARMS shall be measured and tracked throughout the SMLC.	
050-030	*Compliance (performance standards)*	
	[To be determined in a subsequent revision]	

060 Portability System Requirements

The capability of the system to be transferred from one environment to another.

ID	Description	Priority
060-010	*Adaptability*	
060-010-010	HELLOCARMS shall be configured to work on Internet Explorer browsers in the current version and at least one level backwards.	1
060-010-020	HELLOCARMS shall be configured to work with Internet Explorer and Firefox in the current version and at least one level backwards.	2
060-010-030	HELLOCARMS shall be configured to work with all popular browsers that represent 5 percent or more of the currently deployed browsers in any countries where Globobank does business.	3
060-020	*Installability*	
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
060-030	*Co-existence*	
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]	
060-040	*Replaceability*	
	Not applicable	
060-050	*Compliance*	
	[To be determined in a subsequent revision]	

A Acknowledgement

This document is based on an actual project. RBCS would like to thank its client, who wishes to remain unnamed, for permission to adapt and publish anonymous portions of various project documents.

Appendix C – Answers to Sample Questions

Chapter 1

1 C and D

2 C

Chapter 2

1 B and C

2 A

3 B

4 D

Chapter 3

1 B

2 C

3 C

Chapter 4

1 A

2 C

3 C

4 B

5 A

6 D

7 B

8 C

9 A

10 D

11 B

12 D

13 C

14 B

15 A

16 D

17 A

18 D

19 A

20 A

Chapter 5

1 C

2 B

3 D

4 D

5 B

6 B

7 A

Chapter 6

1 B

2 D

3 A

4 C

Chapter 7

1	D
2	A
3	A
4	C
5	B

Chapter 9

1	C
2	D
3	C
4	B
5	A

Chapter 10

1	B

Index

Numerics
0-switch columns 167
0-switch coverage 167
1-switch coverage 169

A
acceptance review 410
acceptance test variations 4
accessibility testing 331
accuracy 326
accuracy testing 326
adaptability 387
Advanced exams 539
Advanced Level Technical Test Analyst 50
Advanced Level Test Analyst 50
agile lifecycle 3
all c-uses (ACU) strategy 287
all c-uses testing 288
all defs (AD) strategy 288
all du path strategy (ADUP) 286
all p-uses (APU) strategy 287
all p-uses testing 288
all-definitions (AD) strategy 287
all-uses (AU) 286
amateur testers 47
anomaly 45, 440
API misuse detection 308
architecture 461
arrays 120, 122
    multidimensional 123
assessing impact 83
assessing likelihood 82
assessment results
    example 87
ATA 103
atomic actions 276
    define last 278
    define-define 277
    define-kill 276
    define-use 276
    first define 276
    first kill 277
    first use 277
    kill last 278
    kill-define 277
    kill-kill 277
    use-define 277
    use-kill 277
    use-use 277
atomic conditions 198
ATTA 103
audit 403, 408
    automation 471
author-bias problem 90
automated test framework 467
automated testing
    consistency 468
    reuse 468
automated tests 461
automating test execution 461
automation
    business case 461
    evolution 490
automation benefits 462
availability 350

B
background testing 359
basic security attacks 257
Basili, Victor 13
basis paths 220
basis test set 220
Bath, Graham 357
BCD representation 127
behavior-based techniques 262
Beizer, Boris 133, 184, 194, 227, 237
Beizer's guidelines 195
Beizer's methodology 133
binary representation 127
binary-coded decimal (BCD) 126

black box 49
black-box techniques 49
black-box testing 105, 178, 179
blended test strategies 41
BMI and age boundary values 150
Boolean operators 189
boundaries for integers 124
boundary testing for accuracy 328
boundary value 119
  exercise 134
boundary value analysis 119
  examples 120
  underlying model 119
boundary value analysis on equivalence classes 120
boundary values
  number of 133
branch coverage 188
branch testing 188
breadth-first approach 86
breakdown of defects 57
breaking encryption 343
BS 7925/2 process 51
BS 7925/2 standard 49
buffer overflow 341
bug hypothesis 106
bug taxonomy 241
business case 414
business value of testing 74

C

C programming language 127
call-graph based integration testing 290, 292
Capability Maturity Model (CMM) 458
Capability Maturity Model Integration (CMMI) 458
capture/replay
  evolution 497
  exercise 495
capture/replay architecture 501
capture/replay failures 497
capture/replay framework 501
capture/replay tools 475
career as a tester 15
cause-effect graphing 49
change management 3
checklist testing 243
  bug hypothesis 244
  data items 245
  declarations 244
Chow's switch coverage 155
Clausewitz 46
CMM 458
CMMI 458
code of ethics 15
code parsing tools 268
  Splint 268
code review
  exercise 423
coexistence testing 392
cohesion 381
Common Attack Pattern Enumeration and Classification (CAPEC) 345
Common Vulnerabilities and Exposures (CVE) 345
communicating incidents 450
communication 530
  levels 530
completeness 54
completeness of coverage 160
complexity analysis 265, 268
complexity view 267
component integration testing 9
component testing 9
condition coverage 197
  bug hypothesis 198
condition determination testing 200
condition testing 197
conditional call 294
configuration control board 451
configuration item 451
configuration management 3, 451
control-flow analysis 265
control-flow graph (example) 182
control-flow graphs 179, 180
  decision point 182
  junction point 181
  process block 181
control-flow testing 179
  condition coverage 180
  condition determination 180
  decision (or branch) coverage 180
  decision/condition coverage 180
  exercise 209
  instruction and code coverage 179
  Linear Code Sequence and Jump (LCSAJ) 180
  loop coverage 180
  MC/DC 180
  multiple condition/decision coverage 180

multiple-condition coverage 180
statement coverage 179
correctness 327
coupling 381
coverage criterion 108
coverage metrics 288
Critical Testing Processes (CTP) 458
criticality 51
criticality level 52
CTP 458
customer product integration testing 4
cyclomatic complexity 221, 223, 266
example 222
exercise 225
cyclomatic complexity testing 220

D

data defects 239
initial value 239
structure 239
type 239
data parameterization 517
data scoping rules 274
data transfer interception 342
data-driven architecture 502
data-driven testing 489
data-flow
exercise 284
data-flow analysis 273, 275
data-flow errors 273
data-flow strategies 285
data-flow testing 273
dd-path 215
dd-path testing 216
DEADBEEF 304
debugging tool 479
decision constructs 190
decision coverage 188, 190
example 191
decision coverage vs. statement coverage 189
decision table 140, 142
collapsing columns 143
test cases 140
decision table testing 140
decision tables
and boundary values 146
and equivalence partitions 146
combining with other techniques 145
exercise 147, 148
nonexclusive rules 147
decision testing 188
decision/condition coverage 200
default value integer 127
defect 437
defect breakdown 56
defect fields 445
defect lifecycle 437
defect taxonomy 237, 238
example 237
defect-based techniques 236, 237
applying 261
defects 436
defects per thousand lines of source code (KLOC) 352
defect-taxonomy-based techniques 236
Delphi 127
denial of service 341
depth-first approach 86
design review 410
design specification template 32
detect defects 436
detection methods 88
Deutsch checklist review
exercise 429
Deutsch, L. Peter 417
Deutsch's design review checklist 417
distributed testing 74
DO-178B standard 51, 90
documenting test conditions 25
domain specific languages 469
dynamic analysis 302
dynamic analysis tools 303, 480, 482
dynamic tests 102
black-box 102
defect-based 103
dynamic analysis 103
experience-based 103
white-box 103

E

e-commerce application (example) 159
effective attacks (example) 256
effectiveness 332
effects analysis 97
efficiency 332, 355, 374
exercise 372
resource utilization 356
time behavior 356
efficiency bugs (examples) 368
efficiency failures 356

efficiency measurements 366
 metrics 367
efficiency testing 355
 multiple flavors 357
emulator 488
enclosed region calculation 222
endurance or soak testing 359
entry criteria 37
enumerations 121
epsilon 132
equivalence partition 108
equivalence partitioning 107, 108
 exercise 115
 invalid equivalence classes 107
 valid equivalence classes 107
 visualizing 109
equivalence partitioning errors 110
 composing test cases 111
 error handler 114
 example 114
error 437
error guessing 242, 243
ethical standards 14
ethics 14
evaluation 333
evaluation of exit criteria 53
exam 535
 evolution 543
 scenario-based questions 541
Example 278
exclusive conditional call 294
exercise
 capture/replay 495
 code review 423
 control-flow testing 209
 cyclomatic complexity 225
 data-flow 284
 Deutsch checklist review 429
 evaluating exit criteria and reporting 60
 exploratory testing 249
 hexadecimal converter 195
 incident management 451
 keyword 511
 maintainability and portability 393
 McCabe design predicate 301
 performance testing 520
 security, reliability, and efficiency 372
 software attack 259
 specification-, defect-, and experience-based 260
 structure-based testing 228
 usability test 335
exit criteria 23
experience-based techniques 41, 237
 applying 261
 error guessing 243
exploratory testing 245
 bug hypothesis 246
 exercise 249
 test charters 247
 usability heuristics 245
exploratory testing process 246
exponent 128

F

factorial 182
failure 45, 437
failure mode 97
Failure Mode and Effect Analysis (FMEA) 81, 87
Failure Mode, Effect and Criticality Analysis (FMECA) 81
failure rate
 confirmation test 57
failures 44
false negative 40
false positive 40
false-fail result 40
false-pass result 40
family of defects 106
fault model 253
 bug hypothesis 253
fault seeding 479
fault seeding tool 479
fault tolerance 349
feature interaction testing 4
fixed-point decimal 128
floating point numbers 128
 boundaries 130
 testing 130
formal review 405
 phases 410
formative evaluation 332
framework 461
 test automation system 461
framework architecture 499
functional boundaries 124
 integers 125
functional decomposition 289
functional defects 238

function 238
specification 238
test 238
functional testing 325

G
goal question metric technique 13
gray-box test 242
GUI 492

H
HALT testing 352
hardware reliability graph 350
hardware-software integration testing 4
HELLOCARMS 549–573
heuristic evaluation 333
hex converter (example) 296
high cohesion 383
high-level test case 27
Highly Accelerated Life Tests (HALT) 352
housekeeping categories 240
hyperlink test tool 487

I
identifying test conditions 25
IEEE 1028 standard
software reviews 401
IEEE 1044 classification process 439
IEEE 1044 defect lifecycle 438
IEEE 1044 incident management lifecycle 438
IEEE 1044 standard
incident lifecycles 435
IEEE 1044.1 incident management system 445
IEEE 754-2008 float representation 129
IEEE 829 standard 48
IEEE 829 standard for test documentation 49, 59
IEEE 829 test case specification 32
IEEE 829 test design specification 31
IEEE 829 test plan template 22
IEEE 829 test procedure specification 48
impact of a problem 76
incident 44, 45, 437
incident logging 437
incident management 435
exercise 451
metrics 449
incident report 437, 453
incremental lifecycle model 3
incremental testing 289
independence of testing 529
individual skills 528
informal review 405
injection tools 479
insourced testing 74
inspection 333, 403
inspection leader (or moderator) 405
installability 390
integers
number of bits 125
signed 125
unsigned 125
integrated test system (example) 471
integration testing 4
big bang 289
call-graph based 290
incremental strategy 289
McCabe design predicate approach 292
interoperability 330
interoperability testing 330
introducing reviews 412
invalid 107
invalid equivalence classes 107
irrational number 130
ISO 9126 349, 355
ISO 9126 categories 338
ISO 9126 quality model 338
ISO 9126 quality standard 33, 34
ISO 9126 standard 323, 338
ISTQB Advanced exams 539
ISTQB fundamental test process 20
iterative call 295
iterative conditional cal 296

J
Jones, Capers 411
Jorgensen, Paul 289

K
keyword-based testing 510
keyword-driven architecture 504
keyword-driven test automation 489
keyword-driven testing 489
keywords 510
exercise 511

L
LCSAJ 215
LCSAJ (example) 217
LCSAJ testing 215
learning objectives 535
  levels 536, 538
level of risk 76
level test plan 71
levels of complexity 266
likelihood of a problem 76
Linear Code Sequence and Jump (LCSAJ) testing 215
load testing 358
logging test results 45
logging-type dynamic analysis tools 304
logic bombs 343
logical test cases 159, 164
loop coverage 191, 192
low coupling 383
low-level test case 27

M
maintainability 375
  analyzability 379
  changeability 381
  exercise 393
  stability 384
  subcharacteristics 379
  testability 385
maintainability defects 376
maintainability problems
  project issues 377
maintainability testing 375
management review 403, 408
mantissa 128
manual test case 493
manual test procedure 504
Marick, Brian 419
Marick's code review checklist 419
masking MC/DC 204
master test plan 71
maturity 349
MC/DC coverage 202
McCabe design predicate
  exercise 301
McCabe design predicate approach 292
McCabe, Thomas 220, 265
McKay, Judy 357
mean time between failures (MTBF) 351
mean time to repair (MTTR) 351
measure 11
measurement 11
measurement scale 11
measurements 11, 53
medium bang testing 292
memory leak 303
memory leak detection 305
metaphors 493
metric 11
metrics 11, 35, 53
metrics (NASA) 354
metrics for coverage 13
modified condition/decision coverage (MC/DC) 201
monitoring tools 485
motivation 529
multiple condition coverage 205
multiple condition testing 205
Myers, Glenford 242

N
N-1 switch coverage 155
  example 166
NASA 353
NASA Software Assurance Standard (NASA-STD-8739.8) 353
negative testing 349
neighborhood integration 291
  example 291
non-functional boundaries 123
non-functional test objectives 23
non-functional testing 24, 325
N-switch testing 166

O
objective vs. goal 13
Open Web Application Security Project (OWASP) 346
OpenLaszlo code review checklist 422
open-source test tools 470
operational acceptance testing 348
operational profiles 348, 349
operational readiness review 410
oracle problem 30
outsourced testing 74

P
pairwise graph 291
  example 290
path testing 214

peer review 403, 407
penetration testing 258
people skills 527
performance test tools 484
  example 485
performance testing 356, 357, 360, 459, 514
  analyze 365
  back-end data 517
  execute 365
  exercise 520
  input data 516
  model 364
  modeling the system 361
  performance acceptance criteria 363
  project success criteria 363
  runtime data 517
  start 519
  target data 516
  test environment 362, 364
  testing scripts 364
  tools 518
  variability of scenarios 364
  web application 362
performance testing tool 483
performance tools 514
piracy 340
pointer-induced failures 308
portability 386
  exercise 393
portability testing 386
PowerPoint 256
Pragmatic Risk Analysis and Management 81, 82
predictive reliability techniques 354
preparing for the exam 535
preventive test strategies 86
preventive testing 85
priority 443
process block 181
process defects 239
  arithmetic 239
  control or sequence 239
  initialization 239
  static logic 239
product risks 75, 76, 80
programming guidelines 270
programming languages 469
programming standards 270
project risk by-products 95
project risks 76, 80

Q

qualification review 410
quality attributes for technical testing 337
quality risk analysis document 88
quality risk management 11
quality risks 75
questionnaires 334

R

radix 128
reactive test strategies 42
real-world boundaries 132
reasonableness 494
recoverability 348, 350
recoverability testing 348
regulatory requirements 470
reliability 348, 349, 373
  exercise 372
reliability growth model 348
reliability testing 348, 359
replaceability 388
reporting of metrics and measure 13
reporting of results 53
requirements defect by-products 95
requirements review 409
requirements specifications 106
requirements-based testing 107
  exercise 175
residual risk 86
resource utilization testing 359
response-time test 358
review 333, 401
review principles 404
review types 406
  informal review 407
  inspections 407
  technical reviews 407
  walk-throughs 407
reviewer 401
reviews 399
  checklists 414
  introducing 412
  success factors 413
risk 76
risk analysis 79, 82, 84
  consensus 84
risk assessment 82
risk category 76
risk control 84

risk identification 79, 80
  example 87
risk level 78
risk management 79
risk mitigation 79, 84
risk type 76
risk-aware testing standards 90
risk-based testing 75, 76
  exercise 92, 96
  level of risk 77
  lifecycle 89
  risk items 77
risk-based testing strategy 22, 26, 77
root cause 437
root cause analysis 437
rounding errors 131

S

safety analysis techniques 10
safety integrity level (SIL) 91
safety-critical system 9, 10, 11
  characteristics and risks 10
sample exam questions 16, 67, 98, 309, 396, 432, 454, 523, 532
scalability testing 358
scheduling and test planning 73
scripting languages 469
Scrum 3
SDLC 24
security 338, 372
  exercise 372
security issues 339
security risks 338
security testing 258, 338
sequential lifecycle model 3
session-based test management 41, 97
set-use pair
  example 278
set-use pair notation 275
  atomic actions 276
severity 443
significand 128
simulator 488
simulators 488
single points of failure 10
single test procedure 44
SMLC 376
sociability 392
software attacks 252
  exercise 259
software characteristics 323
software development lifecycle (SDLC) 24
software lifecycle 2, 6
software maintenance lifecycle (SMLC) 376
software reliability 349
software reliability graph 351
Software Usability Measurement Inventory (SUMI) 334
specification, defect, and experience-based
  exercise 260
specification-based technique 105
specification-based tests 104
  model 105
  overview 104
spike testing 359
SQL injection 340
standards 457
state diagram 154
state table 162
state transition 154
state transition diagram 154, 155, 164
  example 156
state transition tables 162
state transition testing 154
state/transition coverage 158
state-based testing 155
  combining with other techniques 169
  coverage criterion 155
  event 158
  exercise 170
  state 158
  switches 156
statement coverage 184
  example 185
statement testing 184
statement-level coverage 184
static analysis 264, 480
  integration testing 288
static analysis tool 268, 270, 480
  Checkstyle 271
static analyzer 480
static testing 34
  example 35
  reviews 400
static tests
  reviews 102
  static analysis 102
stress testing 358
structural testing 227
structural-based testing 177

structure-based design technique (white-box test design technique) 177
structure-based techniques 262
structure-based testing
 exercise 228
substates 161
suitability 329
suitability testing 329
SUMI 334
summative evaluation 333
superstates 161
surveys 334
switch coverage 166
syntax testing 49
system defects 238
 hardware devices 239
 internal interface 238
 operating system 239
 resource management 239
 software architecture 239
system integration testing 9
system of systems 4
system test exit 58
system test exit review 60
 case study 59
system testing 9
systems 9
systems of systems 7
 characteristics and risks 7
 component integration 8
 fundamental test process 9
 information transfer 8
 integrating 8
 levels of integration 8
 system integration 8
 version management 8

T

taxonomy 240
taxonomy for tests 323
taxonomy of test techniques 101
technical review 403
technical security 338
technical test analyst
 scope 102
techniques for usability testing 333
test analysis 6, 21
test analysis and design
 metrics/measurements 35
 standards 31
test automation 38, 459
 benefits 462, 465
 costs 462
 risks 462, 463, 464
 scalability 499
 strategies 466
test basics 1–16
test basis 104, 105
test basis documents 29, 30
test case 22, 48
test case sequencing guidelines 97
test case values 109
test charter 247
test closure 6, 60
test closure activities 67
test condition 22
 level of detail 25
 structure 25
test control 6, 21, 47, 78
test data 39
test design 21
 27
test design process 28
test design specification 27
test design techniques 102
test environment 38, 39
test environment readiness 39
test estimation 72, 73
test execution 6, 24, 36, 37, 42, 53
 entry criteria 42
test execution schedule 36
test execution tools 475, 476
test implementation 28, 36, 53
test level 71
test log 41, 46
test logging 41
test management 69, 70, 98
 documentation 70
 evaluation of exit criteria 53
 reporting of results 53
test management tool 474
Test Maturity Model (TMM) 458
Test Maturity Model Integration (TMMi) 458
test monitoring 74
test objectives 22
test oracle 29, 105, 106, 489
 real world 30
test plan 21, 71
test plan documentation templates 71

test planning 21
test point analysis (TPA) 72
test policy 71
test procedure 37, 48, 493
test procedure readiness 37
test procedure specification 37
test process 6
test process control 73
test process improvement 457
Test Process Improvement (TPI) 458
test process models 20
test progress monitoring 73
test record tools 477
test schedule 72
test script 37, 49
test strategy 26, 71
test suite summary 54
test summary report 59, 60
test support tools 41
test team composition 527
test team dynamics 528
test techniques 101, 459
    taxonomy 101
test tool categories 473
test tool concepts 460
test tool portfolio 468
test tools 459
    deployment 470
testing
    goals 12
    organization 529
    software lifecycle 2
testing measurement 12
testing metrics 12
    definition 12
    reporting 12, 13
    tracking 12, 13
testware 41
three-value approach 134
time-critical systems 356
timely information 344
tip-over testing 359
TMM 458
TMMi 458
TPI 458
transactional situation 140
trigger a transition 158
troubleshooting tools 479
TTCN-3 469
Tufte, Edward 13
two's complements 125
two-pairs-of-eyes rule 414
typical test levels 4

U

unconditional call 293, 295
uncoupled condition 203
unique cause MC/DC 203
unit testing 184
usability 331
usability checklist (example) 334
usability testing 331
    exercise 335
    inspection 333
    questionnaires 334
    techniques 333
    validation 333
user acceptance testing 9

V

valid equivalence classes 107
validation 333
viruses 343
V-model 5, 6
vulnerabilities 344

W

walk-through 403
WAMMI 334
web testing tools 486
Website Analysis and MeasureMent
    Inventory (WAMMI) 334
white 49
white-box techniques 49
white-box testing 179
Whittaker, James 253
Whittaker's technique 257
whole number representations 126
Wideband Delphi 72
wild pointer 303
wild pointer detection 307
worms 343