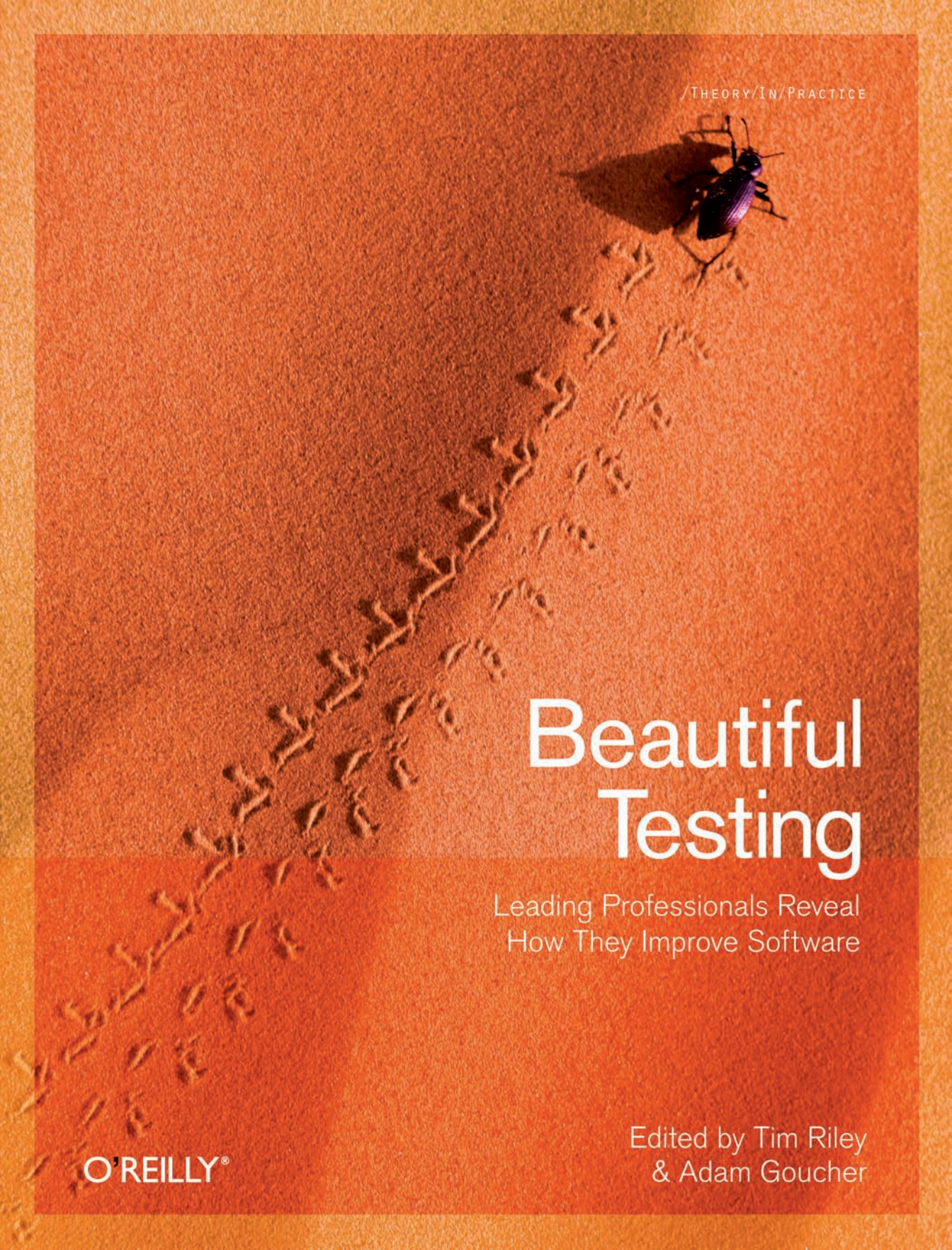
/THEORY/IN/PRACTICE



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

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

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.

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

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 FOUR

Collaboration Is the Cornerstone of Beautiful Performance Testing

Scott Barber

Performance testing is all too frequently the most frustrating, complicated, understaffed, time-crunched, misunderstood, combative, and thankless aspect of a software development project, but it doesn't have to be. I have experienced beautiful performance testing firsthand on several occasions. In fact, it seems like most career performance testers have at least one story about beautiful performance testing.

So, what are the attributes of beautiful performance testing? I think that beautiful performance testing is:

- Desired
- Deliberate
- Useful
- Technical
- Social
- Respectful
- Humble
- Efficient
- (Appropriately) challenging

- Value-driven
- Value-focused

But above all, I think that for performance testing to be beautiful, it must be *collaborative*.

In the stories that follow, I share with you the critical incidents that shaped my view of performance testing beauty. Coincidentally, this chapter is also the story of how one software development company's approach to performance testing became increasingly beautiful over the course of several development projects. While you read them, pay particular attention to two things. First, notice that none of these stories starts out, shall we say, beautifully. Second, notice that in each story collaboration was the key to progress, success, and/or conflict resolution.

Setting the Stage

All of the events that follow occurred over a 14-month period during the Dot-Com Era at a boutique custom software development company where I was the performance testing technical lead and practice manager. The events span several development projects, but the core project team was substantially the same throughout—and when I say project team, I am referring to not just those who wrote the code, but also to the executives, account managers, project management, business analysts, testers, system administrators, and technical support staff.

Although I have done my best to recount these events accurately and objectively, what follows is exclusively my perspective of the events that occurred. The events did occur in the sequence in which they appear in this chapter, and I have not taken any intentional liberties with them, other than to remove or replace offensive epithets with less offensive ones. In addition, I'd be remiss if I didn't mention that much of the identifying information related to individuals, clients, and contracts has been changed to protect the innocent, the guilty, the shy, and those I've lost touch with and couldn't get permission to use their real names.

100%?!? Fail

I'd just been informed that I was to start working on a new project to build a computer-based learning delivery and student progress tracking system (I'll call it eVersity) for a Fortune 50 company on the following Monday. The project was officially entering the development phase, which meant that the client had accepted our proof of concept and it was time to bring the rest of the team onto the project. I was at my desk finishing some documentation for my previous project when Harold, the test manager for the new project, walked up and, without preamble, handed me a single sheet of paper while asking, "Can you test this?"

Though I found the question insulting, I looked at the paper. I got as far as:

"System Performance Requirements:

- 100% of the web pages shall display in 5 seconds or less 100% of the time.
- The application shall…"

before writing "FAIL" on a sticky note, slapping the note on the paper, and handing it back to Harold over my shoulder and going back to work. Harold, making no attempt to conceal his anger at my note, asked, "What's *that* supposed to mean?" Spinning my chair around to face him, I replied, "I can test it if you want, but c'mon, it's the Internet! You never get 100% of anything!" Harold walked off in a huff.

Early the next week, Harold returned with another sheet of paper. Handing it to me, he simply asked "Better?" This time I managed to read all of the bullets.

"System Performance Requirements:

- 95% of the web pages shall display in 5 seconds or less 95% of the time.
- The application shall support 1,000 concurrent users.
- Courses shall download completely and correctly on the first try 98% of the time.
- Courses shall download in 60 seconds or less 95% of the time."

"Better? Yes. But not particularly useful, and entirely untestable. What is this for, anyway?" I responded. Clearly frustrated, but calm, Harold told me that he'd been asked to establish the performance requirements that were going to appear in our contract to the client. Now understanding the intent, I suggested that Harold schedule a conference room for a few hours for us to discuss his task further. He agreed.

As it turned out, it took more than one meeting for Harold to explain to me the client's expectations, the story behind his task, and for me to explain to Harold why we didn't want to be contractually obligated to performance metrics that were inherently ambiguous, what those ambiguities were, and what we could realistically measure that would be valuable. Finally, Harold and I took what were now several sheets of paper with the following bullets to Sandra, our project manager, to review:

"System Performance Testing Requirements:

- Performance testing will be conducted under a variety of loads and usage models, to be determined when system features and workflows are established.
- For internal builds, all performance measurements greater than the following will be reported to the lead developer:
  - Web pages that load in over 5 seconds, at any user volume, more than 5% of the time.
  - Web pages that load in over 8 seconds, at any user volume, more than 1% of the time.
  - Courses that do not download completely or correctly more than 2% of the time.

    - — Courses that take over 60 seconds to download, at any user volume, more than 5% of the time.
    - — The current maximum load the system can maintain for 1 hr with 95% of all web pages loading in 5 seconds or less and 95% of all the courses downloading completely and correctly in 60 seconds or less.
- External builds will be accompanied by a performance testing report including:
    - — Web pages that load in over 5 seconds, at any user volume, more than 5% of the time.
    - — Web pages that load in over 8 seconds, at any user volume, more than 1% of the time.
    - — Courses that do not download completely or correctly more than 2% of the time.
    - — Courses that take over 60 seconds to download, at any user volume, more than 5% of the time.
    - — The current maximum load the system can maintain for 1 hr with 95% of all web pages loading in 5 seconds or less and 95% of the courses downloading completely and correctly in 60 seconds or less.
- At the discretion of the project manager, other performance tests will be conducted that are deemed valuable to the project based on requests or recommendations by [client name deleted], the development team, or the performance test lead."

Much to our chagrin, Sandra replied that Harold and I should work together more often, and added our bullets verbatim into the client contract.

I fully admit that there was nothing beautiful about the *process* that led to Harold and I collaborating to turn the original System Performance Requirements into the ultimate System Performance Testing Requirements, but the *result* was. To be honest, when I found out that Harold had written the original requirements doc that I had "failed" in dramatic fashion, I fully expected to be removed from the project. But regardless of whether Harold tried to have me removed from the project, even he would have acknowledged that there was a certain beauty in the outcome that neither of us would have come up with on our own. Specifically:

- The shift from committing to achieving certain levels of performance to committing to report under what conditions the performance goals were *not* being achieved
- Calling out that it may be some time before enough information would be available to fully define the details of individual performance tests
- Leaving the door open for performance testing that supported the development process, but that didn't directly assess compliance with performance goals

Unfortunately, this was not to be the last un-beautiful interaction between Harold and me.

OK, but What's a Performance Test Case?

A few weeks later, Harold called to tell me he needed me to get all of the "performance test cases" into the eVersity test management system by the end of the following week. I said, "OK, but what's a performance test case?" As you might imagine, that wasn't the response he was expecting. The rest of the conversation was short but heated, and concluded with me agreeing to "do the best I could" by the end of that week so that he would have time to review my work.

As soon as I hung up the phone, I fired up the test management system to see if there were any other test cases for what we called nonfunctional requirements (aka quality factors, or parafunctional requirements), such as security or usability. Finding none, I started looking to the functional test cases for inspiration. What I found was exactly what I had feared: a one-to-one mapping between requirements and test cases, and almost all of the requirements were of the form "The system shall X," and almost all of the test cases were of the form "Verify that the system [does] X."

I stared at the screen long enough for my session to time out twice, trying to decide whether to call Harold back in protest or try to shoehorn *something* into that ridiculous model (for the record, I find that model every bit as ridiculous today as I did then). Ultimately, I decided to do what I was asked, for the simple reason that I didn't think I'd win the protest. The client had mandated this test management system, had paid a lot of money for the licenses, and had sent their staff to training on the system so they could oversee the project remotely. I simply couldn't imagine getting approval to move performance test tracking outside the system, so I created a new requirement type called "Performance" and entered the following items:

- Each web page shall load in 5 seconds or less, at least 95% of the time.
- Each course shall download correctly and completely in 60 seconds or less, at least 98% of the time.
- The system shall support 1,000 hourly users according to a usage model TBD while achieving speed requirements.

I then created the three parallel test cases for those items and crossed my fingers.

To say that Harold was not impressed when he reviewed my work at the end of the week would be a gross understatement. He must have come straight downstairs to the performance and security test lab where I spent most of my time the instant he saw my entry. As he stormed through the door, he demanded, "How can I justify billing four months of your time for three tests?"

Although I had been expecting him to protest, that was not the protest I'd anticipated. Looking at him quizzically, I responded by saying, "You can't. Where did you get the idea that I'd only be conducting three tests?" I'll let you imagine the yelling that went on for the next 15 minutes until I gave up protesting the inadequacy of the test management system, especially for performance testing, and asked Harold what it was that he had in mind. He answered that he wanted to see all of the tests I was going to conduct entered into the system.

I was literally laughing at loud as I opened the project repository from my previous, much smaller, project and invited him to come over and help me add up how many performance tests I'd conducted. The number turned out to be either 967 or 4,719, depending on whether you counted different user data as a different test. Considering that the five-person functional test team had created slightly fewer than 600 test cases for this project, as opposed to approximately 150 on the project I was referencing, even Harold acknowledged that his idea was flawed.

We stared at one another for what felt like a very long time before Harold dialed the phone.

"Sandra, do you have some time to join Scott and me in the lab? Thanks. Can you bring the client contracts and deliverable definitions? Great. Maybe Leah is available to join us as well? See you in a few."

For many hours, through a few arguments, around a little cursing, and over several pizzas, Harold, Sandra, Leah (a stellar test manager in her own right who was filling the testing technical lead role on this project), Chris (a developer specializing in security with whom I shared the lab and who had made the mistake of wandering in while we were meeting), and I became increasingly frustrated with the task at hand. At the onset, even I didn't realize how challenging it was going to be to figure out how and what to capture about performance testing in our tracking system.

We quickly agreed that what we wanted to include in the tracking system were performance tests representing valuable checkpoints, noteworthy performance achievements, or potential decision points. As soon as we decided that, I went to the whiteboard and started listing the tests we might include, thinking we could tune up this list and be done. I couldn't have been more wrong.

I hadn't even finished my list when the complications began. It turns out that what I was listing didn't comply with either the terms of the contract or with the deliverables definitions that the client had finally approved after much debate and many revisions. I don't remember all of the details and no longer have access to those documents, but I do remember how we finally balanced the commitments that had been made to the client, the capabilities of the mandated tracking system, and high-value performance testing.

We started with the first item on my list. Sandra evaluated the item against the contract. Harold evaluated it against the deliverables definitions. Leah assessed it in terms of its usefulness in making quality-related decisions. Chris assessed its informational value for the development team. Only after coming up with a list that was acceptable from each perspective did we worry about how to make it fit into the tracking system.

As it turned out, the performance requirements remained unchanged in the system. The performance test cases, however, were renamed "Performance Testing Checkpoints" and included the following (abbreviated here):

- Collect baseline system performance metrics and verify that each functional task included in the system usage model achieves performance requirements under a user load of 1 for each performance testing build in which the functional task has been implemented.

  — [Functional tasks listed, one per line]
- Collect system performance metrics and verify that each functional task included in the system usage model achieves performance requirements under a user load of 10 for each performance testing build in which the functional task has been implemented.

  — [Functional tasks listed, one per line]
- Collect system performance metrics and verify that the system usage model achieves performance requirements under the following loads to the degree that the usage model has been implemented in each performance testing build.

  — [Increasing loads from 100 users to 3,000 users, listed one per line]
- Collect system performance metrics and verify that the system usage model achieves performance requirements for the duration of a 9-hour, 1,000-user stress test on performance testing builds that the lead developer, performance tester, and project manager deem appropriate.

The beauty here was that what we created was clear, easy to build a strategy around, and mapped directly to information that the client eventually requested in the final report. An added bonus was that from that point forward in the project, whenever someone challenged our approach to performance testing, one or more of the folks who were involved in the creation of the checkpoints always came to my defense—frequently before I even found out about the challenge!

An interesting addendum to this story is that later that week, it became a company policy that I was to be consulted on any contracts or deliverable definitions that included performance testing before they were sent to the client for approval. I'm also fairly certain that this was the catalyst to Performance Testing becoming a practice area, separate from Functional Testing, and also what precipitated performance test leads reporting directly to the project manager instead of to the test manager on subsequent projects.

You Can't Performance Test Everything

One of the joys of being the performance testing technical lead for a company that has several development projects going on at once is that I was almost always involved in more than one project at a time. I mention this because the following story comes from a different project, but did occur chronologically between the previous story and the next one.

This project was to build a web-based financial planning application. Although common today, at the time this was quite innovative. The performance testing of the system was high priority for two reasons:

- We'd been hired for this project only after the client had fired the previous software development company because of the horrible performance of the system it had built.
- The client had already purchased a Super Bowl commercial time slot and started shooting the commercial to advertise the application.

Understandably, Ted, the client, had instructed me that he wanted "every possible navigation path and every possible combination of input data" included in our performance tests. I'd tried several methods to communicate that this was simply not an achievable task before *that* year's Super Bowl, but to no avail. Ted was becoming increasingly angry at what he saw as me refusing to do what he was paying for. After six weeks of trying to solve (or at least simplify) and document a massively complex combinatorics problem, I was becoming increasingly frustrated that I'd been unable to help the developers track down the performance issues that led Ted to hire us in the first place.

One afternoon, after Ted had rejected yet another proposed system usage model, I asked him to join me in the performance test lab to build a model together. I was surprised when he said he'd be right down.

I started the conversation by trying to explain to Ted that including links to websites maintained by other companies as part of our performance tests without their permission was not only of minimal value, but was tantamount to conducting denial-of-service attacks on those websites. Ted wasn't having any of it. At that moment, I realized I was standing, we were both yelling at one another, and my fists were clenched in frustration.

In an attempt to calm down, I walked to the whiteboard and started drawing a sort of sideways flowchart representing the most likely user activities on the website. To my surprise, Ted also picked up a marker and began enhancing the diagram. Before long, we were having a calm and professional discussion about what users were likely to do on the site during their first visit. Somewhere along the way, Chris had joined the conversation and was explaining to us how many of the activities we had modeled were redundant and thus interchangeable based on the underlying architecture of the system.

In less than an hour, we had created a system usage model that we all agreed represented the items most likely to be popular during the Super Bowl marketing campaign as well as the areas of the application that the developers had identified as having the highest risk of performing poorly. We'd also decided that until we were confident in the performance of those aspects of the system, testing and tuning other parts of the application was not a good use of our time.

Within a week of that meeting, we had an early version of the test we'd modeled up and running, and the developers and I were actively identifying and improving performance issues with the system.

Once again, the story had started anything but beautifully. This time the beauty began to blossom when Ted and I started working together to build a model at the whiteboard rather than me emailing models for him to approve. The beauty came into full bloom when Chris

brought a developer's perspective to the conversation. Collaborating in real time enabled us to not only better understand one another's concerns, but also to discuss the ROI of various aspects of the usage model comparatively as opposed to individually, which is what we'd been doing for weeks.

This story also has an interesting addendum. As it happened, the whiteboard sketch that Ted, Chris, and I created that day was the inspiration behind the User Community Modeling Language (UCML™) that has subsequently been adopted as the method of choice for modeling and documenting system usage for a large number of performance testers worldwide. For more about UCML, visit *http://www.perftestplus.com/articles/ucml.pdf*.

The Memory Leak That Wasn't

It was almost two months after the "OK, but what's a performance test?" episode before we were ready to start ramping up load with a reasonably complete system usage model on the eVersity project. The single-user and 10-user tests on this particular build had achieved better than required performance, so I prepared and ran a 100-user test. Since it was the first run of multiple usage scenarios at the same time, I made a point to observe the test and check the few server statistics that I had access to while the test was running.

The test ran for about an hour, and everything seemed fine until I looked at the scatter chart, which showed that all the pages that accessed the application server started slowing down about 10 minutes into the test and kept getting slower until the test ended. I surfed the site manually and it was fine. I checked the logfiles from the load generation tool to verify that I wasn't seeing the effect of some kind of scripting error. Confident that it wasn't something on my end, I ran the test again, only this time I used the site manually while the test was running. After about 15 minutes, I noticed those pages getting slow. I picked up the phone and called Sam, the architect for the project, to tell him that he had a memory leak on the application server.

Sam asked if I was running the test right then. I told him I was. I heard him clicking on his keyboard. He asked if the test was still running. I told him it was. He said, "Nope, no memory leak. It's your tool," and hung up.

I was furious. For the next two days I ran and re-ran the test. I scoured logfiles. I created tables and graphs. I brought them to project meetings. I entered defect reports. I sent Sandra the URL to the performance test environment and asked her to use the application while I ran tests. Everyone seemed to agree that it was acting like a memory leak. By the end of day two, even Sam agreed that it looked like a memory leak, but followed that up by saying, "…but it isn't."

Late on the third day after I'd first reported the issue, Sam called me and asked me to try the test again and hung up. I launched the test. About 20 minutes later, Sam called back to ask how the test looked. It looked great. I asked how he fixed it. He simply said, "Installed the permanent license key. The temp had a limit of three concurrent connections."

Sam didn't talk to me for the next couple of weeks. Since he wasn't very talkative in the first place, I wasn't certain, but I thought I'd offended him. Then a surprising thing happened. Sam called me and asked me to point "that test from the other week" at the development environment and to bring the results upstairs when it was done.

When I got upstairs with the results, Sam said to me, "Impressive work the other week. It took me over 20 hours to track down the license key thing. The tests we ran looked like a memory leak, too…except that the memory counters were showing tons of free memory. Anyway, from now on, why don't we look at weird results together?"

From that time on, Sam demanded that management assigned me to all of his projects. He'd frequently ask me to design and run tests that I didn't completely understand, but that would result in massive performance improvements within a day or two. I'd often call him and say, "I've got some odd-looking results here, would you like to have a look?" Sometimes he'd tell me why I was getting weird results, sometimes he'd want to take a look, and other times he'd ask me to run the test again in an hour, but he never again dismissed my results as a tool problem. And I never again announced the cause of a performance issue before confirming my suspicions with Sam.

Of course, the beauty here is that Sam came to see me as a valuable resource to help him architect better-performing applications with less trial and error on his part. In retrospect, I only wish Sam had been more talkative.

Can't Handle the Load? Change the UI

Very shortly after the "memory leak that wasn't" incident, it was decision time for the financial planning application. The application was functioning, but we simply did not believe that we could improve the performance enough to handle the Super Bowl commercial–inspired peak. I hadn't been included in the discussion of options until at least two weeks after both the client and the development team had become very concerned. I'm not exactly sure why I was invited to the "what are we going to do" meeting with the client that day, but it turned out that whoever invited me, intentionally or accidentally, was probably glad they did.

The short version of the problem was that the application gave all indications of being completely capable of handling the return user load, but that, if the projections were close to being correct, there was no way the architecture could handle the peak new user load generated by the Super Bowl advertising campaign. What I didn't know until I got to the meeting on that day was that we'd reached the point of no return in terms of hardware and infrastructure. The application was going to run in the existing environment. The question now was, what to do about the usage peak generated by the marketing campaign?

For about 30 minutes, I listened to one expensive and/or improbable idea after another get presented and rejected. The most likely option seemed to be to lease four identical sets of hardware and find a data center to host them, which was estimated to cost enough that every time it came up, Ted shook his head and mumbled something about not wanting to lose his job.

Finally, I spoke up. I pointed out that there were really only two things that we couldn't handle large numbers of users doing all at once. One was "Generate your personalized retirement savings plan," and the other was "Generate your personalized college savings plan." I also pointed out that for someone to get to the point where those plans were relatively accurate, they'd have to enter a lot of information that they probably didn't have at their fingertips. I then speculated that if we redesigned the UI so that those options weren't available until users had at least clicked through to the end of the questionnaire (as opposed to making it available on every page of the questionnaire, virtually encouraging folks to click the button after each piece of information they entered so they could watch the charts and graphs change), that might reduce the number of plan generations enough to get through the marketing campaign. I further commented that we could put the plan generation links back on each page after the campaign was over.

The looks of shock and the duration of stunned silence lasted long enough that I actually became uncomfortable. Eventually, a woman I didn't know started scribbling on a sheet of paper, then pulled out a calculator to do some calculations, then scribbled some more before very quietly saying, "It might just work." Everyone turned their stunned stares to her when Ted asked her what she'd said. She repeated it, but added, "Well, not *exactly* what he said, but what if we…."

To be honest, I partly don't remember what she said, and I partly never understood it, because she seemed to be speaking in some secret financial planning language to Ted. Regardless of the details, as soon as she was done explaining, Ted said, "Do it!" and everyone started smiling and praising me for solving the problem.

A few weeks later, I got a new build with a modified UI and usage model to test. It took a couple of iterations of minor UI modifications and tuning, but we achieved our target load. I found out much later that the marketing campaign was a success and that the system held up without a glitch. In fact, the campaign went *so* well that a big bank bought the company, Ted got a promotion, and of course, the big bank had their developers rebuild the entire application to run on their preferred vendor's hardware and software.

The obvious beauty here is that the project was successful and that the solution we came up with was not prohibitively complicated or expensive. The less obvious beauty lies in the often-overlooked value of involving people with several different perspectives in problem solving work groups.

It Can't Be the Network

As it turned out, the eVersity project was canceled before the application made it into production (and by canceled, I mean that client just called one day to tell us that their entire division had been eliminated), so we never got a chance to see how accurate our performance testing had been. On the bright side, it meant that the team was available for the client-server to Web call-center conversion project that showed up a couple of weeks later.

The first several months of the project were uneventful from a performance testing perspective. Sam and the rest of the developers kept me in the loop from the beginning. Jim, the client VP who commissioned the project, used to be a mainframe developer who specialized in performance, so we didn't have any trouble with the contract or deliverables definitions related to performance, and the historical system usage was already documented for us. Sure, we had the typical environment, test data, and scripting challenges, but we all worked through those together as they came up.

Then I ran across the strangest performance issue I've seen to this day. On the web pages that were requesting information from the database, I was seeing a response time pattern that I referred to as "random 4s." It took some work and some help from the developers, but we figured out that half of the time these pages were requested, they returned in about .25 seconds. Half of the rest of the time, they'd return in about 4.25 seconds. Half of the rest of the time, in 8.25 seconds. And so on.

Working together, we systematically figured out all the things that weren't causing the random 4s. In fact, we systematically eliminated every part of the system we had access to, which accounted for everything except the network infrastructure. Feeling good about how well things were going, I thought it was a joke when I was told that I was not allowed to talk to anyone in the IT department, but it wasn't. It seems that some previous development teams had blamed everything on the IT department and wasted a ton of their time, so they'd created a policy to ensure that didn't happen again.

The only way to interact with the IT department was for us to send a memorandum with our request signed by Jim, including detailed instructions, to the VP of the IT department through interdepartmental mail. I drafted a memo. Jim signed it and sent it. Two days later, Jim got it back with the word "No" written on it. Jim suggested that we send another memo that described the testing we'd done that was pointing us in the direction of the network. That memo came back with a note that said, "Checked. It's not us."

This went on for over a month. The more testing we did, the more convinced we were that this was the result of something outside of our control, and the only part of this application that was outside our control was the network. Eventually, Jim managed to arrange for a one-hour working conference call with the IT department, ostensibly to "get us off their back." We set everything up so that all we had to do was literally click a button when the IT folks on the

call were ready. Out entire team was dialed in on the call, just to make sure we could answer any question they may have had.

The IT folks dialed in precisely at the top of the hour and asked for identification numbers of the machines generating the load and the servers related to our application from the stickers their department put on the computers when they were installed. A few minutes later they told us to go ahead. We clicked the button. About five minutes of silence went by before we heard muffled speaking on the line. One of the IT staff asked us to halt the test. He said they were going to mute the line, but asked us to leave the line open. Another 20 minutes or so went by before they came back and asked us to restart the test and let them know if the problem was gone.

It took less than 10 minutes to confirm the problem was gone. During those 10 minutes, someone (I don't remember who) asked the IT staff, who had never so much as told us their names, what they had found. All they would say is that it looked like a router had been physically damaged during a recent rack installation and that they had swapped out the router.

As far as we knew, this interaction didn't make it any easier for the next team to work with this particular IT staff. I just kept thinking how lucky I was to be working on a team where I had the full help and support of the team. During the six weeks between the time I detected this problem and the IT department replaced the damaged router, the developers wrote some utilities, stubbed out sections of the system, stayed late to monitor after-hours tests in real time, and spent a lot of time helping me document the testing we'd done to justify our request for the IT department's time. *That* interaction is what convinced me that performance testing could be beautiful.

It's Too Slow; We Hate It

With the random 4s issue resolved, it was time for the real testing to begin: user acceptance testing (UAT). On some projects, UAT is little more than a formality, but on this project (and all of the projects I've worked on since dealing with call-center support software), UAT was central to go-live decisions. To that point, Susan, a call-center shift manager and UAT lead for this project, had veto authority over any decision about what was released into production and when.

The feature aspects of UAT went as expected. There were some minor revisions to be made, but nothing unreasonable or overly difficult to implement. The feedback that had us all confused and concerned was that every single user acceptance tester mentioned—with greater or lesser vehemence—something about the application being "slow" or "taking too long." Obviously we were concerned, because there is nothing that makes a call-center representative's day worse than having to listen to frustrated customers' colorful epithets when told, "Thank you for your patience, our system is a little slow today." We were confused because the website was fast, especially over the corporate network, and each UAT team was comprised of 5 representatives taking 10 simulated calls each, or about 100 calls per hour.

Testing indicated that the application could handle up to nearly 1,000 calls per hour before slowing down noticeably.

We decided to strip all graphics and extras from the application to make it as fast as possible, and then have myself or one of the developers observe UAT so we could see for ourselves what was slow. It confused us even more that the application was even faster afterward, that not one of the people observing UAT had noticed a user waiting on the application even once, and that the feedback was still that the application was slow. Predictably, we were also getting feedback that the application was ugly.

Finally, I realized that all of our feedback was coming either verbally from Susan or from Susan's summary reports, and I asked if I could see the actual feedback forms. While the protocol was that only the UAT lead got to see the actual forms, I was permitted to review them jointly with Susan. We were at the third or fourth form when I got some insight. The comment on that form was "It takes too long to process calls this way." I asked if I could talk to the user who had made that comment, and Susan set up a time for us to meet.

The next afternoon, I met Juanita in the UAT lab. I asked her to do one of the simulations for me. I timed her as I watched. The simulation took her approximately 2.5 minutes, but it was immediately clear to me that she was uncomfortable with both the flow of the user interface and using the mouse. I asked her if she could perform the same simulation for me on the current system and she said she could. It took about 5 minutes for the current system to load and be ready to use after she logged in. Once it was ready, she turned to me and simply said, "Ready?"

Juanita typed furiously for a while, then turned and looked at me. After a few seconds, I said, "You're done?" She smirked and nodded, and I checked the time: 47 seconds. I thanked her and told her that I had what I needed.

I called back to the office and asked folks to meet me in the conference room in 30 minutes. Everyone was assembled when I arrived. It took me fewer than 10 minutes to explain that when the user acceptance testers said "slow," they didn't mean response time; they meant that the design of the application was slowing down their ability to do their jobs.

My time on the project was pretty much done by then, so I don't know what the redeveloped UI eventually looked like, but Sam told me that they had a lot more interaction with Susan and her user acceptance testers thereafter, and that they were thrilled with the application when it went live.

For a performance tester, there are few things as beautiful as call-center representatives who are happy with an application you have tested.

Wrap-Up

In this chapter, I've shared with you a series of formative episodes in my evolution as a performance tester. The fact that these are real stories from real projects that I worked on, and the fact that they were sequential over approximately 14 months, only makes them more powerful.

Several years ago, I learned a technique called critical incident analysis that can be used to identify common principles used on or applied to complex tasks, such as performance testing. I learned about this technique from Cem Kaner and Rebecca Fiedler during the third Workshop on Heuristic and Exploratory Teaching (WHET). We were trying to determine how effective this approach would be in identifying core skills or concepts that people use when testing software, which would then be valuable to build training around.

According to Wikipedia:

> A critical incident can be described as one that makes a significant contribution—either positively or negatively—to an activity or phenomenon. Critical incidents can be gathered in various ways, but typically respondents are asked to tell a story about an experience they have had.

These stories are my critical incidents related to the importance of collaboration in performance testing. In these stories, a wide variety of performance testing challenges were tackled and resolved through collaboration: collaboration with other testers, collaboration with project management, collaboration with clients, collaboration with the development team, collaboration with IT staff, and collaboration with end users. All of my performance testing experiences corroborate what these stories suggest, that collaboration is the cornerstone of beautiful performance testing.

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.