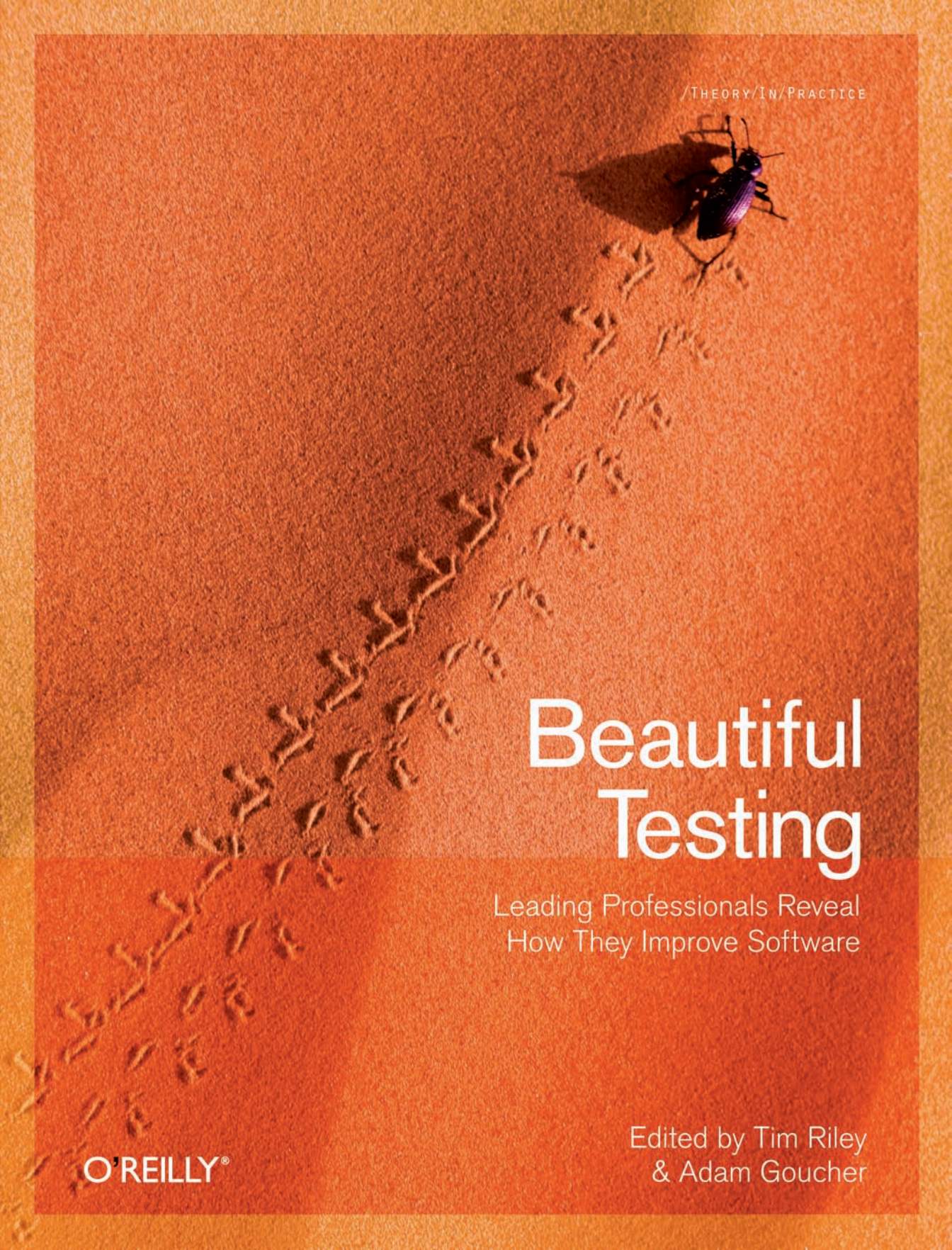
/THEORY/IN/PRACTICE



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 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 269
by Tomasz Kojm

The Clam Anti-Virus Project 270
Testing Methods 270
Summary 283
Credits 283

21 WEB APPLICATION TESTING WITH WINDMILL 285
by Adam Christian

Introduction 285
Overview 286
Writing Tests 286
The Project 292
Comparison 293
Conclusions 293
References 294

22 TESTING ONE MILLION WEB PAGES 295
by Tim Riley

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 303
by Isaac Clerencia

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 SEVEN

Beautiful XMPP Testing

Remko Tronçon

At my first job interview, one of the interviewers asked me if I knew what "unit testing" was and whether I had used it before. Although I had been developing an XMPP-based instant messaging (IM) client for years, I had to admit that I only vaguely knew what unit testing was, and that I hardly did any automated testing at all. I had a perfectly good reason, though: since XMPP clients are all about XML data, networks, and user interaction, they don't lend themselves well to any form of automated testing. A few months after the interview, the experience of working in an agile environment made me realize how weak that excuse was. It took only a couple of months more to discover how beautiful tests could be, *especially* in environments such as XMPP, where you would least expect them to be.

Introduction

The *eXtensible Messaging and Presence Protocol* (XMPP) is an open, XML-based networking protocol for real-time communication. Only a decade after starting out as an instant messaging solution under the name Jabber, XMPP is today being applied in a broad variety of applications, much beyond instant messaging. These applications include social networking, multimedia interaction (such as voice and video), micro-blogging, gaming, and much more.

In this chapter, I will try to share my enthusiasm about testing in the XMPP world, and more specifically in the Swift IM client. Swift is only one of the many XMPP implementations out there, and may not be the only one that applies the testing methods described here. However, it might be the client that takes the most pride in beautiful tests.

So, what do I consider to be "beautiful testing"? As you've probably discovered by now, opinions on the subject vary greatly. My point of view, being a software developer, is that beauty in tests is about the *code* behind the tests. Naturally, beautiful tests look good aesthetically, so layout plays a role. However, we all know that *true* beauty is actually found within. Beauty in tests is about simplicity; it's about being able to understand what a test (and the system being tested) does with a mere glance at the code, even with little or no prior knowledge about the class or component they test; it's about robustness, and not having to fix dozens of tests on every change; it's about having fun both reading *and* writing the tests.

As you might expect, there is a lot of code in the text that follows. And since I'm taking inspiration from Swift, which is written in C++, the examples in this chapter were written in C++ as well. Using a language like Ruby or Python probably would have made the tests look more attractive, but I stand by my point that true beauty in tests goes deeper than looks.

XMPP 101

Before diving into the details of XMPP implementation testing, let's first have a quick crash course about how XMPP works.

The XMPP network consists of a series of interconnected servers with clients connecting to them, as shown in Figure 7-1. The job of XMPP is to route small "packets" of XML between these entities on the network. For example, Alice, who is connected to the `wonderland.lit` server, may want to send a message to her sister, who is connected to the `realworld.lit` server. To do that, she puts her message into a small snippet of XML:

```
<message from="alice@wonderland.lit/RabbitHole"
         to="sister@realworld.lit">
  <body>Hi there</body>
</message>
```

She then delivers this message to her server, which forwards it to the `realworld.lit` server, which in turn delivers it to her sister's client.

Every entity on the XMPP network is addressed using a *Jabber ID* (JID). A JID has the form *`username@domain/resource`*, where *`domain`* is the domain name of the XMPP server and *`username`* identifies an account on that server. One user can be connected to the server with multiple instances of a client; the *`resource`* part of the JID gives a unique name to every connected instance. In some cases, the resource part can be left out, which means the server can route the message to whichever connected instance it deems best.

The small packets of XML that are routed through the network are called *stanzas*, and fall into three categories: *message* stanzas, *presence* stanzas, and *info/query* stanzas. Each type of stanza is routed differently by servers, and handled differently by clients.

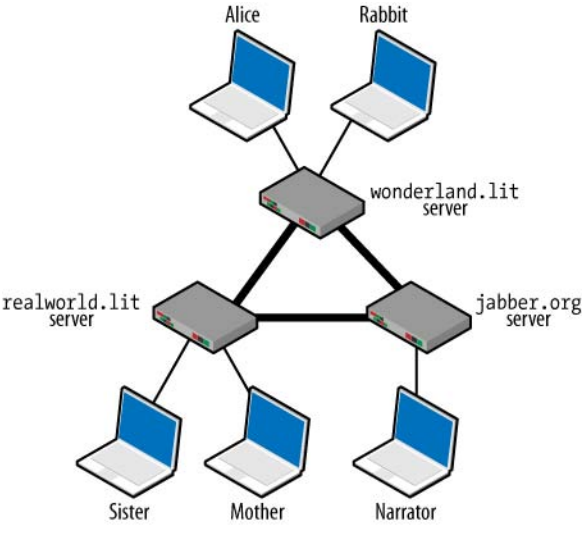


FIGURE 7-1. The decentralized architecture of the XMPP Network; clients connect to servers from different domains, which in turn connect to each other

Message stanzas
: Provide a basic mechanism to get information from one entity to another. As the name implies, message stanzas are typically used to send (text) messages to each other.

Presence stanzas
: "Broadcast" information from one entity to many entities on the network. For example, Alice may want to notify all of her friends that she is currently not available for communication, so she sends out the following presence stanza:

```
<presence from="alice@wonderland.lit/Home">
  <show>away</show>
  <status>Down the rabbit hole!</status>
</presence>
```

Her server then forwards this stanza to each of her contacts, informing them of Alice's unavailability.

Info/query (IQ) stanzas
: Provide a mechanism for request/response interactions between entities, typically used to query or change information on a given entity. For example, Alice could be interested in knowing what client version her sister is using. She therefore sends the following stanza to her sister:

```
<iq type="get" id="aad8a" ❶
    from="alice@wonderland.lit/RabbitHole" to="sister@realworld.lit/Home">
  <query xmlns="jabber:iq:version"/> ❷
</iq>
```

❶ The unique identifier of the stanza is used to match an incoming IQ result to the original IQ request.

❷ An empty child element (or *payload*) in a specific namespace indicates what type of information is requested (in this case, software version information).

Upon receiving this request, her sister's client immediately responds with the name and version of her client software:

```
<iq type="result" id="aad8a" ❶
    from="sister@realworld.lit/Home" to="alice@wonderland.lit/RabbitHole">
  <query xmlns="jabber:iq:version"> ❷
    <name>Swift</name>
    <version>1.0</version>
  </query>
</iq>
```

❶ The `id` attribute of the response matches the one from the request.

❷ The response's payload contains the result of the query.

Stanzas carry information in their *payloads*, which are added as child elements of the stanza. For example, a message can have a `body` payload containing the body text of the message. Different types of payloads are handled differently. By using XML namespaces for payloads, the XMPP protocol can easily be extended to support a virtually unlimited amount of information types, without having to worry about conflicting payload element names. For example, many of the early XMPP protocol extensions (including the software version protocol used in the examples) use the `query` payload. By using namespaces such as `jabber:iq:version`, entities know which type of protocol they are dealing with when they receive a `query` payload, and they know how to interpret the payload.

This section only scratched the surface of XMPP, just enough to get you through the rest of this chapter. If you want to learn more about how XMPP works and what you can do with it, have a look at [XMPP TDG] (see "References" on page 102), or visit *http://xmpp.org*.

Testing XMPP Protocols

One of the important aspects of an XMPP application, be it client or server, is the actual implementation of the XMPP *protocols*. Every XMPP implementation needs to at least implement the XMPP core protocols, as standardized by the IETF in [RFC 3920] and [RFC 3921]. These protocols provide the basic building blocks for XMPP, describing how an XMPP connection is set up, and what you can send over it. On top of the core protocols, the XMPP Standards Foundation created an ever-growing list of *XMPP Extension Protocols* (XEPs). These specifications describe how to extend the core protocol for very specific features, ranging from simple things such as requesting the software version of another client (standardized in [XEP-0092]), up to complex protocols for negotiating audio/video conference calls between clients, transferring files, and so on.

This text focuses on testing the functionality of XMPP protocol implementations, answering questions such as, "Does my client correctly respond to incoming requests?", "Does my client send the right requests at the right time?", "Can my client handle this specific response on this request?", and so on. We start out by looking at the most simple request-response protocols, after which we gradually move up the ladder to more complex protocols. While the complexity of the protocols increases, the level at which the tests are written becomes higher as well, moving from very specific unit tests up to full system tests. Although testing is mainly described from the perspective of a client developer, most of the approaches used here apply to server testing as well.

Unit Testing Simple Request-Response Protocols

Many of the XMPP protocols are simple: one side sends an IQ request, the other side receives the request, processes it, and responds with an IQ result. An example of such a simple request-response protocol is the software version protocol illustrated earlier. An implementation of this protocol consists of two parts:

- The *initiator* implementation sends a software version request and processes the corresponding response when it comes in.
- The *responder* listens for incoming software version requests and responds to them.

These implementations are typically implemented locally in one class for the initiator and one for the responder.* Example 7-1 shows how a `VersionResponder` class is instantiated in a client to respond to incoming software version requests. All this class does is listen for an incoming IQ query of the type `jabber:iq:version`, and respond with the values set through `setVersion`. The class uses the central `XMPPClient` class to send data to and receive data from the XMPP connection.

EXAMPLE 7-1. Using VersionResponder to listen and respond to software version requests

```
class MyClient {
  MyClient() {
    xmppClient = new XMPPClient("alice@wonderland.lit", "mypass");
    versionResponder = new VersionResponder(xmppClient);
    versionResponder->setVersion("Swift", "0.1");
    xmppClient->connect();
  }
  …
};
```

Since the implementation of request-response protocols is local to one class, *unit testing* is a good way to test the functionality of the protocol implementation. So, let's see how we can unit test the `VersionResponder` class.

* Some implementations are even known to put both responder and initiator implementations in one class. Don't try this at home, kids!

First, we need to make sure we can create an isolated instance of `Responder`. The only dependency this class has is the `XMPPClient`, a class that sets up and manages the XMPP connection. Setting up and managing a connection involves quite some work, and in turn brings in other dependencies, such as network interaction, authentication, encryption mechanisms, and so on. Luckily, all `VersionResponder` needs to be able to do is send and receive data from a data stream. It therefore only needs to depend on a `DataChannel` interface, which provides a method to send data and a signal to receive data, as shown in Example 7-2. This interface, implemented by `Client`, can be easily mocked in our unit test.

EXAMPLE 7-2. Abstracting out data interaction in a DataChannel interface; the XMPPClient class is a concrete implementation of this interface

```
class DataChannel {
  public:
    virtual void sendData(const string& data) = 0;
    boost::signal<void (const string&)> onDataReceived; ❶
};
```

❶ The signal `onDataReceived` has one `string` parameter (and no return value). When the signal is emitted, the `string` argument containing the data received will be passed to the connected slot method.

Now that we have all the ingredients for testing our `VersionResponder`, let's have a first attempt at writing a unit test. Example 7-3 shows how we can test the basic behavior of the responder, using a mock data channel to generate and catch incoming and outgoing data, respectively.

EXAMPLE 7-3. Testing VersionResponder using raw serialized XML data

```
void VersionResponderTest::testHandleIncomingRequest() {
  // Set up the test fixture
  MockDataChannel dataChannel;
  VersionResponder responder(&dataChannel);
  responder.setVersion("Swift", "1.0");

  // Fake incoming request data on the data channel
  dataChannel.onDataReceived(
    "<iq type='get' from='alice@wonderland.lit/RabbitHole' id='version-1'>"
      "<query xmlns='jabber:iq:version'/>"
    "</iq>");

  // Verify the outcome
  ASSERT_EQUAL(
    "<iq type='result' to='alice@wonderland.lit/RabbitHole' id='version-1'>"
      "<query xmlns='jabber:iq:version'>"
        "<name>Swift</name>"
        "<version>1.0</version>"
      "</query>"
    "</iq>",
    dataChannel.sentData);
}
```

On first sight, this unit test doesn't look too bad: it's relatively short, easy to understand, structured according to the rules of unit testing style, and isolates testing of the protocol from the low-level network aspects of XMPP. However, the beauty of this test is only skin-deep, as the test turns out to be pretty *fragile*. To see this, we need to look at how XMPP implementations generate the response to a request.

Whenever an XMPP client generates an XML stanza, it typically constructs the XML of the stanza by building up a structured document (e.g., using a *Document Object Model* [DOM] API), and then *serializes* this document into a textual XML representation, which is then sent over the network. In Example 7-3, our test records exactly the serialized XML stanza generated by the responder being tested, and then compares it to a piece of XML that it expects. The problem with this approach is that the same XML element can be serialized in different correct ways. For example, we could have switched the order of the `from` and `type` attributes of the `<iq/>` element and still have a logically equivalent stanza. This means that the smallest change to the way stanzas are serialized could break *all* tests.

One solution to avoid the fragility caused by XML serialization is to ensure that serialized stanzas are always in *Canonical XML* form (see [XML-C14n]). By normalizing away nonmeaningful properties such as attribute order and whitespace, this subset of XML ensures that two equivalent XML stanzas can be compared in a stable way, thus solving the fragility of our tests. Unfortunately, since XMPP implementations typically use off-the-shelf XML implementations, they often have no control over how XML is serialized, and as such cannot make use of this trick to compare stanzas.

The solution most XMPP implementations take to verify responses is to check the structured DOM form of the response instead of comparing the serialized form. As shown in Example 7-4, this means that our `VersionResponder` no longer uses an interface to send raw data, but instead depends on a more structured `XMLElementChannel` interface to send and receive stanzas as XML element data structures.

EXAMPLE 7-4. Testing VersionResponder using the structured XML representation; this test is no longer influenced by changes in the way the XML stanzas are serialized for transferring (e.g., different attribute order, extra whitespace)

```
void VersionResponderTest::testHandleIncomingRequest() {
  // Set up the test fixture
  MockXMLElementChannel xmlElementChannel;
  VersionResponder responder(&xmlElementChannel);
  responder.setVersion("Swift", "1.0");

  // Fake incoming request stanza on the stanza channel
  xmlElementChannel.onXMLElementReceived(XMLElement::fromString(
    "<iq type='get' from='alice@wonderland.lit/RabbitHole' id='version-1'>"
      "<query xmlns='jabber:iq:version'/>"
    "</iq>"));

  // Verify the outcome
  ASSERT_EQUAL(1, xmlElementChannel.sentXMLElements.size());
  XMLElement response = xmlElementChannel.sentXMLElements[0];
```

```
  ASSERT_EQUAL("iq", response.getTagName());
  ASSERT_EQUAL("result", response.getAttribute("type"));
  ASSERT_EQUAL("id", response.getAttribute("version-1"));
  ASSERT_EQUAL("alice@wonderland.lit/RabbitHole", response.getAttribute("to"));
  XMLElement queryElement = response.getElementByTagNameNS(
    "query", "jabber:iq:version");
  ASSERT(queryElement.isValid());
  XMLElement nameElement = queryElement.getElementByTagName("name");
  ASSERT(nameElement.isValid());
  ASSERT_EQUAL("Swift", nameElement.getText());
  XMLElement versionElement = queryElement.getElementByTagName("version");
  ASSERT(versionElement.isValid());
  ASSERT_EQUAL("1.0", versionElement.getText());
}
```

A downside of this test is that it is slightly less appealing than the one from Example 7-3. For this one test, the fact that it has become less compact and readable is only a small price to pay. However, suppose now that we also want to test the case where the user didn't provide a version to the version responder, in which case we want to send back "Unknown version" as a version string. This test would in fact look exactly like Example 7-4, except that the call to `setVersion` will pass an empty string instead of `"1.0"`, and the test would compare the version to `"Unknown version"`. Needless to say, this is a lot of duplicated code just to test a small difference in behavior, which will only get worse the more complex our protocol is (and hence the more tests it needs).

A first part of the duplication lies in checking whether the responder sends an `<iq/>` stanza of type `result`, whether it is addressed to the sender of the original stanza, and whether the identifier matches that of the request. This part can be easily factored out into a "generic" responder base class and tested separately.

A second problem with our test is the fact that we need to analyze the structure of the XML to extract the values we want to test. The real underlying problem here is the fact that our tests are testing two things at once: the *logic* of the protocol (i.e., *what* it should respond) and the *representation* of the responses (i.e., *how* the request and response is represented in XML).

To separate the logic from the representation in our test, we adapt our `VersionResponder` to work on a high-level `IQ` data structure, which in turn contains high-level `Payload` data structures representing the payloads they carry. Using these abstract data structures, we can now focus on testing the `VersionResponder`'s functionality, without worrying about how the `IQ` and `Payload` data structures are actually represented in XML. The resulting test can be seen in Example 7-5.

EXAMPLE 7-5. Testing the logic of VersionResponder; the actual (XML) representation of the stanzas sent and received by VersionResponder are no longer explicitly present in this test, making the test resistant against changes in representation

```
void VersionResponderTest::testHandleIncomingRequest() {
  // Set up the test fixture
  MockIQChannel iqChannel;
  VersionResponder responder(&iqChannel);
```

```
  responder.setVersion("Swift");

  // Fake incoming request stanza on the stanza channel
  iqChannel.onIQReceived(IQ(IQ::Get, new VersionPayload()));

  // Verify the outcome
  ASSERT_EQUAL(1, iqChannel.sentIQs.size());
  const VersionPayload* payload =
    iqChannel.sentIQs[0].getPayload<VersionPayload>();
  ASSERT(payload);
  ASSERT_EQUAL("Swift", payload->getName());
  ASSERT_EQUAL("Unknown version", payload->getVersion());
}
```

The conversion from the `VersionPayload` structure to XML can now be tested independently, as illustrated in Example 7-6. Although this test still isn't very attractive, the clutter coming from the representational part no longer impacts the tests for the more important behavioral part of the protocol.

EXAMPLE 7-6. Testing the conversion of VersionPayload to XML

```
void VersionPayloadSerializerTest::testSerialize() {
  // Set up the test fixture
  VersionPayloadSerializer serializer;
  VersionPayload payload;
  payload.setVersion("Swift", "1.0");

  // Serialize a payload
  XMLElement result = serializer.serialize(payload);

  // Verify the serialized element
  ASSERT_EQUAL("query", result.getTagName());
  ASSERT_EQUAL("jabber:iq:version", result.getNamespace());
  XMLElement* nameElement = queryElement->getElementsByTagName("name");
  ASSERT(nameElement);
  ASSERT_EQUAL("Swift", nameElement->getText());
  XMLElement* versionElement = queryElement->getElementsByTagName("version");
  ASSERT(versionElement);
  ASSERT_EQUAL("1.0", versionElement->getText());
}
```

In this section, we discussed how to test a simple IQ-based request/response protocol. In our first attempt, we tested the protocol at the lowest level possible, by analyzing the actual data sent over the wire. Subsequent versions tested the logic of the protocol at a higher, more structured level, up to the point where the logic of the responder was tested independently of the actual representation of the data sent over the network. Although it might seem overkill to separate the XML parsing and serializing from the actual data structure for a simple protocol like the one shown here, it makes testing the more complex (multistage) protocols from the next sections a lot cleaner.

Unit Testing Multistage Protocols

So far, the class of protocols we have considered was rather simple: one side sent out a request, the other side responded, and we were done. Although many of the XMPP protocols fall within this category, there are several others that consist of multiple iterations of these request-response cycles. These protocols start by doing a request, and then take subsequent steps based on the response of previous requests. Testing these types of protocols is the focus of this section.

Besides person-to-person conversations, XMPP also allows users to join chat "rooms" to communicate with multiple people at once. Whenever a user wants to join such a *multiuser chat* (MUC for short), an IM client needs to detect the MUC rooms that are available on a server and present this list to the server. Obtaining this list requires a chain of multiple *service discovery* (often called *disco* in the XMPP world) requests. For example, let's assume Alice wants to get a list of all the available rooms on the `wonderland.lit` server. She starts by requesting all the available services of her server, which is done by sending a `disco#items` request to the server:

```
<iq type="get" id="muc-1" to="wonderland.lit">
  <query xmlns="http://jabber.org/protocol/disco#items"/>
</iq>
```

The server then responds with the list of all its services:

```
<iq type="result" id="muc-1"
    from="wonderland.lit" to="alice@wonderland.lit/RabbitHole">
  <query xmlns="http://jabber.org/protocol/disco#items">
    <item jid="pubsub.wonderland.lit"/>
    <item jid="rooms.wonderland.lit"/>
  </query>
</iq>
```

Alice now needs to determine which one of these services provides chat rooms. She therefore sends a `disco#info` request to each service, asking them which protocols they support:

```
<iq type="get" id="muc-2" to="pubsub.wonderland.lit">
  <query xmlns="http://jabber.org/protocol/disco#info"/>
</iq>

<iq type="get" id="muc-3" to="rooms.wonderland.lit">
  <query xmlns="http://jabber.org/protocol/disco#info"/>
</iq>
```

The first service responds:

```
<iq type="result" id="muc-2"
    from="pubsub.wonderland.lit" to="alice@wonderland.lit/RabbitHole">
  <query xmlns="http://jabber.org/protocol/disco#info">
    <feature var="http://jabber.org/protocol/pubsub"/>
  </query>
</iq>
```

This service seems to support only the PubSub protocol (feature), which is not what Alice was looking for. The second service, however, responds with the following feature list:

```
<iq type="result" id="muc-3"
    from="rooms.wonderland.lit" to="alice@wonderland.lit/RabbitHole">
  <query xmlns="http://jabber.org/protocol/disco#info">
    <feature var="http://jabber.org/protocol/muc"/>
  </query>
</iq>
```

Bingo! Now that she found the MUC service, all she needs to do is ask for the list of rooms, which is done using another `disco#items` request:

```
<iq type="get" id="muc-4" to="rooms.wonderland.lit">
  <query xmlns="http://jabber.org/protocol/disco#items"/>
</iq>
```

This request results in the list of all the MUC rooms on the `rooms.wonderland.lit` server (in this case, a tea party and a room for discussing croquet):

```
<iq type="result" id="muc-4"
    from="rooms.wonderland.lit" to="alice@wonderland.lit/RabbitHole">
  <query xmlns="http://jabber.org/protocol/disco#items">
    <item jid="teaparty@rooms.wonderland.lit"/>
    <item jid="croquet@rooms.wonderland.lit"/>
  </query>
</iq>
```

As you can tell from this scenario, a lot of stanzas are going back and forth. Things become even more complex if you take into consideration that every step can result in an error response from the responding entity. Testing this protocol therefore involves multiple tests, for determining whether our client can handle every type of response from the server, both successful and unsuccessful. Luckily, because of the high level at which we test our protocols, creating a test for one scenario can be very compact and straightforward. For example, a test for the "happy," error-less scenario described earlier is shown in Example 7-7.

EXAMPLE 7-7. Testing RoomDiscoverer

```
void RoomDiscovererTest::testDiscoverRooms() {
  // Set up the responses
  itemsResponses["wonderland.lit"] = ❶
    DiscoItems("pubsub.wonderland.lit", "rooms.wonderland.lit");
  infoResponses["pubsub.wonderland.lit"] = ❷
    DiscoInfo("http://jabber.org/protocol/pubsub");
  itemsResponses["pubsub.wonderland.lit"] = ❸
    DiscoItems("blogs@pubsub.wonderland.lit", "croquet@pubsub.wonderland.lit");
  infoResponses["rooms.wonderland.lit"] = ❹
    DiscoInfo("http://jabber.org/protocol/muc");
  itemsResponses["rooms.wonderland.lit"] = ❺
    DiscoItems("teaparty@rooms.wonderland.lit", "croquet@rooms.wonderland.lit");

  // Set up room discoverer
  RoomDiscoverer discoverer(channel);
```

```
  // Execute room discovery
  discoverer.discoverRooms();

  // Test results
  ASSERT(discoverer.isFinished());
  StringList rooms = discoverer.getDiscoveredRooms();
  ASSERT_EQUAL(2, rooms.size());
  ASSERT(rooms.contains("teaparty@rooms.wonderland.lit"));
  ASSERT(rooms.contains("croquet@rooms.wonderland.lit"));
}
```

❶ Specify the response to a `disco#items` request for the top-level `wonderland.lit` domain. In this case, two items are returned: `pubsub` and `rooms`.

❷ Specify the response to a `disco#info` request for the `pubsub` service. In this case, respond with the namespace of the PubSub protocol.

❸ Specify the items belonging to the `pubsub` service. These are added to test whether `RoomDiscoverer` doesn't pick up items from non-MUC services.

❹ Respond that the `rooms` service supports MUC.

❺ Specify the list of items (i.e., rooms) of the MUC service.

The test specifies what responses should be sent for both `disco#info` and `disco#items` queries directed to specific JIDs. The `RoomDiscoverer` (which is the class that is responsible for discovering rooms) is then put in action, after which the test checks whether it indeed discovered both MUC rooms (and didn't accidentally include the PubSub service items). Not only is the test simple, but the auxiliary methods used by this test (including the fixture setup and tear down) can be kept very simple as well, as can be seen in Example 7-8.

EXAMPLE 7-8. Setting up the RoomDiscovererTest fixture

```
void RoomDiscovererTest::setUp() {
  channel = new MockIQChannel();
  channel->onSendIQ.connect(bind(&RoomDiscovererTest::respondToIQ, this, _1)); ❶
}

void RoomDiscovererTest::tearDown() {
  delete channel;
}

void RoomDiscovererTest::respondToIQ(const IQ& iq) {
  ASSERT(iq.getType() == IQ::Get);
  if (iq.getPayload<DiscoItems>()) {
    ItemsResponseMap::const_iterator response = itemsResponses.find(iq.getTo());
    ASSERT(response != itemsResponses.end());
    channel->onIQReceived(iq.createResponse(new DiscoItems(response->second)));
  }
  else if (iq.getPayload<DiscoInfo>()) {
    InfoResponseMap::const_iterator response = infoResponses.find(iq.getTo());
    ASSERT(response != infoResponses.end());
    channel->onIQReceived(iq.createResponse(new DiscoInfo(response->second)));
```

```
  }
  else {
    FAIL("Unexpected IQ");
  }
}
```

❶ Whenever an IQ is sent, pass it to `respondToIQ`, which will respond to it.

In this section, I showed how you can apply the high level of testing described in the previous section on more complex multistage protocols. Because the tests aren't cluttered by low-level protocol representational details, the tests can focus on testing the actual logic of the protocol, allowing the number of tests to grow without compromising the beauty of the protocol test suite.

Testing Session Initialization

By looking at both the single- and multistage request/response protocols from the previous sections, we covered most of the XMPP protocols out there. Although the level of testing for these protocols was already rather high, some protocols are still so complex that even testing at the level of "abstract" payloads results in too much clutter for a beautiful test. These are typically protocols that have a complex state diagram, and possibly even require user input during the process. We therefore bring in a higher level of testing: *scenario testing*.

One of the most complex protocols in XMPP is *session initialization*. Session initialization in an IM client involves creating a connection to the server; negotiating parameters of the connection (e.g., using stream compression for lower bandwidth consumption, encrypting the stream for better security, and so on); and finally authenticating with the server (typically involving sending a username and password to the server). Which parameters to negotiate with the server depends on what features the client and the server support, and also on the user preferences of the client. For example, a server might not support stream encryption; depending on whether the user has stated that he only wants to communicate over an encrypted connection, the client should either report an error or fall back on an unencrypted connection, respectively.

Testing all the possible code paths in session initialization requires a concise way of describing a session initialization scenario. Example 7-9 shows such a scenario test where the client encrypts the connection. By introducing helper methods describing what the client is supposed to send and what the server would send in response, we can clearly see how the encryption scenario is supposed to happen. It is easy to create scenarios for error conditions such as the server not supporting encryption (as shown in Example 7-10), and even to test the client's reaction to failing network connections (shown in Example 7-11). Moreover, creating these helper methods doesn't require all that much code, as they involve only setting expectations and responses on payloads, which can be written at the same level as the sections before.

EXAMPLE 7-9. Testing session encryption negotiation

```
void SessionTest::testStart_Encrypt() {
  Session* session = createSession("alice@wonderland.lit/RabbitHole");
  session->setEncryptConnection(Session::EncryptWhenAvailable);
  session->start();

  sessionOpensConnection();
  serverAcceptsConnection();
  sessionSendsStreamStart(); ❶
  serverSendsStreamStart(); ❷
  serverSendsStreamFeaturesWithStartTLS(); ❸
  sessionSendsStartTLS(); ❹
  serverSendsTLSProceed(); ❺

  ASSERT(session->isNegotiatingTLS());

  completeTLSHandshake(); ❻
  sessionSendsStreamStart(); /* (*) Immediately after the handshake, the
    stream is reset, and the stream header is resent in an encrypted form. */
  serverSendsStreamStart();

  ASSERT(session->isConnected());
  ASSERT(session->isEncrypted());
}
```

❶ Before sending XML elements over the stream, the client initializes the stream by sending an opening `<stream>` tag. All subsequent elements are children of this element. When the connection is closed, the closing `</stream>` tag is sent.

❷ Similar to the client, the server also starts the stream by sending a `<stream>` tag.

❸ Immediately after sending the opening stream tag, the server sends a list of all the features it supports. In this case, it announces support for stream encryption using StartTLS.

❹ The client sends a `<starttls/>` element to request the server to encrypt the connection.

❺ The server responds with a `<proceed/>`, indicating that the TLS negotiation (or *handshake*) can start.

❻ Fake a successful TLS handshake.

EXAMPLE 7-10. Testing session failure due to the server not supporting encryption

```
void SessionTest::testStart_ForceEncyptWithoutServerSupport() {
  Session* session = createSession("alice@wonderland.lit/RabbitHole");
  session->setEncryptConnection(Session::AlwaysEncrypt);
  session->start();

  sessionOpensConnection();
  serverAcceptsConnection();
  sessionSendsStreamStart();
  serverSendsStreamStart();
  serverSendsStreamFeaturesWithoutStartTLS();
```

```
  ASSERT(session->hasError());
}
```

EXAMPLE 7-11. Testing session failure due to a failing connection

```
void SessionTest::testStart_FailingConnection() {
  Session* session = createSession("alice@wonderland.lit/RabbitHole");
  session->start();

  sessionOpensConnection();
  serverAcceptsConnection();
  sessionSendsStreamStart();
  serverSendsStreamStart();
  closeConnection();

  ASSERT(session->hasError());
}
```

With scenario-based testing, it is possible to test the most complex class of protocols, covering all their corner cases. Although many of the corner cases of each "stage" in such protocols can be tested separately in isolation, scenarios are still needed to test the interaction between the multiple stages of the protocol.

Automated Interoperability Testing

By using unit tests to test our protocols in isolation (without a real network connection to an XMPP server), we were able to test all corner cases of a protocol while keeping our tests clean, simple, fast, and reliable. However, an XMPP client doesn't live in isolation; its purpose is to eventually connect to a *real* XMPP server and talk to *real* clients. Testing an XMPP client in the real world is important for several reasons. First of all, it allows you to check the functionality of your application at a larger scale than the local unit testing, ensuring that all the components work together correctly. Second, by communicating with other XMPP protocol implementations, you can test whether your interpretation of the protocol specification is correct. Finally, by testing your client against many different XMPP implementations, you are able to ensure interoperability with a wide collection of XMPP software. Unless you are developing a dedicated client to connect to only one specific server, testing interoperability with other clients and servers is very important in an open, heterogeneous network such as XMPP.

Because IM clients are driven by a user interface, testing interoperability between two clients is typically done manually: both clients are started, they connect to a server, an operation is triggered through the user interface of one client, and the other client is checked to determine whether it responds correctly to the operation. Fully automating UI-driven features is very hard.

Testing client-to-server interoperability is somewhat easier than testing client-to-client communication. By creating a small headless test program on top of the client's XMPP protocol

implementation, we can test whether the basic XMPP functionality of the backend works correctly, and even test complex protocols such as session initialization in action. For example, consider the test program in Example 7-12. This program logs into the server, fetches the user's contact list (also called the *roster*), and returns successfully if it received the roster from the server. By running this program, we can test whether most parts of our XMPP client's backend work: network connection, session initialization, stream compression, stream encryption, sending IQ requests, notifications of IQ responses, and so on.

EXAMPLE 7-12. A test program to connect to a server and request the roster; the JID and password of the account are passed through the environment

```
XMPPClient* xmppClient = NULL;
bool rosterReceived = false;

int main(int argc, char* argv[]) {
  xmppClient = new XMPPClient(getenv("TEST_JID"), getenv("TEST_PASS"));
  xmppClient->onConnected.connect(&handleConnected); ❶
  xmppClient->connect();
  return rosterReceived;
}

void handleConnected() {
  GetRosterRequest* rosterRequest = new GetRosterRequest(xmppClient);
  rosterRequest->onResponse.connect(bind(&handleRosterResponse, _1, _2)); ❷
  rosterRequest->send();
}

void handleRosterResponse(RosterPayload*, optional<Error> error) {
  rosterReceived = !error; ❸
  xmppClient->disconnect();
}
```

❶ When connected (and authenticated), call `handleConnected`.

❷ When a response for the roster request is received, call `handleRosterReceived` with the response and status as arguments.

❸ If there was no error, we received the roster properly.

A program similar to the one from Example 7-12 is run as part of Swift's automated test suite. We use a few different server implementations on every test run, passing the test JID and password of each server through the environment. If `ClientTest` fails due to a bug in a protocol implementation, a new unit test is added and the protocol is fixed. If the bug is due to a certain combination of protocols not being handled properly (either by the client or by the server), a scenario test is added to reproduce the scenario, after which either the client bug is fixed or the client is adapted to work around a specific server implementation bug.

When using automated tests like the one just described, a project is of course always limited to testing against the implementations it has access to. Although it is always possible to test against the handful of free XMPP server implementations out there, testing against

implementations from commercial vendors isn't always straightforward. To make it easier for the XMPP community to test their implementations against each other, there has been an initiative to create a centralized place that provides access to test accounts on all server implementations out there, including ones from commercial vendors.† This initiative paves the way to easy, automated interoperability testing for XMPP projects.

Diamond in the Rough: Testing XML Validity

When testing the functionality of our protocol in the previous sections, we separated the stanza representation from the actual logic of the protocol to improve the focus of our tests. This split made testing the logic of the protocol straightforward and clean. Unfortunately, testing the representational part that converts the abstract stanzas into XML and back is still tedious and error-prone. One of the things that needs to be checked is whether every variation of the payload transforms correctly to a standards-compliant XML element. For example, for the version payload we used earlier, we need to test the representation of a payload with and without a version number. Conversely, the transformation from XML to a payload data structure needs to be tested for every possible compliant XML element. It would be handy if we could automatically check whether our XML parser and serializer handles all of the possible variations of payloads and stanzas allowed by the protocol standard.

A possible approach for testing XML parsers and serializers is automated *XML validation*. Every protocol specification published by the XMPP Standards Foundation comes with an *XML schema*. This schema describes the syntax and constraints of the XML used in the protocol. For example, it specifies the names of the XML elements that can occur in the payload, the names and types of the attributes of these elements, the number of times an element can occur, and so on. Such XML schemas are typically used to test whether a piece of XML is syntactically valid according to the rules specified in the schema. Unfortunately, the XMPP schemas currently serve only a descriptive purpose, and are only used to document the protocol. This is why using the XMPP schemas in automated processes, such as validity checking, is still mostly unexplored terrain. However, there has been interest lately in making normative XML schemas, which would open up some more possibilities for making the tedious testing of XMPP parsing and serialization more pleasant and, who knows, even beautiful!

Conclusions

In our quest to create beautiful tests for checking XMPP protocol implementations, we started out by testing simple request-response protocols at the lowest level: the data sent over the network stream. After discovering that this form of testing does not really scale well, we abstracted out the protocol to a higher level, up to the point where the tests used only

† Unfortunately, this initiative has currently been put on hold for more urgent matters, but it will hopefully be revived soon.

high-level data structures. By testing protocol behavior on a high level, we were able to write tests for more complex protocols without compromising the clarity of the tests. For the most complex protocols, writing scenarios helped to cover all of the possible situations that can arise in a protocol session. Finally, since XMPP is an open protocol with many different implementations, it's very important to test an XMPP application on the real network, to ensure interoperability with other implementations. By running small test programs regularly, we were able to test the system in its entirety, and check whether our implementation of the protocol plays together nicely with other entities on the network.

The focus of this chapter has mostly been on testing the protocol functionality in XMPP implementations, as this is probably the most important part of quality control in XMPP. However, many other forms of testing exist in the XMPP world besides protocol tests. For example, performance testing is very crucial in the world of XMPP servers. Simple test scripts or programs like the ones described earlier can be used to generate a high load on the server, to test whether the server can handle increasing amounts of traffic. In XMPP clients, on the other hand, testing the user interface's functionality is very important. Although automated UI testing is known to be hard, many complex parts, such as contact list representation, can be unit tested in isolation, which can avoid bugs in vital pieces of client code.

Although it's already possible to write simple, clean, and thorough tests for many aspects of XMPP, there's still a lot of beauty in testing waiting to be discovered. If you have suggestions or ideas, or want to help work on improving testing in the XMPP community, feel free to stop by *http://xmpp.org* and join the conversation!

References

[XML-C14n] Boyer, John. 2001. *Canonical XML*.

[DOM] Le Hégaret, Philippe. 2002. The W3C Document Object Model (DOM).

[RFC 3920] Saint-Andre, Peter. 2004. *Extensible Messaging and Presence Protocol: Core*.

[RFC 3921] Saint-Andre, Peter. 2004. *Extensible Messaging and Presence Protocol: Instant Messaging and Presence*.

[XEP-0092] Saint-Andre, Peter. XEP-0092: *Software Version*.

[XMPP TDG] Saint-Andre, Peter. Smith, Kevin. Tronçon, Remko. 2009. *XMPP: The Definitive Guide*. Cambridge: O'Reilly.

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.