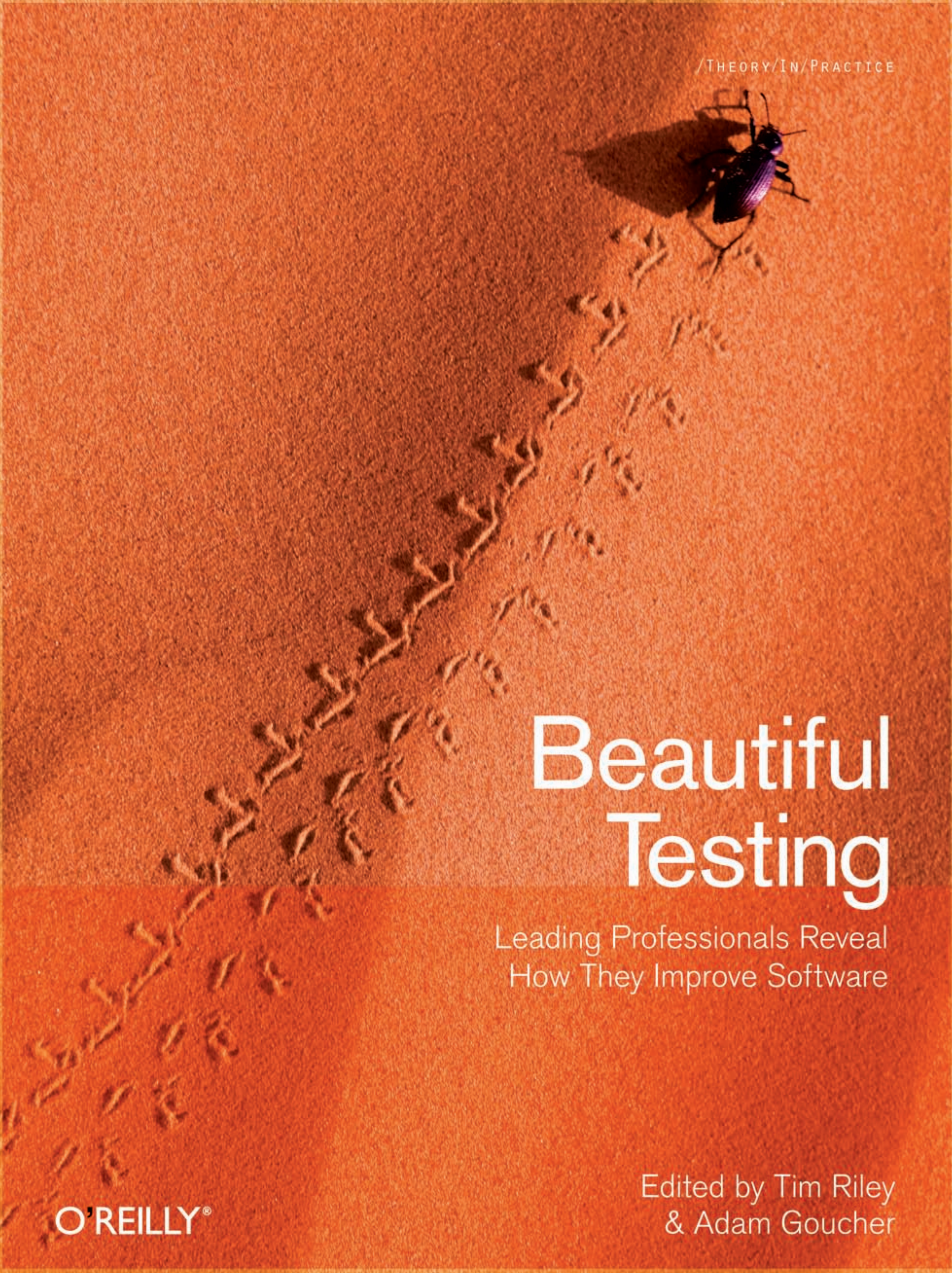
/THEORY/IN/PRACTICE



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

54999

9 780596 159818

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

O'REILLY® oreilly.com

Beautiful Testing

Edited by Tim Riley and Adam Goucher

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Beautiful Testing
Edited by Tim Riley and Adam Goucher

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

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 TEN

Testing a Random Number Generator

John D. Cook

According to the classical definition of beauty, something is beautiful if it exhibits both complexity and unity. Professor Gene Veith explained this idea in an editorial by describing two kind of paintings:*

> In painting a black canvas has unity, but it has no complexity. A canvas of random paint splatterings has complexity, but it has no unity.

Michelangelo's painting of the Sistine Chapel ceiling has rich detail along with order and balance. It exhibits complexity and unity. It is beautiful.

Some works of beauty are easy to appreciate because both the complexity and the unity are apparent. I would say the Sistine Chapel falls in this category. However, other works require more education to appreciate because it takes knowledge and skill to see either the complexity or the unity. Modern jazz may fall into this latter category. The complexity is obvious, but the unity may not be apparent to untrained ears. Tests for random number generators may be more like modern jazz than the Sistine Chapel; the complexity is easier to see than the unity. But with some introduction, the unity can be appreciated.

* Veith, Gene Edward. "Acquired taste," *World Magazine*. February 29, 2008.

What Makes Random Number Generators Subtle to Test?

Software random number generators are technically *pseudo*random number generators because the output of a deterministic program cannot really be random. We will leave the "pseudo" qualification understood and simply speak of random number generators (RNGs). Even though the output of an RNG cannot actually be random, there are RNGs that do a remarkably good job of producing sequences of numbers that for many purposes might as well be truly random. But how do you know when the output of an RNG is sufficiently similar to what you would expect from a true random source?

A good RNG leads us to believe the output is random, so long as we look only at the output and don't restart the sequence. This is our first hint that RNGs are going to be subtle to test: *there is a tension in the requirements for an RNG.* The output should be unpredictable from one perspective, even though it's completely predictable from another perspective. Tests must verify that generators have the right properties from the perspective of user applications while not being distracted by incidental properties.

The idea of what constitutes a good RNG depends on how the RNG is applied. That is why, for example, a generator may be considered high-quality for simulation while being considered unacceptable for cryptography. This chapter looks at tests of statistical quality and does not address tests for cryptographical security.

Suppose we are asked to generate random values between 3 and 4. What if we wrote a program that always returned 3? That would not satisfy anyone's idea of a random sequence, because something "random" should be unpredictable in some sense. Random values should jiggle around. So next we write a program that yields the sequence 3, 3.1, 3.2, ..., 4, 3, 3.1, 3.2, ... in a cycle. The output values move around, but in a predictable way. We shouldn't be able to predict the next value of the sequence. The output values should spread out between 3 and 4, but not in such a simple way. Thinking about this leads to another reason RNGs are subtle to test: *there's no way to say whether a particular value is correct*. We cannot test individual values; we have to look at things like averages.

Even when we look at averages, there are still difficulties. Suppose the output of a random number generator is supposed to have an average value of 7, and the first output value is 6.5. That's OK, because the sequence does not (and should not) always return 7; it should just average to 7 in the long run. Should the next value be 7.5 so that the average is correct? No, that would make the second value predictable. So should the average work out to 7 after three outputs? No, then the third value would be predictable. We can never require the average to be exactly 7 after any fixed number of outputs. What we can say is that as we average over longer and longer output sequences, the average should *often* be *close* to 7. The weasel words "often" and "close" are the heart of the difficulty. These terms can be made precise, but it takes work to do so.

Since we must write tests that verify that certain things happen "often," we have to quantify what we mean by "often." And we cannot simply say "always." This brings up a third reason why testing RNGs is subtle: *any test we write will fail occasionally*. If a test never fails, then it demonstrates a predictable attribute of our random number sequence. So not only *can* our tests fail from time to time, they *should* fail from time to time!

What are we to do if it is impossible in principle to write tests that will always pass? Ultimately, some subjective judgment is required. However, we can do better than simply printing out a list of output values and asking a statistician whether the sequence looks OK. It all goes back to the terms "often" and "close." These two concepts are often traded off against each other. We can decide what "often" means and then pick the corresponding notion of "close." If we want tests that should only fail around once in every 1,000 runs, we can pick a definition of "close" to make that happen. But only on average! Even a test that fails on average once every 1,000 runs may fail twice in a row.

Uniform Random Number Generators

It doesn't make sense to ask for a program that generates random numbers without some further information. What kind of numbers: integers or floating-point? From what range? Should all values be equally likely, or should some values be more likely than others? In statistical terms, we need to know what distribution the numbers should follow. Only then can we test whether software satisfies the requirements.

The most fundamental RNG produces values from the unit interval[†] with all values equally likely. This is called a uniform RNG. When people say they want random numbers but give no further details, this is often what they have in mind. This is the most important random number generation problem because, once it is solved, we can bootstrap the solution to solve other problems. In other words, the grand strategy for generating random numbers is as follows:

1. Generate random values uniformly distributed in the interval (0, 1).
2. Transform those values into whatever other distribution you need.

The first step in this grand strategy is the hardest part. Fortunately, this problem has been solved for most practical purposes. There are uniform RNG algorithms, such as the Mersenne Twister,[‡] that have been extensively studied by experts. These algorithms have good theoretical properties and have been through empirical gauntlets such as George Marsaglia's DIEHARD battery of tests.[§] It's very easy to *think* you have created a good RNG when you haven't, so

† Generators are inconsistent as to whether the end points should be included. In my opinion, excluding both end points causes the lest difficulty. But some generators include one or both end points.

‡ Matsumoto, Makoto, and Takuji Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Trans. Model. Comput. Simul.* Vol. 8, No. 1 (1998).

§ Gentle, James E. *Random Number Generation and Monte Carlo Methods*. Springer, 1998.

most of us should use a uniform RNG that has been vetted by experts. Implementations of these standard algorithms are easy to find.

Although most people will not need to come up with new *algorithms* for uniform random number generation, more people may need to come up with new *implementations* of standard algorithms, and of course these implementations will need to be tested. For example, you may need to test a port of a standard algorithm implementation to a new programming language. In that case, you could generate, say, a million values from the original generator, then set the same seed values in the new generator and verify that you get the exact same sequence of outputs. This is an exception to the rule that there are no deterministic tests for random number generators.

Most testers will be concerned with the second step of the grand strategy: testing code that transforms uniform random sequences into sequences with other distributions. Although very few people develop their own uniform generators, many people have reasons to develop custom nonuniform generators.

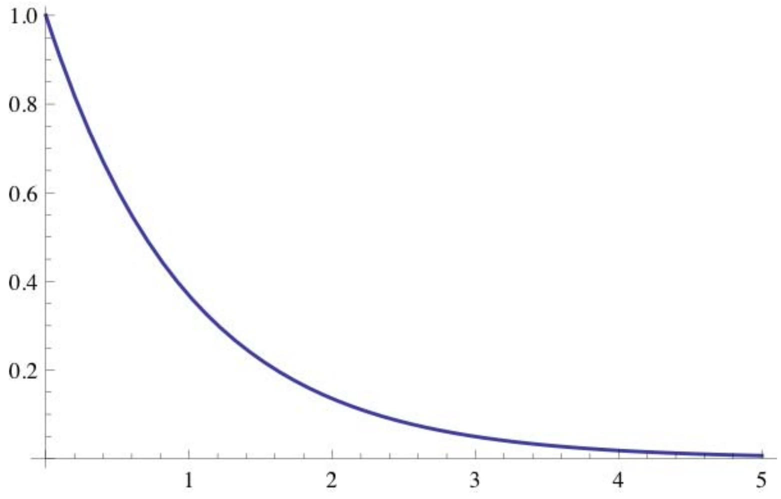
Nonuniform Random Number Generators

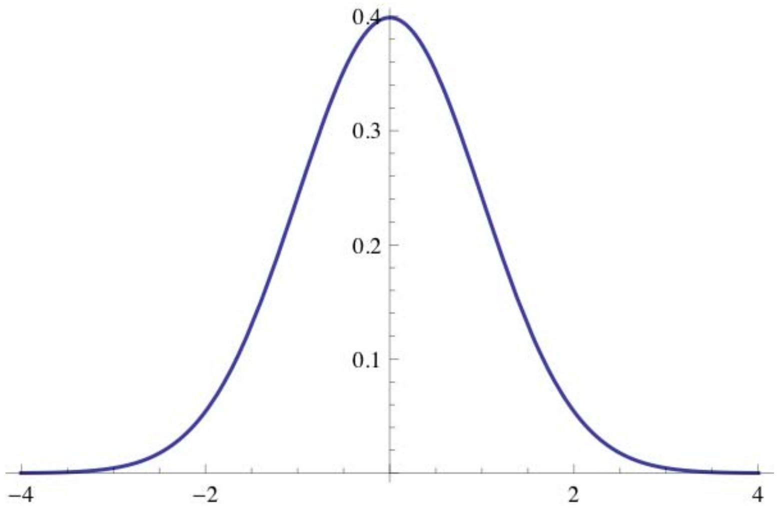
Suppose you want to decide how many cashiers to hire for a grocery store. If shoppers arrived at regular intervals and all took the same amount of time to check out, this would be a simple problem. But in reality, shoppers do not arrive like clockwork. Nor do they all take the same amount of time to check out. Some shoppers dart into the store for a pack of gum, whereas others come to stock up on a week's worth of provisions for a family of eight.

There is an entire branch of mathematics called *queuing theory* devoted to studying problems like how many cashiers a grocery store needs. Often, queuing theory assumes that the time needed to serve a customer is exponentially distributed. That is, the distribution of service times looks like the function e^{-x}. A lot of customers are quick to check out, some take a little longer, and a few take very long. There's no theoretical limit to how long service may take, but the probability decreases rapidly as the length of time increases, as shown in the first image that follows. The same distribution is often used to model the times between customer arrivals.

The exponential distribution is a common example of a nonuniform distribution. Another common example is the Gaussian or "normal" distribution.[‖] The normal distribution provides a good model for many situations: estimating measurement errors, describing the heights of Australian men, predicting IQ test scores, etc. With a normal distribution, values tend to clump around the average and thin out symmetrically as you move away from the middle, as shown in the second image that follows.

‖ The normal distribution was not so named to imply that it is "normal" in the sense of being typical. The name goes back to the Latin word *normalis* for perpendicular. The name has to do with a problem that motivated Carl Friedrich Gauss to study the distribution.





Software is readily available to generate random values from well-known distributions such as the exponential and the normal. Popular software libraries have often been very well tested, though not always. Still, someone using an RNG from a library would do well to write a few tests of their own, even if they trust the library. The point is not only to test the quality of the library itself, but also to test the user's *understanding* of the library.

One of the most common errors along these lines involves misunderstanding parameterizations. For example, the normal distribution has two parameters: the mean μ and standard deviation σ. It is common for people to specify either the standard deviation σ or the variance σ^2. If the documentation says an RNG gives samples from a normal (3, 8) distribution, does that mean the standard deviation is 8 (and the variance is 64), or does it mean the standard deviation is $\sqrt{8}$ and the variance is 8?

To make matters worse, some of the most common differences in parameterization conventions don't show up when using default values. For example, the standard deviation for a normal distribution defaults to 1. But if $\sigma = 1$, then $\sigma^2 = 1$ as well. Some people even parameterize the normal in terms of the precision, $1/\sigma^2$, but that also equals 1 in this case. A test using default values would not uncover a misunderstanding of the parameter conventions. The exponential distribution is similar. Some folks parameterize in terms of the mean μ, and others use the rate $1/\mu$. Again the default value is 1, and so any confusion over whether to use mean or rate would be masked by the default parameterization.

No library is going to contain every distribution that every user would want. The C++ Standard Library, for example, provides only seven distributions, but there are dozens of distribution families in common use. New distributions that may be unique to a particular problem are invented all the time. Many people have to write custom nonuniform random number generators, and there are common techniques for doing so (inverse CDF transformation, accept-reject algorithms, etc.).

A Progression of Tests

If a nonuniform random generator has a high-quality uniform random number generator at its core, the main thing to test is whether the generator output has the correct distribution. Fortunately, tiny coding mistakes often result in egregious output errors, so simple tests may be adequate to flush out bugs. However, some bugs are more subtle, and so more sophisticated tests may be necessary. The recommendation is to start with the simplest tests and work up to more advanced tests. The simplest tests, besides being easiest to implement, are also the easiest to understand. A software developer is more likely to respond well to being told, "Looks like the average of your generator is 7 when it should be 8," than to being told, "I'm getting a small *p*-value from my Kolmogorov-Smirnov test."

Range Tests

If a probability distribution has a limited range, the simplest thing to test is whether the output values fall in that range. For example, an exponential distribution produces only positive values. If your test detects a single negative value, you've found a bug. However, for other distributions, such as the normal, there are no theoretical bounds on the outputs; all output values are possible, though some values are exceptionally unlikely.

There is one aspect of output ranges that cannot be tested effectively by black-box testing: boundary values. It may be impractical to test whether the endpoints of intervals are included. For example, suppose an RNG is expected to return values from the half-open interval $(0, 1]$. It may not be practical to verify that 1 is a possible return value because it would only be returned rarely. Also, if algorithm incorrectly returned 0, but did so rarely, it may not be detected in testing. These sort of boundary value errors are common, they can be hard to detect via testing, and they can cause software to crash.

Mean Test

One of the most obvious things to do to test a random number generator is to average a large number of values to see whether the average is close to the theoretical value. For example, if you average a million values from a normal random generator with mean 4 and standard deviation 3, you'd expect the average to be near 4. But *how* near?

For the special case of the normal distribution, there's a simple answer. The average of a sequence of independent normal values is itself a normal random value with the same mean. So if we average a million samples from a normal with mean 4, the average will have a normal distribution with mean 4. But the standard deviation of the mean is smaller than the standard deviation of the individual samples by a factor of $1/\sqrt{n}$, where n is the number of samples. So in this example the average of our samples will have standard deviation $3/\sqrt{10^6} = 0.003$. An important rule of thumb about normal distributions is that samples will lie within 2 standard deviations of the mean about 95% of the time. That means if we take a million values from a normal random number generator with mean 4 and standard deviation 3, we would expect the average to be between 3.994 and 4.006 around 95% of the time. However, such a test will fail about 5% of the time. If you'd like your test to pass more often, make your criteria looser. For example, samples from a normal distribution are within 3 standard deviations of the mean 99.7% of the time, so testing whether the average of a million samples is between 3.991 and 4.009 will fail in only about three out of every thousand tests.

In summary, the way to test samples from a normal random number generator with mean μ and standard deviation σ is to average n values for some large value of n, say $n = 10^6$. Then look for the average to be between $\mu - 2\sigma/\sqrt{n}$ and $\mu + 2\sigma/\sqrt{n}$ around 95% of the time, or between $\mu - 3\sigma/\sqrt{n}$ and $\mu + 3\sigma/\sqrt{n}$ around 99.7% of the time.

In practice, exactly how often the tests fail might not be important. If there's a bug that throws off the average of the random values, it's likely your tests will fail every time. These kinds of errors are usually not subtle. Either the tests will pass most of the time or fail nearly every time. I cannot recall ever wondering whether a test was failing a little too often.

How do you test an RNG that doesn't produce a normal distribution? How would you test, for example, an exponential RNG? There's a bit of magic called the central limit theorem that says if we average enough values, any[#] distribution acts like a normal distribution. For an exponential distribution with mean μ, the standard deviation is also μ. If we average a million samples from that distribution, the average will very nearly have a normal distribution with mean μ and standard deviation μ/1,000. As before, we can test whether the average falls between two or three standard deviations of what we expect.

The central limit theorem provides a better approximation for some distributions than others. The more the distributions start out looking something like a normal distribution, the faster the averages will converge to a normal distribution. However, even for distributions that start out looking nothing like a normal, the averages start to be approximately normal fairly quickly.[*] However, as was mentioned earlier, we can often get by without paying too close attention to such details. Suppose you expect your test to fail about 5% of the time, but instead it fails 10% of the time. In that case you probably do not have a bug that is throwing off the average. And if the test fails every time, a bug is a more plausible explanation for the failure than any subtle behavior of the averages.

The only case that is likely to cause problems is if the distribution being tested does not even have a mean. For example, the Cauchy distribution looks something like a normal distribution, but goes to zero more slowly as you move away from the middle. If fact, it goes to zero so slowly that the mean value does not exist. Of course you can always take the average of any number of samples, but the averages are not going to settle down to any particular value. In that case, the central limit theorem does not apply. However, you could do something analogous to a mean test by testing the *median*, which always exists.[†]

Variance Test

The mean test can flush out certain kinds of coding errors. However, it is easy to write incorrect code that will pass the mean test: always return the mean! For example, suppose an RNG is supposed to return values from a normal distribution with mean 4 and standard deviation 9. Code that always returns 4 will pass the test. Code that returns normal random values with mean 4 and *variance* 9 would also pass the mean test. The latter is more subtle and more likely to occur in practice.

The combination of a mean and variance test gives much more assurance that software is correct than just the mean test alone. Some software errors, such as the standard deviation–variance confusion just mentioned, leave the mean unaffected but will throw off the variance.

Incidentally, it is a good idea to use widely different values for mean and standard deviation. Suppose software swapped the values of the mean and standard deviation parameters on input.

# Certain restrictions apply. See your probability textbook for details.

* If you really want to know the details of how quickly the averages converge to the normal distribution, look up the Berry-Esséen theorem.

† See *http://www.johndcook.com/Cauchy_estimation.html*.

Testing with samples from a normal with mean 7 and standard deviation 7 would not uncover such a bug. Testing with samples from a normal with mean −2 and standard deviation 7 would be much better because −2 is an illegal value for a standard deviation and should cause an immediate error if parameters are reversed.

Just as the mean test compares the mean of the samples to the mean of the distribution, the variance test compares the variance of the samples to the variance of the distribution. Suppose we compute the sample variance of some large number of outputs from an RNG; say, one million outputs. The outputs are random, so the sample variance is random.[‡] We expect that it will "often" be "near" the variance of the distribution. Here we return to the perennial question: how often and how near?

As with testing the mean, the answer is simplest for the normal distribution. Suppose an RNG produces values from a normal distribution with variance σ^2. Let S^2 be the sample variance based on n values from the RNG. If n is very large, then S^2 approximately has a normal distribution with mean σ^2 and variance $2\sigma^4/(n-1)$.[§] As before, we apply the idea that anything with a normal distribution will lie within two standard deviations of its mean 95% of the time.

For example, suppose we're testing the variance of samples from a generator that is supposed to return values from a normal distribution with mean 7 and variance 25 (standard deviation 5). The mean value 7 is irrelevant for testing the variance. Suppose we compute the sample variance S^2 based on 1,000 values. Then S^2 should have mean 25 and variance $2 \cdot 5^4/999$. This would put the standard deviation of S^2 at about 1.12. Thus we would expect S^2 to be between 22.76 and 27.24 about 95% of the time.

If the random number generator to be tested does not generate values from a normal distribution, we fall back on the central limit theorem once again and gloss over the nonnormality. However, the use of the central limit theorem is not as justified here as it was when we were testing means. Typically, sample variances will be more variable than the normal approximation predicts. This means our tests will fail more often than predicted. But as before, we may not need to be too careful. Coding errors are likely to cause the tests to fail every time, not just a little more often than expected. Also, the tests in the following sections do a more careful job of testing the distribution of the samples and have a solid theoretical justification.

Just as some distributions do not have a mean, some distributions do not have a variance. Again, the Cauchy distribution is the canonical example. Calculating the sample variance of Cauchy random numbers is an exercise in futility. However, the following tests apply perfectly well to a Cauchy distribution.

‡ See *http://www.johndcook.com/standard_deviation.html* for how to compute sample variance. If you just code up a formula from a statistics book, you might be in for an unpleasant numerical surprise.

§ This isn't the way this result is usually presented in textbooks. More precisely, $(n-1)S^2/\sigma^2$ has a $\chi^2(n-1)$ distribution, that is, a chi-squared distribution with $n-1$ degrees of freedom. But when n is large, we can approximate the χ^2 distribution with a normal and obtain this result.

Bucket Test

Suppose an RNG passes both the mean and the variance test. That gives you some confidence that the code generates samples from *some* distribution with the right mean and variance, but it's still possible the samples are coming from an entirely wrong distribution.

To illustrate this potential problem, consider two generators. The first returns values from an exponential distribution with mean 1 (and hence standard deviation 1). The second returns values from a normal distribution with mean 1 and standard deviation 1. The mean and variance tests could not detect an error that resulted in calls to the two generators being swapped. What kind of test could tell the two generators apart? You could count how many values fall into various "buckets," or ranges of values. For example, you could count how many output values fall between −1 and 0, between 0 and 2, and greater than 2. The difference between the distributions will be most obvious in the bucket of values between −1 and 0. The exponential generator will have *zero* values in that bucket, whereas the normal generator will have about 19% of its values in the same range. Usually we would be looking for more subtle cases where all buckets would have some values and the only question is whether some buckets have too many or too few samples. (What we call "the bucket test" here is commonly known as the chi-square (χ^2) test.‖ And some people use the term "bin" where we use "bucket.")

Here's how to do a bucket test. Divide your output range into *k* buckets. The buckets should cover the entire range of the output and not overlap. Let E_i be the expected number of samples for the *i*th bucket, and let O_i be the number of samples you actually observe. Then, compute the chi-square statistic:

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

If this value is too large, that says the observed counts are too different from the expected counts and so maybe our generator does not follow the right distribution. If this value is too small, that says the expected counts agree too well with the expected values and don't have enough random variation.

The description here leaves several questions unanswered. How many buckets should you use? Where should you put them? How do you know when χ^2 is too large or too small?

First we consider the number of buckets. If there are too few buckets, the test is not very demanding and errors could go undetected. On the other hand, if there are too many buckets, then we do not expect to find many samples in each bucket and the theoretical requirements of the test are not met. A common rule of thumb is that the expected number of samples in

‖ Knuth, Donald E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Third Edition. Addison-Wesley, 1998.

each bucket should be at least five.[#] This is no problem because we are *generating* our data rather than collecting it. We can determine our number of buckets first, then choose the number of samples n so large that we expect well more than five samples in each bucket.

Next, how do we decide where to place the buckets? We could arbitrarily decide on, say, 10 buckets and pick the boundaries so that we expect an equal number of samples in each bucket. For example, if we wanted to test a Cauchy random number generator, we could use the Cauchy distribution function to find cutoff values x_i for $i = 1, 2, 3, \ldots, 9$ so that we expect 10% of the samples to be less than x_1, 10% between x_1 and x_2, 10% between x_2 and x_3, on up to 10% of the values greater than x_9.[*] Also, if you are particularly interested in some region, you could make that one of your buckets. For example, if someone is concerned that there are too many values greater than 17 for some generator, you could make a bucket for values greater than 17.

Finally, we address the range of values we should expect. If we have b buckets, the statistic χ^2 has a chi-square distribution with $b-1$ degrees of freedom. Since we're generating our samples and can have as many as we want, we might as well make b fairly large. (Remember to make the number of samples n large enough so that each bucket is expected to have at least five samples.) Then we can approximate the chi-square distribution by a normal and not have to consult tables to find a typical range of values. For large b, a chi-square distribution with $b-1$ degrees of freedom has approximately the same distribution as a normal distribution with mean $b-1$ and variance $2b-2$. Then we can use the same rules as before regarding how often a normal random variable is within two or three standard deviations of its mean.

Kolmogorov-Smirnov Test

One shortcoming of the bucket test is that it is "chunky." It is possible that a random number generator puts approximately the expected number of samples in each bucket and yet the samples are not properly distributed within the buckets. For example, suppose the bucket boundaries were at $m + 1/2$ for several integer values of m. If a bug in the random number generator causes all floating-point values to be rounded to the nearest integer, all samples would land exactly in the middle of the bucket. The counts in each bucket would not be affected by such a bug, and if the generator were otherwise correct, the bucket test would pass most of the time.

We would like a more fine-grained test of how the random samples are distributed. Here's one way to proceed. Take a large number of samples n. For each sample x_i we can compare the actual proportion of samples less than x_i to the proportion of samples we would expect to have

# Knuth, Donald E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Third Edition. Addison-Wesley, 1998.

* In case you're curious: $x_i = \tan(\pi(0.1\, i - 0.5))$.

seen. In other words, we will compare the *empirical* distribution function with the *theoretical* distribution function. The empirical distribution is defined as:

$$F_n(x) = \frac{\text{the number of } x_i \text{ values} \leq x}{n}$$

and the theoretical distribution function $F(x)$ is the theoretical probability of the RNG returning a value no greater than x. We want to look at the differences between $F(x)$ and $F_n(x)$. In short, we want to do a direct comparison of the theoretical and empirical distribution of values. This is the idea behind the Kolmogorov-Smirnov (K-S) test.†

To carry out the test, we calculate two numbers:

$$K^+ = \sqrt{n} \max_{\infty} \left(F_n(x) - F(x)\right)$$
$$K^- = \sqrt{n} \max_{\infty} \left(F(x) - F_n(x)\right)$$

Aside from the factor $\sqrt{n}$, the number K^+ is the maximum amount by which the empirical distribution exceeds the theoretical distribution. Similarly, K^- is the maximum amount by which the theoretical distribution exceeds the empirical distribution, multiplied by $\sqrt{n}$. If the theoretical and empirical distributions were to line up perfectly, K^+ and K^- would both be zero. It would be almost impossible for K^+ or K^- to be zero in practice, and values near zero are suspicious. At the other extreme, if the theoretical and empirical distributions do not line up well, either K^+ or K^- would be large and indicate a problem. So what would be suspiciously small and suspiciously large values of our statistics?

With the previous test, we appealed to the central limit theorem to reduce our problem to a normal distribution. That's not going to work here. We're going to pull a rabbit out of the hat and give range values without saying where they came from. For large n, we expect K^+ to be between 0.07089 and 1.5174 around 98% of the time. How large does n need to be for this to hold? Values of n that you would want to use for testing, say $n = 1{,}000$, are more than big enough. Donald Knuth's book gives more details concerning the K-S test, such as an explanation of where the range values come from and how to find your own values based on how often you want the test to pass.

If the K-S test usually passes, this is strong evidence that the transformation from uniform to nonuniform random values was implemented correctly. In that case, if the uniform RNG is trustworthy, then the nonuniform generator is trustworthy. Of course it is possible that a bug could still slip through the process, but this is unlikely. If the K-S test fails, examining the values of x that determine K^+ and K^- could help developers locate the bug in the RNG.

† Knuth, Donald E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Third Edition. Addison-Wesley, 1998.

Conclusions

Tests for evaluating RNGs can exhibit complexity as well as unifying order. Such tests are beautiful by the classical definition of beauty. RNGs are complex because they are deterministic programs that must give the illusion of being nondeterministic. Tests of RNGs are at least as complex as the generators they validate. These tests are complex because we can seldom say anything absolute about how the RNG should behave. We have to be content with statements about how they should *usually* behave.

There are several unifying principles in testing random number generators:

- Tests boil down to saying some statistic should often be inside a certain range. The more demanding your idea of "often" is, the wider your range must be.
- Nonuniform RNGs transform the output of uniform RNGs. If you have confidence in your uniform RNG, you only have to test the distribution properties of your nonuniform RNG. The more subtle properties of randomness are inherited from the uniform RNG and do not need to be as carefully tested.
- Precise analysis of random sequences requires advanced statistics, and yet we can get a lot of mileage out of one simple observation: samples from a normal distribution are often within two or three standard deviations of their mean.
- We do not have to look too closely at how often tests pass. Correct generators usually pass tests, and buggy generators usually fail.

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.