/THEORY/IN/PRACTICE

Beautiful Testing

Leading Professionals Reveal How They Improve Software

Edited by Tim Riley
& Adam Goucher

O'REILLY®

Software Engineering

Beautiful Testing

"Any one of the insights or practical suggestions from these testing gurus would be worth the price of the book. The ideas are elegant and possibly challenging, yet are presented clearly and enthusiastically. This comprehensive, ambitious, engaging, and entertaining collection belongs on the bookshelf of every testing professional."

—Ken Doran, *QA Lead, Stanford University; Chair, Silicon Valley Software Quality Association*

Successful software depends as much on scrupulous testing as it does on solid architecture or elegant code. But testing is not a routine process; it's a constant exploration of methods and an evolution of good ideas.

Beautiful Testing offers 23 essays—from 27 leading testers and developers—that illustrate the qualities and techniques that make testing an art. Through personal anecdotes, you'll learn how each of these professionals developed beautiful ways of testing a wide range of products—valuable knowledge that you can apply to your own projects.

Here's a sample of what you'll find inside:

- **Microsoft's Alan Page knows a lot about large-scale test automation, and shares some of his secrets on how to make it beautiful**
- **Scott Barber explains why performance testing needs to be a collaborative process, rather than simply an exercise in measuring speed**
- **Karen N. Johnson describes how her professional experience intersected her personal life while testing medical software**
- **Rex Black reveals how satisfying stakeholders for 25 years is a beautiful thing**
- **Mathematician John D. Cook applies a classic definition of beauty, based on complexity and unity, to testing random number generators**

This book includes contributions from:

Adam Goucher
Linda Wilkinson
Rex Black
Martin Schröder
Clint Talbert
Scott Barber
Kamran Khan
Emily Chen and Brian Nitz
Remko Tronçon
Alan Page
Neal Norwitz, Michelle Levesque, and Jeffrey Yasskin
John D. Cook
Murali Nandigama
Karen N. Johnson
Chris McMahon
Jennitta Andrea
Lisa Crispin
Matthew Heusser
Andreas Zeller and David Schuler
Tomasz Kojm
Adam Christian
Tim Riley
Isaac Clerencia

All author royalties will be donated to the Nothing But Nets campaign to prevent malaria.

US $49.99 CAN $62.99
ISBN: 978-0-596-15981-8

Safari Books Online
Free online edition
for 45 days with purchase of this book. Details on last page.

O'REILLY® oreilly.com

Beautiful Testing

Edited by Tim Riley and Adam Goucher

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Beautiful Testing
Edited by Tim Riley and Adam Goucher

Copyright © 2010 O'Reilly Media, Inc.. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Mary E. Treseler
Production Editor: Sarah Schneider
Copyeditor: Genevieve d'Entremont
Proofreader: Sarah Schneider
Indexer: John Bickelhaupt
Cover Designer: Mark Paglietti
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
October 2009: First Edition.

O'Reilly and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Beautiful Testing*, the image of a beetle, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-15981-8

[V]

1255122093

All royalties from this book will be donated to the UN Foundation's Nothing But Nets campaign to save lives by preventing malaria, a disease that kills millions of children in Africa each year.

CONTENTS

PREFACE xiii
by Adam Goucher

Part One BEAUTIFUL TESTERS

1 WAS IT GOOD FOR YOU? 3
by Linda Wilkinson

2 BEAUTIFUL TESTING SATISFIES STAKEHOLDERS 15
by Rex Black

For Whom Do We Test? 16
What Satisfies? 18
What Beauty Is External? 20
What Beauty Is Internal? 23
Conclusions 25

3 BUILDING OPEN SOURCE QA COMMUNITIES 27
by Martin Schröder and Clint Talbert

Communication 27
Volunteers 28
Coordination 29
Events 32
Conclusions 35

4 COLLABORATION IS THE CORNERSTONE OF BEAUTIFUL PERFORMANCE TESTING 37
by Scott Barber

Setting the Stage 38
100%?!? Fail 38
The Memory Leak That Wasn't 45
Can't Handle the Load? Change the UI 46
It Can't Be the Network 48
Wrap-Up 51

Part Two BEAUTIFUL PROCESS

5 JUST PEACHY: MAKING OFFICE SOFTWARE MORE RELIABLE WITH FUZZ TESTING 55
by Kamran Khan

User Expectations 55
What Is Fuzzing? 57
Why Fuzz Test? 57

Fuzz Testing 60
Future Considerations 65

6 BUG MANAGEMENT AND TEST CASE EFFECTIVENESS 67
by Emily Chen and Brian Nitz

Bug Management 68
The First Step in Managing a Defect Is Defining It 70
Test Case Effectiveness 77
Case Study of the OpenSolaris Desktop Team 79
Conclusions 83
Acknowledgments 83
References 84

7 BEAUTIFUL XMPP TESTING 85
by Remko Tronçon

Introduction 85
XMPP 101 86
Testing XMPP Protocols 88
Unit Testing Simple Request-Response Protocols 89
Unit Testing Multistage Protocols 94
Testing Session Initialization 97
Automated Interoperability Testing 99
Diamond in the Rough: Testing XML Validity 101
Conclusions 101
References 102

8 BEAUTIFUL LARGE-SCALE TEST AUTOMATION 103
by Alan Page

Before We Start 104
What Is Large-Scale Test Automation? 104
The First Steps 106
Automated Tests and Test Case Management 107
The Automated Test Lab 111
Test Distribution 112
Failure Analysis 114
Reporting 114
Putting It All Together 116

9 BEAUTIFUL IS BETTER THAN UGLY 119
by Neal Norwitz, Michelle Levesque, and Jeffrey Yasskin

The Value of Stability 120
Ensuring Correctness 121
Conclusions 127

10 TESTING A RANDOM NUMBER GENERATOR 129
by John D. Cook

What Makes Random Number Generators Subtle to Test? 130
Uniform Random Number Generators 131

Nonuniform Random Number Generators 132
A Progression of Tests 134
Conclusions 141

11 CHANGE-CENTRIC TESTING 143
by Murali Nandigama

How to Set Up the Document-Driven, Change-Centric Testing Framework? 145
Change-Centric Testing for Complex Code Development Models 146
What Have We Learned So Far? 152
Conclusions 154

12 SOFTWARE IN USE 155
by Karen N. Johnson

A Connection to My Work 156
From the Inside 157
Adding Different Perspectives 159
Exploratory, Ad-Hoc, and Scripted Testing 161
Multiuser Testing 163
The Science Lab 165
Simulating Real Use 166
Testing in the Regulated World 168
At the End 169

13 SOFTWARE DEVELOPMENT IS A CREATIVE PROCESS 171
by Chris McMahon

Agile Development As Performance 172
Practice, Rehearse, Perform 173
Evaluating the Ineffable 174
Two Critical Tools 174
Software Testing Movements 176
The Beauty of Agile Testing 177
QA Is Not Evil 178
Beauty Is the Nature of This Work 179
References 179

14 TEST-DRIVEN DEVELOPMENT: DRIVING NEW STANDARDS OF BEAUTY 181
by Jennitta Andrea

Beauty As Proportion and Balance 181
Agile: A New Proportion and Balance 182
Test-Driven Development 182
Examples Versus Tests 184
Readable Examples 185
Permanent Requirement Artifacts 186
Testable Designs 187
Tool Support 189
Team Collaboration 192
Experience the Beauty of TDD 193
References 194

15 BEAUTIFUL TESTING AS THE CORNERSTONE OF BUSINESS SUCCESS 195
by Lisa Crispin

The Whole-Team Approach 197
Automating Tests 199
Driving Development with Tests 202
Delivering Value 206
A Success Story 208
Post Script 208

16 PEELING THE GLASS ONION AT SOCIALTEXT 209
by Matthew Heusser

It's Not Business…It's Personal 209
Tester Remains On-Stage; Enter Beauty, Stage Right 210
Come Walk with Me, The Best Is Yet to Be 213
Automated Testing Isn't 214
Into Socialtext 215
A Balanced Breakfast Approach 227
Regression and Process Improvement 231
The Last Pieces of the Puzzle 231
Acknowledgments 233

17 BEAUTIFUL TESTING IS EFFICIENT TESTING 235
by Adam Goucher

SLIME 235
Scripting 239
Discovering Developer Notes 240
Oracles and Test Data Generation 241
Mindmaps 242
Efficiency Achieved 244

Part Three BEAUTIFUL TOOLS

18 SEEDING BUGS TO FIND BUGS: BEAUTIFUL MUTATION TESTING 247
by Andreas Zeller and David Schuler

Assessing Test Suite Quality 247
Watching the Watchmen 249
An AspectJ Example 252
Equivalent Mutants 253
Focusing on Impact 254
The Javalanche Framework 255
Odds and Ends 255
Acknowledgments 256
References 256

19 REFERENCE TESTING AS BEAUTIFUL TESTING 257
by Clint Talbert

Reference Test Structure 258

	Reference Test Extensibility	261
	Building Community	266
20	CLAM ANTI-VIRUS: TESTING OPEN SOURCE WITH OPEN TOOLS *by Tomasz Kojm*	269
	The Clam Anti-Virus Project	270
	Testing Methods	270
	Summary	283
	Credits	283
21	WEB APPLICATION TESTING WITH WINDMILL *by Adam Christian*	285
	Introduction	285
	Overview	286
	Writing Tests	286
	The Project	292
	Comparison	293
	Conclusions	293
	References	294
22	TESTING ONE MILLION WEB PAGES *by Tim Riley*	295
	In the Beginning…	296
	The Tools Merge and Evolve	297
	The Nitty-Gritty	299
	Summary	301
	Acknowledgments	301
23	TESTING NETWORK SERVICES IN MULTIMACHINE SCENARIOS *by Isaac Clerencia*	303
	The Need for an Advanced Testing Tool in eBox	303
	Development of ANSTE to Improve the eBox QA Process	304
	How eBox Uses ANSTE	307
	How Other Projects Can Benefit from ANSTE	315
A	CONTRIBUTORS	317
	INDEX	323

Preface

I don't think Beautiful Testing could have been proposed, much less published, when I started my career a decade ago. Testing departments were unglamorous places, only slightly higher on the corporate hierarchy than front-line support, and filled with unhappy drones doing rote executions of canned tests.

There were glimmers of beauty out there, though.

Once you start seeing the glimmers, you can't help but seek out more of them. Follow the trail long enough and you will find yourself doing testing that is:

- Fun
- Challenging
- Engaging
- Experiential
- Thoughtful
- Valuable

Or, put another way, beautiful.

Testing as a recognized practice has, I think, become a lot more beautiful as well. This is partly due to the influence of ideas such as test-driven development (TDD), agile, and craftsmanship, but also the types of applications being developed now. As the products we develop and the

ways in which we develop them become more social and less robotic, there is a realization that testing them doesn't have to be robotic, or ugly.

Of course, beauty is in the eye of the beholder. So how did we choose content for *Beautiful Testing* if everyone has a different idea of beauty?

Early on we decided that we didn't want to create just another book of dry case studies. We wanted the chapters to provide a peek into the contributors' views of beauty and testing. *Beautiful Testing* is a collection of chapter-length essays by over 20 people: some testers, some developers, some who do both. Each contributor understands and approaches the idea of beautiful testing differently, as their ideas are evolving based on the inputs of their previous and current environments.

Each contributor also waived any royalties for their work. Instead, all profits from *Beautiful Testing* will be donated to the UN Foundation's Nothing But Nets campaign. For every $10 in donations, a mosquito net is purchased to protect people in Africa against the scourge of malaria. Helping to prevent the almost one million deaths attributed to the disease, the large majority of whom are children under 5, is in itself a Beautiful Act. Tim and I are both very grateful for the time and effort everyone put into their chapters in order to make this happen.

How This Book Is Organized

While waiting for chapters to trickle in, we were afraid we would end up with different versions of "this is how you test" or "keep the bar green." Much to our relief, we ended up with a diverse mixture. Manifestos, detailed case studies, touching experience reports, and war stories from the trenches—*Beautiful Testing* has a bit of each.

The chapters themselves almost seemed to organize themselves naturally into sections.

Part I, Beautiful Testers

Testing is an inherently human activity; someone needs to think of the test cases to be automated, and even those tests can't think, feel, or get frustrated. *Beautiful Testing* therefore starts with the human aspects of testing, whether it is the testers themselves or the interactions of testers with the wider world.

Chapter 1, *Was It Good for You?*
: Linda Wilkinson brings her unique perspective on the tester's psyche.

Chapter 2, *Beautiful Testing Satisfies Stakeholders*
: Rex Black has been satisfying stakeholders for 25 years. He explains how that is beautiful.

Chapter 3, *Building Open Source QA Communities*
: Open source projects live and die by their supporting communities. Clint Talbert and Martin Schröder share their experiences building a beautiful community of testers.

Chapter 4, *Collaboration Is the Cornerstone of Beautiful Performance Testing*

Think performance testing is all about measuring speed? Scott Barber explains why, above everything else, beautiful performance testing needs to be collaborative.

Part II, Beautiful Process

We then progress to the largest section, which is about the testing process. Chapters here give a peek at what the test group is doing and, more importantly, why.

Chapter 5, *Just Peachy: Making Office Software More Reliable with Fuzz Testing*

To Kamran Khan, beauty in office suites is in hiding the complexity. Fuzzing is a test technique that follows that same pattern.

Chapter 6, *Bug Management and Test Case Effectiveness*

Brian Nitz and Emily Chen believe that how you track your test cases and bugs can be beautiful. They use their experience with OpenSolaris to illustrate this.

Chapter 7, Beautiful XMPP Testing

Remko Tronçon is deeply involved in the XMPP community. In this chapter, he explains how the XMPP protocols are tested and describes their evolution from ugly to beautiful.

Chapter 8, *Beautiful Large-Scale Test Automation*

Working at Microsoft, Alan Page knows a thing or two about large-scale test automation. He shares some of his secrets to making it beautiful.

Chapter 9, *Beautiful Is Better Than Ugly*

Beauty has always been central to the development of Python. Neal Noritz, Michelle Levesque, and Jeffrey Yasskin point out that one aspect of beauty for a programming language is stability, and that achieving it requires some beautiful testing.

Chapter 10, Testing a Random Number Generator

John D. Cook is a mathematician and applies a classic definition of beauty, one based on complexity and unity, to testing random number generators.

Chapter 11, *Change-Centric Testing*

Testing code that has not changed is neither efficient nor beautiful, says Murali Nandigama; however, change-centric testing is.

Chapter 12, *Software in Use*

Karen N. Johnson shares how she tested a piece of medical software that has had a direct impact on her nonwork life.

Chapter 13, *Software Development Is a Creative Process*

Chris McMahon was a professional musician before coming to testing. It is not surprising, then, that he thinks beautiful testing has more to do with jazz bands than manufacturing organizations.

Chapter 14, *Test-Driven Development: Driving New Standards of Beauty*

Jennitta Andrea shows how TDD can act as a catalyst for beauty in software projects.

Chapter 15, *Beautiful Testing As the Cornerstone of Business Success*

Lisa Crispin discusses how a team's commitment to testing is beautiful, and how that can be a key driver of business success.

Chapter 16, *Peeling the Glass Onion at Socialtext*

Matthew Heusser has worked at a number of different companies in his career, but in this chapter we see why he thinks his current employer's process is not just good, but beautiful.

Chapter 17, *Beautiful Testing Is Efficient Testing*

Beautiful testing has minimal retesting effort, says Adam Goucher. He shares three techniques for how to reduce it.

Part III, Beautiful Tools

Beautiful Testing concludes with a final section on the tools that help testers do their jobs more effectively.

Chapter 18, *Seeding Bugs to Find Bugs: Beautiful Mutation Testing*

Trust is a facet of beauty. The implication is that if you can't trust your test suite, then your testing can't be beautiful. Andreas Zeller and David Schuler explain how you can seed artificial bugs into your product to gain trust in your testing.

Chapter 19, *Reference Testing As Beautiful Testing*

Clint Talbert shows how Mozilla is rethinking its automated regression suite as a tool for anticipatory and forward-looking testing rather than just regression.

Chapter 20, *Clam Anti-Virus: Testing Open Source with Open Tools*

Tomasz Kojm discusses how the ClamAV team chooses and uses different testing tools, and how the embodiment of the KISS principle is beautiful when it comes to testing.

Chapter 21, *Web Application Testing with Windmill*

Adam Christian gives readers an introduction to the Windmill project and explains how even though individual aspects of web automation are not beautiful, their combination is.

Chapter 22, *Testing One Million Web Pages*

Tim Riley sees beauty in the evolution and growth of a test tool that started as something simple and is now anything but.

Chapter 23, Testing Network Services in Multimachine Scenarios

When trying for 100% test automation, the involvement of multiple machines for a single scenario can add complexity and non-beauty. Isaac Clerencia showcases ANSTE and explains how it can increase beauty in this type of testing.

Beautiful Testers following a Beautiful Process, assisted by Beautiful Tools, makes for Beautiful Testing. Or at least we think so. We hope you do as well.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Beautiful Testing*, edited by Tim Riley and Adam Goucher. Copyright 2010 O'Reilly Media, Inc., 978-0-596-15981-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreilly.com/catalog/9780596159818

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://oreilly.com

Acknowledgments

We would like to thank the following people for helping make *Beautiful Testing* happen:

- Dr. Greg Wilson. If he had not written *Beautiful Code*, we would never have had the idea nor a publisher for *Beautiful Testing*.
- All the contributors who spent many hours writing, rewriting, and sometimes rewriting again their chapters, knowing that they will get nothing in return but the satisfaction of helping prevent the spread of malaria.
- Our technical reviewers: Kent Beck, Michael Feathers, Paul Carvalho, and Gary Pollice. Giving useful feedback is sometimes as hard as receiving it, but what we got from them certainly made this book more beautiful.
- And, of course, our wives and children, who put up with us doing "book stuff" over the last year.

—Adam Goucher

CHAPTER EIGHT

Beautiful Large-Scale Test Automation

Alan Page

Automated testing can be much more than simply writing and running tests that operate without human intervention. Alas, for many testers, automated testing consists only of the manual generation of test scripts or code that executes some specified test scenario or piece of product functionality. Consideration of the logistics of running the tests is too often an afterthought of the automation process.

Most testers are familiar with the claim that automated testing has the potential to save time. However, in many cases, test automation doesn't actually save the amount of time that testers and their management team have anticipated. In reality, many automation attempts fail because other than the actual test execution, none of the remainder of the process is automatic. For automation to be successful, especially on a large scale, the entire end-to-end process—from the moment the tester completes the authoring of the test until results are analyzed and available for viewing—must be automatic. Without this level of automation, the amount of time testers spend monitoring automation systems will quickly grow out of hand.

When I was young, my parents went to a fireworks stand and bought one of those big packages of fireworks. We eagerly waited until it got dark, and then our family went outside on our patio and began our show. We had quite a variety of explosions and showers of sparks and were generally entertained. It was a bit of a pain sometimes to try to find some of the fuses in the dark, and a few duds disappointed us, but for the most part, we were adequately amused.

A few years later, I attended my first *professional* fireworks show. Not only were the explosions and sparkles huge, but everything ran a million times smoother than our home fireworks show.

People oohed and aahed at the shapes and marveled over how well the show was synchronized to music. There was nothing wrong with our home fireworks show, but the professional show—due to the size, complexity, and how smoothly it ran—was truly beautiful.

A system where testers spend the majority of their time monitoring progress, examining errors (aka "duds"), and pushing tests and their artifacts from one stage to the next is far from beautiful. Beauty occurs when the entire system reaches a point of automation at which the tester can concentrate on what they do best: testing software.

Before We Start

For this topic, a few words on the approach to automation are in order. These days, there's a bit of a controversy in the testing world centered on when automated tests help test teams and when they stop teams dead in their tracks. Testers are worried about what to automate and how much to automate. Test managers are worried about showing ROI on the investment they've made on automation tools and in writing the tests, and in leveraging their automation investment to the fullest potential.

These testers and their managers often struggle as they try to determine how much of their testing effort they should automate. I have an answer to this dilemma that works for every test team, a metric they can use to make sure they are automating the right amount of tests. The metric is the same for every team, and unlike most metrics, it is always correct. *You should automate 100% of the tests that should be automated*. The metric itself is simple (albeit inherently un-actionable); the difficulty lies in determining exactly *which* tests to automate. Product architecture, stakeholders, schedule, and many other factors can help lead test teams to the correct automation decisions. I see many automation attempts fail because testers spend too much time automating (or attempting to automate) behaviors or scenarios where automation simply isn't worth the effort in the first place. Likewise, test automation efforts also can fail when testers fail to automate tasks that are clearly prime targets for automation. Deciding which tests to automate is extremely difficult and up to the test team and stakeholders to determine for their particular situation (and fortunately for the attention span of the reader, far beyond the scope of this chapter).

Also, note that for the sake of this essay, I'm referring to testing performed by an independent test team. Automation is the perfect and commonly acceptable solution for developer-written unit tests. The difficult decisions about what tests to automate typically occur in areas such as end-to-end scenarios or advanced GUI manipulation, or other areas where a human set of eyes and a brain are sometimes the most efficient oracle for determining success.

What Is Large-Scale Test Automation?

On a small software product, it may be possible for a single tester to write and run automated tests from his desk and report results via email or by manually updating a web page or

spreadsheet. Testers are also likely responsible for entering bugs for any issues discovered by the tests, and verifying that resolved bugs are fixed. For a system to be capable of handling thousands of tests or more distributed across hundreds of machines, the system must, at every step—from the moment test authoring is complete to the point where results are available—be completely automatic. Manual support for automated testing can take a significant amount of testing time away from the test team. Before we discuss implementation and architecture details of such a system, it will be helpful to discuss the workflow and the life of tests in an automation system.

The Basics of a Test Automation System

Even the bare necessities of a beautiful automation system enable efficiencies from end to end, but automation systems come in all shapes, sizes, and ranges of splendor. For all test teams that use automated tests, some basic steps are common. Figure 8-1 shows a basic automation test workflow.

FIGURE 8-1. Automated test life cycle workflow

Somewhere at or near the beginning of testing, a tester will write one or more automated tests. Testers may create *test suites* (a set of tests targeting a particular area or small feature) as well. They may write scripts, such as JavaScript, or create a compiled binary written in a language such as C# or C++. At some point in the process, the tests inevitably execute and a process (or person) tracks test results (e.g., “Pass” or “Fail”). Once the test results are available, the tester enters newly discovered errors into the bug tracking system, and may cross-reference fixed bugs to ensure that the relevant tests are now correctly passing.

The problem with the basics is that there are numerous scenarios where manual intervention may be necessary. Once the tester writes his tests, it may be his responsibility to copy the scripts or binaries (along with any needed support files) to a network share where other testers can access them. In other cases, the tester may be responsible for running the tests himself, analyzing the results, and reporting. Often, the tester is also responsible for entering bugs to correspond with failed tests, and verifying that bugs resolved as fixed truly are fixed.

A Beautiful System

The end-to-end automation process can certainly be more eye-catching. Let’s look at what a more sophisticated automation system and process might look like.

Once the tester completes the authoring of their code or script, they check it into a source control management system (SCM), where an automated build process builds the tests. Once the building process is complete, the system copies tests and support files to a common location where testers (as well as the tools that make up the automation system) can access them. Configuration of the systems used for running the tests happens automatically. The deployment to these machines and subsequent execution occurs automatically. A common database stores all test results for easy reporting and analysis. Finally, the system automatically records test failures in the bug database.

Even in a system such as this, there are many stages where nonautomated tasks can begin to monopolize a tester's time. I've personally seen many test teams create thousands of automated tests thinking that they will save massive amounts of testing time, but end up spending countless hours on the work surrounding those same automated tests. Indeed, on many teams, the testers spend so much time babysitting tests and analyzing test results that they have little time left for actual testing.

The First Steps

The first step for success is simple: write great tests. Many test teams set themselves up for failure by holding the quality bar for their test code to much lower standards than their product code. Teams that write poor tests in an effort to save time almost always end up putting a large effort into maintaining their tests and investigating failed tests.

Test code must be easily maintainable. Whether the tests are going to be around for 10 days or 10 years, on a large team, chances are that someone other than the author will have to read, debug, or modify the code at some point. On many teams I've observed, once authoring of an automated test and integration of that test into the system are complete, that test remains in the system forever (unless, of course, the team removes the component under test from the product). Whether or not the "once a test, always a test" concept is appropriate (as well as mitigation techniques for the issue) is a topic for a different essay.

Code reviews and static analysis of the test code are also important. Hardcoded paths to test servers or hardcoded usernames and passwords are frequent causes of fragile tests, as are hardcoded resource strings that fail the first time the test runs on non-English builds. Tools or scripts that analyze source code at or extremely close to compile time are essential for detecting many common programming errors such as these.

Finally, it's critical that test code is under source control and builds in a central location. Source control eases maintenance by enabling testers to investigate an entire history of changes to test code, overlaying infrastructure or "collateral" (noncode files used by the tests, such as media files or input data). Frequent builds or continuous integration are just as applicable to test code as they are to product code. Central builds allow the latest tests to run on every build of the production code, and ensure that tests are available from a central location. If testers build their own tests and are responsible for copying the tests to a server, mistakes are eventually bound

to occur. A central build also enables the embedding of consistent version information into every test. Version information greatly simplifies debugging when tests suddenly begin to fail or behave differently than expected.

Test Infrastructure Is Critical

All of the statements in the previous section apply directly to test infrastructure and tools. Chances are that if your team has a system similar to the one described in this essay, the development of at least some parts of the system occurs in-house. Moreover, these testing tools need to be inherently reliable and trusted. I have watched test teams struggle with poor tools and spend time writing workarounds for poor implementation only to have their workarounds break when someone finally gets around to updating the tools.

To that end, code reviews, unit testing, and acceptance testing are critical for all major pieces of a test infrastructure. Nobody wants to use a test tool that wastes testers' time, but nearly everyone wants to use tools that are intuitive, reliable, and just work.

Test Collateral

Another mistake that many teams make is mismanagement of their test collateral. By collateral, I mean data files, documents, add-ins, and any other bits of data used by the tests. It is vital to ensure that all of these files are under source control, but even more critical to guarantee that these files are available in a location that satisfies two key criteria. First, the files must be available to the automated testing system. For example, files on a share accessible to the test team members might not be available to the automation system if it has different access rights on the network. Second, the collateral files need to be stored in a nontemporary location. Sustained engineering or maintenance teams cringe when they inherit tests that attempt to copy collateral from a server or share that doesn't exist anymore.

Mitigation for this issue can include setting up a permanent server for test collateral or using a database to store all collateral items. Regardless of the solution, including a plan for handling test collateral is essential to ensure the robustness of the overall automation system.

Automated Tests and Test Case Management

As the number of test cases grows from hundreds to millions, test case management becomes a critical element of the overall system. In order to track what every test is doing and which tests are passing and failing, all tests need an associated unique identification number. At the very least, assigning test ID numbers enables tracking of test results across a number of testing configurations and scenarios, but it can also enable tracing of requirements or user stories to the tests that verify these requirements or stories.

Unique IDs assigned to individual test cases are a minimum requirement, but implementing a larger set of attributes associated with the test enables a much larger set of functionality.

Table 8-1 shows a small excerpt of test case IDs and samples of associated attributes in a test case management system.

TABLE 8-1. Example test case management identification attributes

Test case ID	Test binary or script	Command line	Configurations	Tags
10001	*perfTest.dll*	`/entry:perfTestOne`	All	Performance; Core
10002	*perfTest.dll*	`/entry:perfTestTwo`	Main	Performance
11001	*scenarioTest.exe*	`/Persona:Peggy`	Main	Scenarios
11002	*scenarioTest.exe*	`/Persona:Carl`	Mobile	Scenarios
12001	*Applications.js*		All	Applications
13002	*filesysBVT.dll*	`/entry:createFileTests /full`	Main	FileSystem; BVT
13023	*filesysBVT.dll*	`/entry:usbStorageTests`	USBStorage	FileSystem; BVT; USB

A brief explanation of the fields in Table 8-1 follow:

Test case ID
: This is a globally unique identifier assigned to each test. It's not necessary for the numbers to be sequential, but they must be unique. On large systems, a method for automatically assigning unique IDs is necessary. The sidebar "Breaking It All Down" on page 109 explains some common methods for generating IDs.

Test binary or script
: Automated tests generally "live" in a binary (EXE or DLL) or a script such as JavaScript, QuickTest Pro, PowerShell, or Python. The system (or harness) that is responsible for running the tests uses the binary or script name when executing the tests.

Command line
: The test system uses the command line along with the binary or script name in order to run the test. This enables customization, such as adding data references for data-driven tests, or enables the isolation of single tests within larger test files so they can run independently.

Configurations
: Early in planning for a software application or system, the team will define the configurations where the software will run. The configurations may be based on common installation scenarios, languages, or compatible platforms. For example, a team building a web portal and service may decide that valid test platforms are the last two versions of Internet Explorer and Firefox, and the latest versions of Chrome, Safari, and Opera. It's likely that not every test will need to run on every platform. For some test teams, the

excitement of test automation encourages them to run *every* test on *every* build. In the end, what these teams find is an abundance of redundant test and an automated test pass that takes far too long to complete to be helpful. Smartly tagging tests using pair-wise or other methods of matrix reduction for determining test configurations is a smarter way to deploy large-scale automation.

Tags

Tags are an optional method of describing additional metadata about the test. With tags in place, both the test run and the reporting of results are easily customizable. Running all "performance tests," for example, is simple if all performance tests are tagged as such. With multiple tags, it's also just as easy to view all results with similar customizations, or to drill down into special areas. Example queries could be "from all tests, view just performance tests" or "from all File System tests, show BVT tests, except for those run on Windows XP".

BREAKING IT ALL DOWN

For test automation authors, it often makes architectural sense to create several tests within one test binary. One of the most common approaches is to write test cases in a library (a *.dll* on Windows), and use a "test harness" to run the tests within the DLL.

For managed (e.g., C# or VB.NET) test binaries, attributes are a commonly used method:

```
[TestCaseAttribute( "Checkout Tests", ID=1)]
public TestResult BuyItems()
{
    TestResult.AddComment("executing Checkout Tests: Buy One Item");
    //implementation removed ...
    return TestResult.Pass;
}
[TestCaseAttribute( "Checkout Tests", ID=2)]
public TestResult BuyItems()
{
    TestResult.AddComment("executing Checkout Tests: Buy Zero Items");
    //code removed ...
    return TestResult.Pass;
}
```

The harness (or "runner") for managed code uses .NET Reflection to discover and run the tests in the library. Unit test frameworks such as NUnit and MSTest use this method to execute tests embedded in managed libraries.

Native (e.g., C or C++) binaries can use an alternate method of assigning tests to internal library functions. One somewhat common method is to embed a table within the library containing function addresses and other information about the test:

```
struct functionTable[] =
{
    { "Checkout tests: Buy one Item", 101, BuyItems, 1 },
    { "Checkout tests: Buy invalid number of items (0)", 102, BuyItems, 0 },
    { "Checkout tests: Buy negative number of items (-1)", 103, BuyItems, -1 },
    { "Checkout tests: Buying Scenario_One", 104, BuyingScenarioOne, 0 },
    // ...
};
```

The layout of the structure in this simple example contains four elements. They are:

Test description
: A plain-text description of the test (likely used in logging and reporting).

Test ID
: Unique identifier of the test.

Function pointer
: Address of the function that the harness should call for this test.

Optional parameters
: Value for the harness to pass to the function specified in the function pointer. This allows the test author to create one test function that, depending on a parameter, executes multiple test cases.

In the structure listed here, the description of the first test is "Checkout tests: Buy one Item," the Test ID is `104`, the function called by the harness is `BuyItems`, and the parameter passed to the function is the value `1`. There are, of course, multiple methods for implementing a test harness in native code, but this approach is simple and likely common.

A simple method that native code test harnesses can use to obtain information about tests in a library is for each test library to have an entry point that dumps the test IDs and entry points for all tests. Other than being useful for those wanting a quick method for determining which tests are in a binary, automation tools can manually parse this data in order to create test cases in a test case manager directly from the descriptions and unique IDs within a test binary:

```
void PrintFunctionTable()
{
    // display descriptions, entry points, test ID and parameters
    // for functionTable
}
```

There's one remaining hurdle to address. In a large system with hundreds of test libraries written by dozens of testers, it's virtually impossible to guarantee that test IDs will be unique across the test binaries or scripts. In the code excerpts shown earlier, for example, the managed function "Checkout Tests" and the native function "Checkout tests: Buy one Item" are both assigned the same ID.

One solution is to assign a bank of test IDs to each tester. As long as testers only use the IDs assigned to them, there's no problem. The usefulness of this solution fades quickly as testers come and go on a project, and as test libraries are shared between teams. A much better solution, of course, is an

automated solution. One way to accomplish this is to use a test harness or a similar tool and take steps such as the following in order to assign unique IDs to every test across an entire product:

1. Obtain the embedded test information from a library using the techniques described earlier (Reflection for managed code or a function table for native code).
2. Create new unique IDs for each test in the library.
3. Store a mapping of the library name, original ID, and unique ID in a database (see Table 8-2). The ID mappings enable uniqueness of IDs across a large system with minimal impact to the individual testers' authoring needs.

TABLE 8-2. Table with mapping of local IDs to global IDs

Library name	Local ID	Unique ID	Command line (samples)
buyTest.dll	1	1000001	`harness.exe buyTest.dll/id:1`
buyTest.dll	2	1000002	`harness.exe buyTest.dll/id:1`
shoppingTest.dll	1	1100001	`nHarness.exe shoppingTest.dll 1`
shoppingTest.dll	2	1100002	`nHarness.exe shoppingTest.dll 2`
shoppingTest.dll	3	1100003	`nHarness.exe shoppingTest.dll 3`

The example here implies that there are (at least) two test harnesses in use by the test team, and that they use different command lines to execute tests. Much more metadata about the tests (e.g., target module, test history) is often included in the test metadata.

The point of thoughtful, planned test case management is to save time at later stages of the automation process. If there's a failure in a test, giving a tester even 30 minutes to "dig in" to the failure and get more details is just too long. When there's a failure, you need to know exactly which test failed, what component it was testing, what caused the failure, and what kind of test was running. Otherwise, you may as well shrug your shoulders and say, "I don't know, boss, something went wrong." You can't expect to run millions of tests and keep everything in order without careful preparation and organization of the test data.

The Automated Test Lab

Tests need a place to run. Whether it's 10 machines in an office or hundreds of machines in an offsite data center, careful planning of the lab, including a test deployment strategy, is crucial. For efficiency reasons, it makes sense to run tests in parallel across a bank of test machines rather than sequentially on a single machine. If compatibility is a concern on the test team, or if a variety of environments are required for any other reason, the number of machines needed to complete automated testing in a reasonable amount of time grows quickly.

An efficiently configured test lab requires that there are enough machines available to allow the automated tests to complete in a reasonable amount of time while not having so many computers that machine utilization is too low. Test labs require computers, space, power, and cooling. To best offset the cost of running the test lab, the machines in an automated test lab should be as busy as possible. In addition to effectively using the machines in the test lab for running automated tests, another tactic is to use the machines in the lab to run extended versions of tests, stress tests, or specific customer scenarios between runs of the automated test pass.

Deploying the Test Bed

The test lab will likely contain both physical machines and hosted virtual machines. Deploying virtual machines is usually as simple as copying the appropriate virtual hard drives to the host system. For physical machines, installing a fresh version of an operating system plus updates and any necessary test applications is too time-consuming for practical test automation. If testing requires a fresh operating installation to be in place, a more efficient approach for the task of OS and application installation is via a disk-imaging tool that can quickly write an image of an operating system plus relevant applications to disk. Any time computers in the lab are being *prepared* for testing is time that they are *not* testing. Going through a 2-hour installation process in order to run a 10-minute test is something that few people would consider efficient. Minimizing test bed preparation time is a key part of increasing lab efficiency.

Other Considerations

Outside of the scope of testing technology but still necessary in order to deploy a successful test lab are plans for maintenance, power, and cooling. A well-planned and well-organized lab will save time if there are ever any problems with computer hardware or networking that need investigation. It's also certainly possible to house the test lab in a remote location. If this is the case, the lab should include remote control power strips or a 24-hours-a-day service-level agreement in the off chance that a machine hangs so hard during testing that normal means of rebooting are not possible.

Test Distribution

Once the computers in the test lab are prepared, the next piece (and possibly the most significant piece of the overall system) is to deploy and execute the tests. If you've invested in a test lab filled with hundreds of test machines, you want to make sure machine utilization is high—i.e., reducing as much as possible the amount of time those machines are idle, waiting for the instruction to run tests.

The flow chart in Figure 8-2 describes what decisions in test distribution may look like.

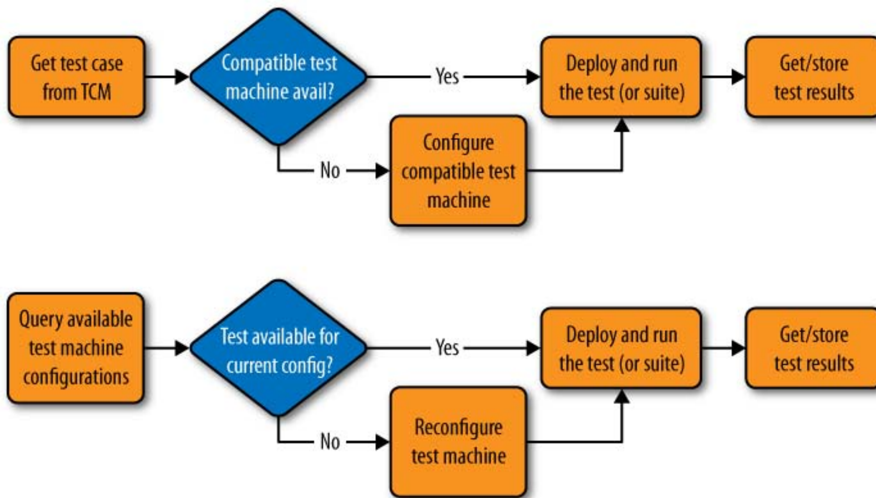

FIGURE 8-2. Test distribution flow chart

Depending on implementation, the first step is to take an inventory of either the test cases or the configurations available for testing. When starting with test cases, the next step is to begin deploying operating system and application configurations to test machines. When starting with available configurations, the secondary step is matching un-run test cases with the current set of machine configurations. In general, the former works a bit better when testing on a known smaller set of configurations, whereas the latter is slightly more convenient when dealing with a larger number of configurations.

One or more machines (sometimes called *test controllers*) are in charge of distributing the tests to the waiting machines in the test labs. These machines implement a connection system using sockets, message queues, or another multi-machine-aware synchronism mechanism.

Once preparation of the test computer is complete, the system copies the test to the target machine and the test executes. A test controller waits for the test to complete (or hang or crash), and then obtains the logfile from the test computer. The controller can parse the logfile for results directly, but more often it sends the logfile back to the test case manager or to another computer for parsing and analysis.

Yet another thing to consider in an automation system is a mechanism for knowing when the test has completed. Waiting a fixed time is risky, as tests will take a varying amount of time to complete on different systems. On the other hand, simply allotting a large amount of time for a test to complete usually means that test machines are idle for extended lengths of time. For tests that pass, the simple solution is for the test harness to signal the controller when the test

process exits. Additionally, associating a maximum time length for tests will trigger the controller to obtain results and crash log information from the test machine if the test does not complete (i.e., the harness does not signal completion) within the specified time.

Failure Analysis

If your tests are doing anything interesting, some of them are bound to fail. Some of the failures will be because of product bugs, and others will be due to errors in the tests. Some failures in related areas may be due to the same bug or, if you are running a single test on multiple configurations, it may fail in multiple (or all) configurations. If the same test (e.g., "verify widget control activates menu items") fails on both Windows XP and Windows Vista, failure analysis can analyze logfiles or other test collateral. If it determines that the same issue causes both, it can report only one failure.

If a team has a lot of tests and any significant number are failing, the test team can end up spending a lot of time investigating failed tests—so much time, in fact, that they have little time left for actually testing. The unfortunate alternative to this *analysis paralysis* is to simply gloss over the failure investigation and hope for the best (a practice that often ends with a critical bug finding its way into a customer's hands).

The solution to this predicament is to automate the analysis and investigation of test failures. The primary item that enables analysis to work effectively is to have consistent logging implemented across all tests. Matching algorithms implemented by the failure analysis system can look for similarities in the logs among failed tests and identify failures potentially triggered by the same root cause. The failure analysis system can also analyze call stacks or other debugging information automatically for any tests that cause a crash.

When tests fail, the system can either create a new bug report or update an existing report, depending on whether the failure is a new issue or already known (see Figure 8-3). An excellent return on investment for large-scale test automation requires integration in all stages of the automation, check-in, and bug tracking systems. A successful solution greatly reduces the need for manual intervention in analyzing test results and failures.

Reporting

From management's point of view, test results may be the most important artifact of the test team. Current and relevant test results are another key aspect of an automation system, and it is critical that test results are current, accurate, and easily available.

The role of an automation system in reporting is to ensure that the items in the previous paragraph are possible. Tracking and tagging of test results is imperative. For any new failures (automatic failure analysis should filter errors found in previous testing), information about

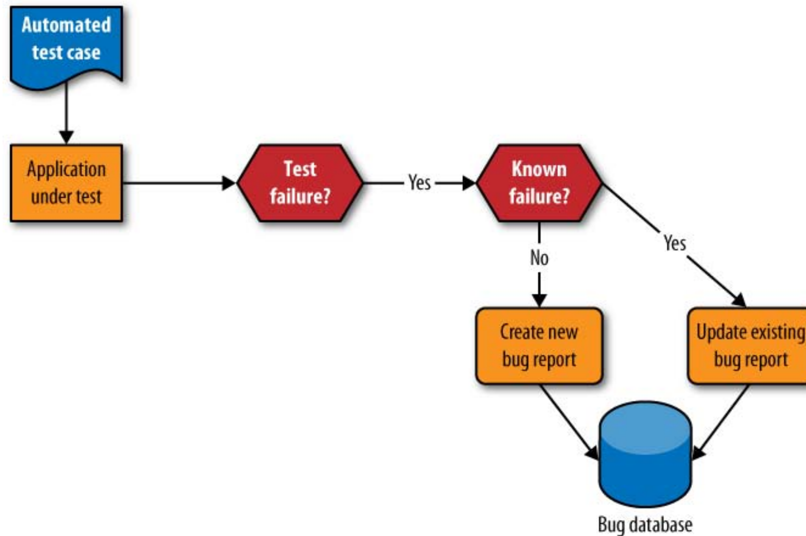

FIGURE 8-3. Automated failure analysis

the failure such as call stacks and logfiles need to be readily and quickly available so that diagnosis of the issue by the tester is as efficient as possible.

THE TRUTH ABOUT TEST RESULTS

Every team I know of that runs automated tests also tracks test results closely. Most of these teams have goals to reach or exceed, and a specified pass rate by the end of a milestone or by the time of release. For example, they may have a goal of passing 95% of their tests, or passing 100% of their "priority 1" test cases.

When teams tell me this, I always follow up by asking them what they do if they don't reach their goal. I ask, "What do you do if you're at a milestone exit, and your pass rate is only 94%?" They inevitably answer, "Oh, it depends on the failure. If the errors blocking us from reaching our goal don't meet the bar, we go ahead and move on." This, I suppose, is better than telling me that they just stop running that test (which is, unfortunately, something else I've heard in my career).

So, what is a reasonable goal for test pass rates? The answer is right in front of you. Rather than aim for a magic number, what you really want is to investigate 100% of the failures and ensure that none of those failures are serious enough to block the release. Of course, if you have a high pass rate, you will have fewer failures to investigate, but I see nothing special in reaching a magic pass rate number or letting the pursuit of that number drive your testing.

Putting It All Together

Once the whole package is together, test automation truly begins to become a benefit to testing. Testers can focus on one thing only: writing fantastic automated tests. These are tests that don't require constant maintenance, and tests that always generate meaningful actionable results. Figure 8-4 shows one description of the entire automation system described here.

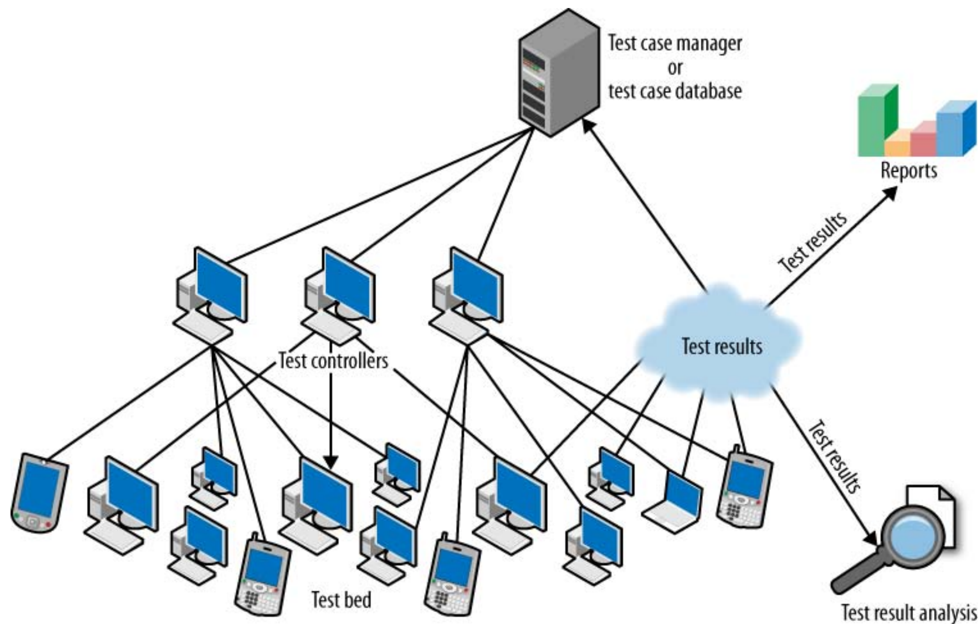

FIGURE 8-4. Large-scale test automation

Of course, if your automation system isn't at this level of maturity, you shouldn't expect it to get there instantly. The implementation of the details, as well as the relevant cultural changes, won't happen overnight. As with any large-scale change, changing and growing a little at a time is the best approach to turn an inefficient system into a beautiful one. Start by writing better tests; poor test automation is one of the biggest obstacles to automation success. Also make sure that test scripts and code are part of a source control system, and that tests go through some sort of continuous integration process to ensure some level of test quality and consistency.

Then, set up a test lab for running automated tests. Start working out a way to distribute tests to these machines and execute the tests. Then, begin gathering and aggregating simple high-level test results. Next, find a way to log test failures in the bug database. After that, investigate how you can automatically investigate failures and separate new failures from known failures. Finally, look for any other areas where testers are investing significant time and energy keeping the system running, and find a way to automate those tasks.

Before you know it, you will have a beautiful system. As it improves, testers on your team will have more and more time to do what they are best at, and something that they were (hopefully) hired to do in the first place: *test software*. Otherwise, despite thousands of automated tests and testers who are perpetually busy, you actually may not end up with a well-tested product. And that would be a sad ending to what should be a beautiful story.

APPENDIX

Contributors

Jennitta Andrea has been a multifaceted, hands-on practitioner (analyst, tester, developer, manager), and coach on over a dozen different types of agile projects since 2000. Naturally a keen observer of teams and processes, Jennitta has published many experience-based papers for conferences and software journals, and delivers practical, simulation-based tutorials and in-house training covering agile requirements, process adaptation, automated examples, and project retrospectives. Jennitta's ongoing work has culminated in international recognition as a thought leader in the area of agile requirements and automated examples. She is very active in the agile community, serving a third term on the Agile Alliance Board of Directors, director of the Agile Alliance Functional Test Tool Program to advance the state of the art of automated functional test tools, member of the Advisory Board of IEEE Software, and member of many conference committees. Jennitta founded The Andrea Group in 2007 where she remains actively engaged on agile projects as a hands-on practitioner and coach, and continues to bridge theory and practice in her writing and teaching.

Scott Barber is the chief technologist of PerfTestPlus, executive director of the Association for Software Testing, cofounder of the Workshop on Performance and Reliability, and coauthor of *Performance Testing Guidance for Web Applications* (Microsoft Press). He is widely recognized as a thought leader in software performance testing and is an international keynote speaker. A trainer of software testers, Mr. Barber is an AST-certified On-Line Lead Instructor who has authored over 100 educational articles on software testing. He is a member of ACM, IEEE, American Mensa, and the Context-Driven School of Software Testing, and is a signatory to the Manifesto for Agile Software Development. See *http://www.perftestplus.com/ScottBarber* for more information.

Rex Black, who has a quarter-century of software and systems engineering experience, is president of RBCS, a leader in software, hardware, and systems testing. For over 15 years, RBCS has delivered services in consulting, outsourcing, and training for software and hardware testing. Employing the industry's most experienced and recognized consultants, RBCS conducts product testing, builds and improves testing groups, and hires testing staff for hundreds of clients worldwide. Ranging from Fortune 20 companies to startups, RBCS clients save time and money through improved product development, decreased tech support calls, improved corporate reputation, and more. As the leader of RBCS, Rex is the most prolific author practicing in the field of software testing today. His popular first book, *Managing the Testing Process* (Wiley), has sold over 35,000 copies around the world, including Japanese, Chinese, and Indian releases, and is now in its third edition. His five other books on testing, *Advanced Software Testing: Volume I*, *Advanced Software Testing: Volume II* (Rocky Nook), *Critical Testing Processes* (Addison-Wesley Professional), *Foundations of Software Testing* (Cengage), and *Pragmatic Software Testing* (Wiley), have also sold tens of thousands of copies, including Hebrew, Indian, Chinese, Japanese, and Russian editions. He has written over 30 articles, presented hundreds of papers, workshops, and seminars, and given about 50 keynotes and other speeches at conferences and events around the world. Rex has also served as the president of the International Software Testing Qualifications Board and of the American Software Testing Qualifications Board.

Emily Chen is a software engineer working on OpenSolaris desktop. Now she is responsible for the quality of Mozilla products such as Firefox and Thunderbird on OpenSolaris. She is passionate about open source. She is a core contributor of the OpenSolaris community, and she worked on the Google Summer of Code program as a mentor in 2006 and 2007. She organized the first-ever GNOME.Asia Summit 2008 in Beijing and founded the Beijing GNOME Users Group. She graduated from the Beijing Institute of Technology with a master's degree in computer science. In her spare time, she likes snowboarding, hiking, and swimming.

Adam Christian is a JavaScript developer doing test automation and AJAX UI development. He is the cocreator of the Windmill Testing Framework, Mozmill, and various other open source projects. He grew up in the northwest as an avid hiker, skier, and sailer and attended Washington State University studying computer science and business. His personal blog is at *http://www.adamchristian.com*. He is currently employed by Slide, Inc.

Isaac Clerencia is a software developer at eBox Technologies. Since 2001 he has been involved in several free software projects, including Debian and Battle for Wesnoth. He, along with other partners, founded Warp Networks in 2004. Warp Networks is the open source–oriented software company from which eBox Technologies was later spun off. Other interests of his are artificial intelligence and natural language processing.

John D. Cook is a very applied mathematician. After receiving a Ph.D. in from the University of Texas, he taught mathematics at Vanderbilt University. He then left academia to work as a software developer and consultant. He currently works as a research statistician at M. D. Anderson Cancer Center. His career has been a blend of research, software development,

consulting, and management. His areas of application have ranged from the search for oil deposits to the search for a cure for cancer. He lives in Houston with his wife and four daughters. He writes a blog at *http://www.johndcook.com/blog*.

LISA CRISPIN is an agile testing coach and practitioner. She is the coauthor, with Janet Gregory, of *Agile Testing: A Practical Guide for Testers and Agile Teams* (Addison-Wesley). She works as the director of agile software development at Ultimate Software. Lisa specializes in showing testers and agile teams how testers can add value and how to guide development with business-facing tests. Her mission is to bring agile joy to the software testing world and testing joy to the agile development world. Lisa joined her first agile team in 2000, having enjoyed many years working as a programmer, analyst, tester, and QA director. From 2003 until 2009, she was a tester on a Scrum/XP team at ePlan Services, Inc. She frequently leads tutorials and workshops on agile testing at conferences in North America and Europe. Lisa regularly contributes articles about agile testing to publications such as *Better Software* magazine, *IEEE Software*, and *Methods and Tools*. Lisa also coauthored *Testing Extreme Programming* (Addison-Wesley) with Tip House. For more about Lisa's work, visit *http://www.lisacrispin.com*.

ADAM GOUCHER has been testing software professionally for over 10 years. In that time he has worked with startups, large multinationals, and those in between, in both traditional and agile testing environments. A believer in the communication of ideas big and small, he writes frequently at *http://adam.goucher.ca* and teaches testing skills at a Toronto-area technical college. In his off hours he can be found either playing or coaching box lacrosse—and then promptly applying lessons learned to testing. He is also an active member of the Association for Software Testing.

MATTHEW HEUSSER is a member of the technical staff ("QA lead") at Socialtext and has spent his adult life developing, testing, and managing software projects. In addition to Socialtext, Matthew is a contributing editor for *Software Test and Performance Magazine* and an adjunct instructor in the computer science department at Calvin College. He is the lead organizer of both the Great Lakes Software Excellence Conference and the peer workshop on Technical Debt. Matthew's blog, Creative Chaos, is consistently ranked in the top-100 blogs for developers and dev managers, and the top-10 for software test automation. Equally important, Matthew is a whole person with a lifetime of experience. As a cadet, and later officer, in the Civil Air Patrol, Matthew soloed in a Cessna 172 light aircraft before he had a driver's license. He currently resides in Allegan, Michigan with his family, and has even been known to coach soccer.

KAREN N. JOHNSON is an independent software test consultant based in Chicago, Illinois. She views software testing as an intellectual challenge and believes in context-driven testing. She teaches and consults on a variety of topics in software testing and frequently speaks at software testing conferences. She's been published in *Better Software* and *Software Test and Performance* magazines and on InformIT.com and StickyMinds.com. She is the cofounder of WREST, the Workshop on Regulated Software Testing. Karen is also a hosted software testing expert on Tech Target's website. For more information about Karen, visit *http://www.karennjohnson.com*.

Kamran Khan contributes to a number of open source office projects, including AbiWord (a word processor), Gnumeric (a spreadsheet program), libwpd and libwpg (WordPerfect libraries), and libgoffice and libgsf (general office libraries). He has been testing office software for more than five years, focusing particularly on bugs that affect reliability and stability.

Tomasz Kojm is the original author of Clam AntiVirus, an open source antivirus solution. ClamAV is freely available under the GNU General Public License, and as of 2009, has been installed on more than two million computer systems, primarily email gateways. Together with his team, Tomasz has been researching and deploying antivirus testing techniques since 2002 to make the software meet mission-critical requirements for reliability and availability.

Michelle Levesque is the tech lead of Ads UI at Google, where she works to make useful, beautiful ads on the search results page. She also writes and directs internal educational videos, teaches Python classes, leads the readability team, helps coordinate the massive postering of Google restroom stalls with weekly flyers that promote testing, and interviews potential chefs and masseuses.

Chris McMahon is a dedicated agile tester and a dedicated telecommuter. He has amassed a remarkable amount of professional experience in more than a decade of testing, from telecom networks to social networking, from COBOL to Ruby. A three-time college dropout and former professional musician, librarian, and waiter, Chris got his start as a software tester a little later than most, but his unique and varied background gives his work a sense of maturity that few others have. He lives in rural southwest Colorado, but contributes to a couple of magazines, several mailing lists, and is even a character in a book about software testing.

Murali Nandigama is a quality consultant and has more than 15 years of experience in various organizations, including TCS, Sun, Oracle, and Mozilla. Murali is a Certified Software Quality Analyst, Six Sigma lead, and senior member of IEEE. He has been awarded with multiple software patents in advanced software testing methodologies and has published in international journals and presented at many conferences. Murali holds a doctorate from the University of Hyderabad, India.

Brian Nitz has been a software engineer since 1988. He has spent time working on all aspects of the software life cycle, from design and development to QA and support. His accomplishments include development of a dataflow-based visual compiler, support of radiology workstations, QA, performance, and service productivity tools, and the successful deployment of over 7,000 Linux desktops at a large bank. He lives in Ireland with his wife and two kids where he enjoys travel, sailing, and photography.

Neal Norwitz is a software developer at Google and a Python committer. He has been involved with most aspects of testing within Google and Python, including leading the Testing Grouplet at Google and setting up and maintaining much of the Python testing infrastructure. He got deeply involved with testing when he learned how much his code sucked.

Alan Page began his career as a tester in 1993. He joined Microsoft in 1995, and is currently the director of test excellence, where he oversees the technical training program for testers and

various other activities focused on improving testers, testing, and test tools. Alan writes about testing on his blog, and is the lead author on *How We Test Software at Microsoft* (Microsoft Press). You can contact him at *alan.page@microsoft.com*.

Tim Riley is the director of quality assurance at Mozilla. He has tested software for 18 years, including everything from spacecraft simulators, ground control systems, high-security operating systems, language platforms, application servers, hosted services, and open source web applications. He has managed software testing teams in companies from startups to large corporations, consisting of 3 to 120 people, in six countries. He has a software patent for a testing execution framework that matches test suites to available test systems. He enjoys being a breeder caretaker for Canine Companions for Independence, as well as live and studio sound engineering.

Martin Schröder studied computer science at the University of Würzburg, Germany, from which he also received his master's degree in 2009. While studying, he started to volunteer in the community-driven Mozilla Calendar Project in 2006. Since mid-2007, he has been coordinating the QA volunteer team. His interests center on working in open source software projects involving development, quality assurance, and community building.

David Schuler is a research assistant at the software engineering chair at Saarland University, Germany. His research interests include mutation testing and dynamic program analysis, focusing on techniques that characterize program runs to detect equivalent mutants. For that purpose, he has developed the Javalanche mutation-testing framework, which allows efficient mutation testing and assessing the impact of mutations.

Clint Talbert has been working as a software engineer for over 10 years, bouncing between development and testing at established companies and startups. His accomplishments include working on a peer-to-peer database replication engine, designing a rational way for applications to get time zone data, and bringing people from all over the world to work on testing projects. These days, he leads the Mozilla Test Development team concentrating on QA for the Gecko platform, which is the substrate layer for Firefox and many other applications. He is also an aspiring fiction writer. When not testing or writing, he loves to rock climb and surf everywhere from Austin, Texas to Ocean Beach, California.

Remko Tronçon is a member of the XMPP Standards Foundation's council, coauthor of several XMPP protocol extensions, former lead developer of Psi, developer of the Swift Jabber/XMPP project, and a coauthor of the book *XMPP: The Definitive Guide* (O'Reilly). He holds a Ph.D. in engineering (computer science) from the Katholieke Universiteit Leuven. His blog can be found at *http://el-tramo.be*.

Linda Wilkinson is a QA manager with more than 25 years of software testing experience. She has worked in the nonprofit, banking, insurance, telecom, retail, state and federal government, travel, and aviation fields. Linda's blog is available at *http://practicalqa.com*, and she has been known to drop in at the forums on *http://softwaretestingclub.com* to talk to her Cohorts in Crime (i.e., other testing professionals).

Jeffrey Yasskin is a software developer at Google and a Python committer. He works on the Unladen Swallow project, which is trying to dramatically improve Python's performance by compiling hot functions to machine code and taking advantage of the last 30 years of virtual machine research. He got into testing when he noticed how much it reduced the knowledge needed to make safe changes.

Andreas Zeller is a professor of software engineering at Saarland University, Germany. His research centers on programmer productivity—in particular, on finding and fixing problems in code and development processes. He is best known for GNU DDD (Data Display Debugger), a visual debugger for Linux and Unix; for Delta Debugging, a technique that automatically isolates failure causes for computer programs; and for his work on mining the software repositories of companies such as Microsoft, IBM, and SAP. His recent work focuses on assessing and improving test suite quality, in particular mutation testing.

COLOPHON

The cover image is from Getty Images. The cover fonts are Akzidenz Grotesk and Orator. The text font is Adobe's Meridien; the heading font is ITC Bailey.