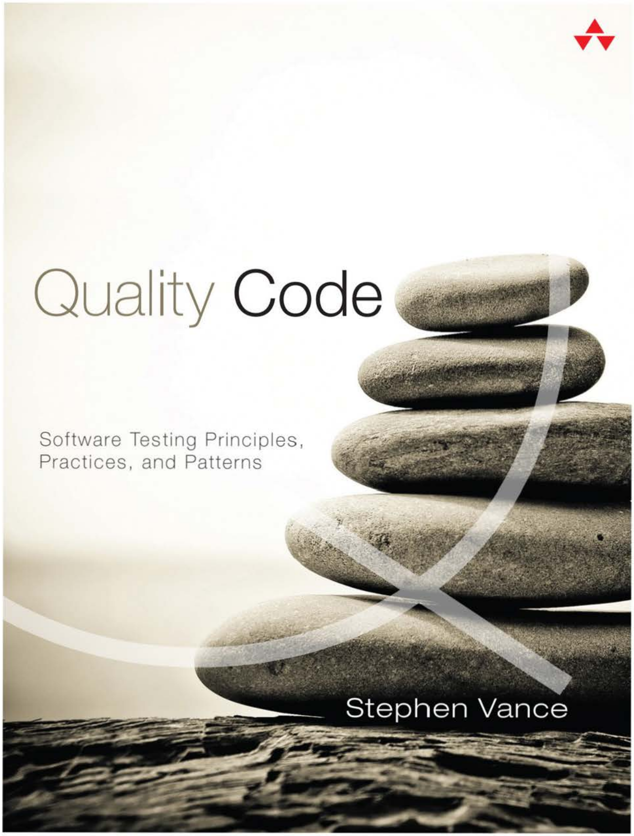
Quality Code

Software Testing Principles, Practices, and Patterns

Stephen Vance



Quality Code

This page intentionally left blank

Quality Code

Software Testing Principles, Practices, and Patterns

Stephen Vance

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Vance, Stephen.

Quality code : software testing principles, practices, and patterns / Stephen Vance.

pages cm

Includes index.

ISBN 978-0-321-83298-6 (alk. paper)—ISBN 0-321-83298-1 (alk. paper)

1. Computer software—Testing. 2. Software patterns. I. Title. II. Title: Software testing principles, practices, and patterns.

QA76.76.T48V36 2013

005.1′4—dc23 2013036018

ISBN-13: 978-0-321-83298-6
ISBN-10: 0-321-83298-1
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, November 2013

To my 자기야
for your continuous love and support.

To my parents
for preparing me for this and all other accomplishments.

Contents

Preface

Lean production has revolutionized manufacturing over the last several decades. Frameworks like TQM, Just-In-Time, Theory of Constraints, and the Toyota Production System have vastly improved the general state of automotive and manufacturing quality and spawned a vital competitive landscape. The family of Agile software development approaches brings the principles of Lean production to the knowledge discipline of software product development, but it needs to adapt these principles to a less mechanical context.

The idea of building the quality into the product in order to increase customer satisfaction and reduce the overall lifetime cost of maintenance has resulted in practices like test-driven development (TDD) and other test-first and test-early approaches. Regardless of which flavor you espouse, you need to understand what testable software looks like and master a wide range of techniques for implementing tests to be successful. I have found the gap between the principles and the state of the practice is often an unacknowledged source of failure in testing regimens. It is easy to say "use test-driven development," but when faced with a project, many developers do not know where to start.

In showing people how to apply test-driven, or at least test-early, development, I often find that one of the obstacles is the mechanics of writing tests. Exercising a method as a mathematical function that purely translates inputs to easily accessible outputs is no problem. But much software has side effects, behavioral characteristics, or contextual constraints that are not as easy to test.

This book arose from a recurring need to show developers how to test the specific situations that stymied them. Consistently, we would sit down for a few minutes, we would write a unit test for the troublesome code, and the developer would have a new tool in his belt.

What Is This Book About?

This book is about easily and routinely testing all of your software. It primarily focuses on unit tests, but many of the techniques apply to higher-level tests as well. This book will give you the tools—the implementation patterns—to test almost any code and to recognize when the code needs to change to be testable.

What Are the Goals of This Book?

This book will help you

- Understand how to easily unit test all of your code
- Improve the testability of your software design
- Identify the applicable testing alternatives for your code
- And write tests in a way that will scale with the growth of your applications

  In support of these goals, this book provides you with

- Over two-dozen detailed techniques for testing your code with lots of examples
- Guidance on the right testing technique for a wide range of situations
- And an approach to unit testing that helps you focus your efforts

Who Should Read This Book?

This book is primarily written for the professional software developer and software developer in test who want to increase their skill at code-level testing as a means to improve the quality of their code. This book should be particularly useful to test-driven development (TDD) and test-early practitioners to help them ensure the correctness of their code from the start. Many of the techniques in this book also apply to integration and system testing.

What Background Do You Need?

This book assumes that the following points are true.

- You are sufficiently fluent in object-oriented languages to be able to read and understand examples in Java, C++, and other languages and apply them to your language of choice.
- You are familiar with concepts of code-level testing and xUnit framework members such as JUnit.
- You are conversant in or have ready reference to information on design patterns and refactoring.

What Is in This Book?

Part I (Chapters 1–5) covers the principles and practices that guide successful testing. Chapter 1 puts the approaches in the book into an engineering context, discussing engineering, craftsmanship, and first-time quality in general as well as some software-specific concerns. Chapter 2 examines the role of intent. Chapter 3 outlines an approach to testing to help you focus your efforts. Chapter 4 discusses the interaction between design and testability, including some thoughts on scaling your testing efforts. Chapter 5 presents a number of testing principles that you can use to guide your testing decisions.

Part II (Chapters 6–13) details the implementation patterns for testing, starting with bootstrapping your tests and the basic catalog of techniques in Chapter 6. Chapters 7 through 12 elaborate on the topics introduced in Chapter 6, with an interlude to investigate intent more deeply in Chapter 9. Chapter 13 takes a deep technical dive into what many people consider impossible by introducing techniques for deterministically reproducing race conditions.

Part III (Chapters 14 and 15) takes the principles and techniques from the rest of the book and provides a narrative for applying them in two real-life, worked examples. Chapter 14 examines using test-driven development to create a Java application from scratch, showing how to get started and how to apply the techniques in a strictly typed language. Chapter 15 takes an untested, open-source JavaScript jQuery plugin and brings it under test, demonstrating an approach to taming legacy code in a dynamic language. Both examples contain references to the detailed GitHub commit history behind the narrative.

Those Who Came Before

All advances build on the prior efforts of others. This work grew up in the context of a number of influences from the last fifteen years of computing. This list is not exhaustive, and I hope I do not offend those whom I missed or who did not receive as much publicity, but I would like to call out

- The influencers and signatories of the Agile Manifesto
- The pioneers of early Agile development approaches, such as Kent Beck with eXtreme Programming
- The Gang of Four and Martin Fowler for design patterns and refactoring
- The software craftsmanship movement and Robert C. "Uncle Bob" Martin
- More recent seminal work in software testing by Michael Feathers and Gerard Meszaros

I have had the good fortune to work with several of the colleagues of these luminaries from their formative teams.

Acknowledgments

All authors say that they could not have written their books without the support of their partners in life. I did not appreciate the full meaning of that until going through the process myself. I really could not have brought this book to fruition without the continual support and encouragement of my wife, Jenny, from her patience with the time investment to the daily statement of, "I'll clean up dinner. You go write."

I would like to thank Greg Doench for his patience with a first-time author and his experienced guidance through the editorial process. The review feedback of Zee Spencer, Eric E. Meyer, and Jerry Eshbaugh helped me focus and refine the material to its current form; I hope I did their feedback justice.

Trent Richardson, author of the jQuery Timepicker Addon used as the subject of the worked example in Chapter 15, has been wonderfully enthusiastic and supportive of the efforts to bring the project under test. He has accepted all pull requests to date. As of this writing, the first release with tests just went live.

I have had the opportunity to coach and manage several teams over the years. You do not really know something until you know how to teach it. You all know who you are. Thank you for being my reason to grow.

The authors of many articles in *Dr. Dobbs' Journal* and *Software Development* magazine stimulated my early thinking on test-early approaches. Eric E. Meyer, Jean-Pierre Lejacq, and Ken Faw showed me the disciplined ways of TDD. Many of my formative opportunities occurred with or through Christopher Beale. Our careers have intertwined, including our work with Joe Dionise; much of my early design and architecture experience was forged under their mentorship.

Finally, two professors have unknowingly had a strong influence that culminated in this book. Professor Lee, formerly of the University of Michigan in the early 1990s, showed me that computer science could be more than just a hobby. In my first class of my master's program, Professor Janusz Laski of Oakland University introduced me to formal verification and static and dynamic analysis methods, instrumental in my understanding and advocacy of tools to support the software-development process.

About the Author

Stephen Vance has been a professional software developer, consultant, manager, mentor, and instructor since 1992. He has practiced and taught code-level, automated testing techniques since 1997. He has worked across a broad range of industries for companies ranging from start-ups to Fortune 100 corporations. He has spoken at software conferences throughout the United States and Europe. Stephen lives with his wife near Boston, Massachusetts.

This page intentionally left blank

Chapter 1

Engineering, Craftsmanship, and First-Time Quality

Our industry largely considers the titles software developer, computer programmer, code-slinger, and software engineer synonymous. However, in some places the latter title cannot be used because of strict regulation of the term "engineer." Software aspires to be an engineering discipline along with the more traditional engineering disciplines such as civil, mechanical, electrical, and aerospace. It has started to be recognized as such in recent years, but with considerable debate over questions of licensure, certification, and the minimal requisite body of knowledge.

At the same time, the software craftsmanship movement has grown steadily. With software craftsmanship groups around the world and events like CodeMash[1] and Global Day of Code Retreat,[2] an increasing number of developers want to focus on the skilled creation of code.

1. http://codemash.org
2. http://globalday.coderetreat.org

Engineering and Craftsmanship

One of the things that differentiates software engineering from other engineering disciplines is that we regularly and fully practice all aspects of the software construction process. Other types of engineers generally have interest and skills in their fields' associated construction processes, but they rarely live them as a daily activity.

Automotive engineers may spend time on the assembly line, but they do not work it regularly. Civil engineers may supervise and inspect the building of bridges or buildings, but they spend little time driving rivets, pouring concrete, or stringing suspension cables. Probably the closest to software engineers' total immersion might be the handful of test pilots who are also aeronautical engineers, in that they participate in design, construction, inspection, and verification of the craft they fly.

As a result of this renaissance approach to software, we tend to blur the lines between craft, engineering, and creative endeavor. Add to that a healthy dose of participation in the problem domain of our software, and it is no wonder that we entangle the concerns of our work. But what do I mean by that?

Professional software practitioners code, design, architect, test, measure, and analyze on multiple levels. Several of these activities clearly constitute engineering. The design and validation of others are also clearly engineering. But no matter how you cut it, the act of coding is an act of craftsmanship, of doing. We may "engineer" concurrently with the motions of our fingers on the keyboard, but we still exercise a craft.

The Role of Craftsmanship in First-Time Quality

Craftsmanship connotes skill. The skill with which we code affects the outcome of what we do. No matter how much we architect, design, or conceive our software, a poor implementation can undermine it all.

Traditionally, we relied on verification by our own attention or that of a quality assurance group to manually make sure the software behaved correctly. A developer once said to me, "I write my code, then I train it to do the right thing." While I lauded the underlying attentiveness to ultimate correctness, that statement also communicates a lack of intention for or attention to the initial product. The punch line came when he spent a week changing some core code in a complicated system and asked for three weeks to test the result.

Lean manufacturing operates under a principle of building the quality into a product. Rework is a form of waste to be eliminated from the system.[3] Writing software so you can rewrite it or patch it once the bugs are found is rework.

Testing software that should be correct can be seen as wasteful. Given that we have not figured out how to create software without defects, some degree of post-coding testing is required. However, testing it more than once is clearly inefficient, which is what happens when a defect is found, fixed, and retested.

Another form of waste is inventory, which can be seen as opportunity cost. Time spent fixing bugs is time during which the correctly written parts of the software have not been delivered and are therefore not delivering customer or business value. That time is also time in which a developer could have been working on other valuable activities, including professional development.

Improving the craftsmanship aspect of your work pays you and your company back with multipliers. It allows you to deliver more value with less waste in less time and with greater opportunity for personal satisfaction.

A Word on Methodologies

I will say up front that I am a fan of Agile and Lean approaches to software development. However, I am also a firm believer that there is no one methodology or family of methodologies that works best for every team, company, project, or product. I have done fairly standard software projects using both Agile and Waterfall approaches. I have done Agile development embedded in an otherwise Waterfall process. I have even applied largely Agile techniques to products that involved real-time control and human safety concerns. I have also taken very aggressively Agile and Lean approaches to cutting-edge web development with an aim toward full organizational transformation.

Regardless of the methodology you use, you probably have some form of automated testing in your project, if for no other reason than that you are tired of pressing the buttons yourself. More likely, you have some form of continuous integration system with unit, integration, system, component, functional, and other forms of test running frequently.

And where there is test automation, someone is writing code that tests other code. If that is the case, you will find the techniques in this book useful. For unit or isolation testing, almost all of the techniques will apply. For larger scopes of testing, you may have fewer options to apply. Some techniques will only pertain

3. There are many references on Lean, the Toyota Production System, and waste (or *muda*) in that context. An accessible place to start is http://en.wikipedia.org/wiki/Toyota_Production_System.

to some languages. You can use others in almost any language or programming paradigm.

I mention approaches to testing that are common in the Agile playbook. Do not let that deter you from trying them if that is not your flavor of development. Whether you practice test-driven development, test first, test early, or test after, you still need to tame the code under test. Happy testing!

Practices Supporting Software Craftsmanship

The question then becomes: How can we constantly increase the likelihood that our software is correct from the start? First, let computers do what computers do best, the rote mechanical activities and detailed analyses that are time-consuming and error-prone for humans. The ways we can leverage automated code quality tools leads to a taxonomy of software hygiene.[4]

- Style: Whitespace, curly braces, indentation, etc.
- Syntax: Compilation, static analysis
- Simplicity: Cyclomatic complexity, coupling, YAGNI[5]
- Solution: Runtime, correctness, whether it works, TDD, BDD
- Scaling the effort: Continuous integration, continuous delivery

Starting with the integrated development environment (IDE),[6] we have a wealth of features at our disposal covering Style to Solution. Code completion, syntax highlighting, and project and file organization and management only start the list. Refactoring support and other tool integrations sweeten the pot. If you can, configure all of your tools to give immediate, real-time feedback within your IDE. The less time that passes between when you write the code and when you get the feedback, the closer you come to first-time quality.

Do not concern yourself with the placement of curly braces and the amount and type of whitespace. Configure your IDE or your build to

4. Thanks to Zee Spencer for his suggestions in this area.

5. "You aren't going to need it" from extreme programming. [XPI, p. 190]

6. Or your editor of choice. A number of text editors, such as Sublime Text (www.sublimetext.com), support many IDE-style features in a lighter-weight package.

do all of your formatting for you. Ideally, your IDE will do it so that it is applied instantly. The next best thing gives that responsibility to a build target that you can run locally at any time.

Further protect yourself with static analysis tools to help with Syntax and Simplicity. Tools like Checkstyle for Java, jshint or jslint for JavaScript, perlcritic for Perl, lint for C and C++, or the equivalent for your language of choice help to check for stylistic problems. More sophisticated tools like PMD[7] or FindBugs for Java go beyond formatting and style and explicitly look for everything from unused and uninitialized variables to framework-specific conventions and complexity metrics. Some are even extensible. PMD in particular has a very flexible capability to define rules based on XPath expressions against the abstract syntax tree. PMD also has a module called CPD, the copy–paste detector, that highlights code that was copied when perhaps it should have been refactored to reuse.

Code coverage tools help you guide your testing by numerically and graphically showing you what parts of your code have been executed, guiding you toward the right Solution. We will look more closely at code coverage ahead.

Code generation tools will create thousands of lines of code based either on your source or their own domain-specific language, potentially helping with Simplicity and Solution. Tools exist to generate network service interfaces, database access code, and more, even though in many cases the use of code generation is a code smell[8] of a missed opportunity for code reuse, also Simplicity. In some cases, you even have tools to help generate your tests. If you trust your code generator, you can skip testing the generated code, saving time and effort on two fronts.

Finally for tools, continuous integration can run all of these tools along with the build and the tests to provide qualification of your code independent of the local environment, helping with Scaling. The results of the continuous integration builds should be available via the web, on monitors, and through e-mail. Many groups have tied their

7. PMD is a static analysis tool for Java and other languages. It can be found at http://pmd.sourceforge.net. The developers claim that it is not an acronym, but a "backronym." See http://pmd.sourceforge.net/meaning.html for explanation and possible interpretations.

8. Another of Kent Beck's contributions to software craftsmanship, meaning a symptom of deeper problems.

failing continuous integration results to sirens or flashers to make it obvious. Continuous integration builds should provide results in minutes for best effect. Continuous integration provides yet another form of rapid feedback, trading a little bit of responsiveness for a more comprehensive evaluation.

Testing

Of all the practices you can leverage to assist your craftsmanship, you will get the most benefit from testing. You may say, "We have been testing software for years, and we still have lots of bugs." Therein lies the key to going forward.

In a Lean system, you try to prevent defects rather than catch them. The tradition of software testing has been one of catching defects. It should not surprise us, then, that we have an inefficient process. The earlier you test, the sooner you catch defects.

Does that mean you are still just catching defects? If you write the tests after the code, then yes. If you close the gap between creating and catching the defects, less change occurs between the two events. You have a higher likelihood of the context of the created code being fresh in your mind. If you write the test immediately after writing the code, it can verify your conceptions and assumptions and keep you on course.

You also have an opportunity to really catch the bugs before they occur, however. Test-first and test-driven approaches to development bring the testing before the coding. When you test first, you capture your intent in an automatable and executable form. You focus on what you are about to write in a way that works to prevent defects rather than create them. The tests you write serve as a persistent reinforcement of that intent going forward. In addition to helping you do the *thing right*, it helps you to do the *right thing*. Remember, it is still a bug if you implement the wrong thing really well.

Within this model, **test-driven development** (**TDD**) is a subset of test-first approaches. Test first allows you to write as many and as fully coded tests as you want prior to writing your code. Test first brings the quality forward, but it gives you greater latitude to speculate about what you need and the form it will take. If you designed a solution that is more general than you need, you run the risk not only of spending more time in the creation, but also of creating a larger maintenance burden than you need, a cost that carries forward indefinitely. Additionally, writing too many tests in advance increases the cost of change if you discover a better approach to the problem. For these reasons, TDD gives you a more disciplined and leaner approach to the creation of software. You create just what you need right as you need it. The discoveries you make about insufficient foresight and incomplete prior understanding guide the direction of your code.

The principles put forth in this book will help you test effectively, regardless of when you test relative to when you code, but the earlier you test, the more this book will help. One of the guidelines around Agile development and testing is the Automated Testing Triangle (Figure 1-1). Traditionally, we have focused on system-level testing, which tends to either be brittle if it verifies too deeply or cursory if it aims to be more maintainable. We will never be able to replace system-level testing, but we can shift the balance of tests to lower-level unit tests.

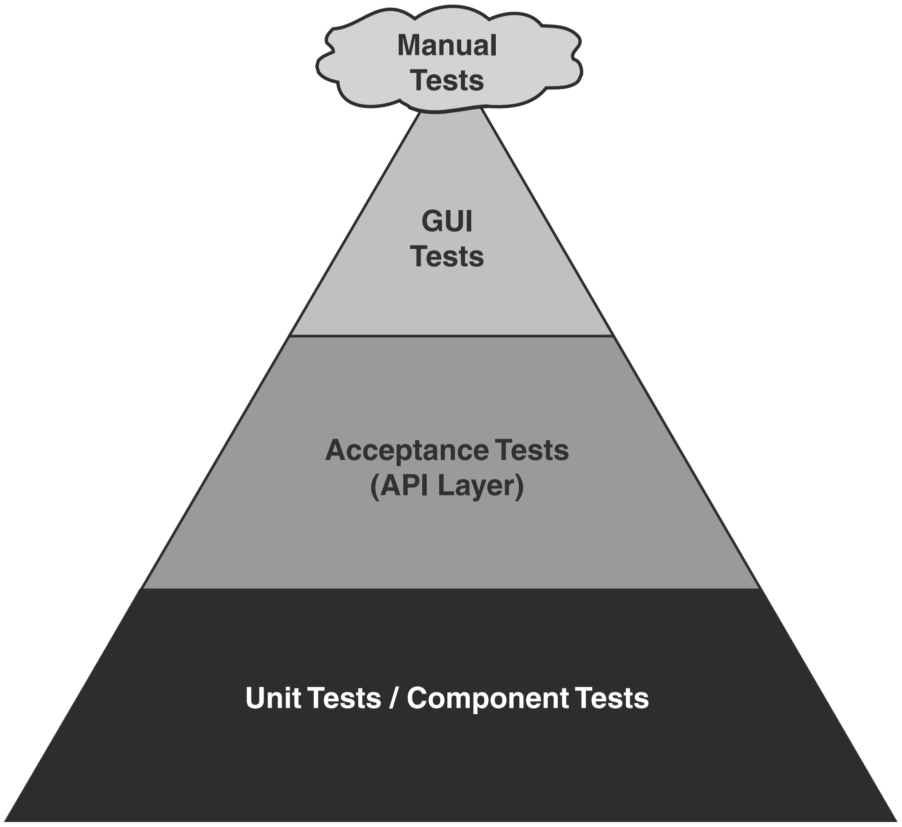


Figure 1-1: *The Automated Testing Triangle puts high investment in smaller tests because, when well written, they are less brittle and easier to maintain.*

I have heard people object to extensive unit tests on the grounds that they are brittle and hard to maintain. Well-written unit tests have exactly the opposite characteristics. The principles and techniques in this book will help you understand the reasons unit testing can become entangled and how to avoid them.

Unit Testing under Code Checker Constraints

Earlier, I mentioned that you should use tools to automate as many craftsmanship concerns as you can. If your tests drive the quality of your development, then you need to apply good craftsmanship to your tests as well. Not only should your tools check your application code, they should also check your test code to almost the same standards. I say "almost" because some reuse heuristics for production code need to be relaxed in test code. For example, repetition of string values between tests helps to keep your tests independent of each other, although using different strings is even better. Similarly, while refactoring for reuse benefits test code, you should not refactor tests in a way that blurs the distinction between setup, execution, and verification.

Applying static checker constraints to your unit tests will narrow the range of solutions you can apply. For example, a chain of default PMD rules, if left in effect, will push you to take many of your constant values outside of your test methods if you need to use them in nested classes. You can address this in a number of ways, but if it becomes too much of a burden with little value for your team, you may want to apply a subset of your normal rules to your test code, or even an equivalent but customized set.

Regardless, treat your test code to the same standards as your application code and it will treat you well far into the future. If the form of some of the examples in the book seem a little strangely implemented, try implementing them the way you think they should look with static checkers turned on, and you will probably understand why.

Unit Testing for Coverage

Code coverage tools provide wonderful assistance in writing, crafting, and maintaining your tests. They graphically show you which code you executed in your tests, letting you know if you hit the code you intended and which code you have not hit yet.

Code coverage by no means gives you an exhaustive understanding of how you have exercised your code. Many constructs branch to multiple code paths in ways that cannot be statically determined. Some of the most powerful features of our various languages—the features that give us the most useful abstractions—work in ways that are transparent to static analysis. While the coverage tool may show you what has been executed when you use these mechanisms, it has no way to know in advance how many permutations exist in order to compute the degree of coverage from execution.

Data-driven execution replaces chained conditionals with lookups, generally from an array or table. In Listing 1-1, the coverage tool will give an accurate assessment of statement coverage for this case, but it cannot tell you whether each of the conditions represented as the transition points in the table have been covered as it could if you implemented it with chained `if...else` statements. If the functions in the `rateComment` array were defined elsewhere and potentially reused, the coverage tool would miss even more.

Listing 1-1: *An example of data-driven execution in JavaScript demonstrating a blind spot of code coverage*

```
function commentOnInterestRate(rate) {
  var rateComment = [
    [-10.0, function() { throw "I'm ruined!!!"; }],
    [0.0,
      function() { gripe("I hope this passes quickly."); }],
    [3.0, function() { mumble("Hope for low inflation."); }],
    [6.0, function() { state("I can live with this."); }],
    [10.0,
      function() { proclaim("Retirement, here I come."); }],
    [100.0, function() { throw "Jackpot!"; }]
  ];

  for (var index = 0; index < rateComment.length; index++) {
    var candidateComment = rateComment[index];
    if (rate < candidateComment[0]) {
      candidateComment[1]();
      break;
    }
  }
}
```

Dynamic dispatch provides another mechanism by which code execution defies static analysis. Take the following simple line of JavaScript.

```
someObject[method]();
```

The value of the variable `method` is not known until runtime, making it impossible in the general case for a coverage tool to determine the number of available paths. In JavaScript, the number of methods can change over the course of execution, so the even the known methods of the object cannot be used to inform a coverage calculation. This problem is not restricted to dynamic languages. Even canonically statically typed languages such as C++, through virtual functions and pointer-to-member references, and Java, through reflection, have dynamic-dispatch features.

Other situations happen more naturally, what I call semantically handled edge cases. These are cases in which the language or runtime environment automatically translates exceptional conditions into variations that need to be handled differently from normal execution. Java unchecked exceptions, exceptions that do not have to be declared in the method signature, encapsulate a number of these, most famously the dreaded `NullPointerException` that occurs when trying to use a `null` reference. The handling of divide-by-zero errors across languages varies from full application crashes to catchable exceptions to the return of `NaN`[9] from the calculation.

Additionally, code coverage can deceive. Coverage only shows you the code that you executed, not the code you verified. *The usefulness of coverage is only as good as the tests that drive it.* Even well-intentioned developers can become complacent in the face of a coverage report. Here are some anecdotes of innocent mistakes from teams I have lead in the past that let you begin to imagine the abuse that can be intentionally wrought.

- A developer wrote the setup and execution phases of a test, then got distracted before going home for the weekend. Losing his context, he ran his build Monday morning and committed the code after verifying that he had achieved full coverage. Later inspection of the code revealed that he had committed tests that fully exercised the code under test but contained no assertions. The test achieved code coverage and passed, but the code contained bugs.

9. `NaN` is the symbolic representation of "not a number" from the IEEE floating-point specifications.

- A developer wrote a web controller that acted as the switchyard between pages in the application. Not knowing the destination page for a particular condition, this developer used the empty string as a placeholder and wrote a test that verified that the placeholder was returned as expected, giving passing tests with full coverage. Two months later, a user reported that the application returned to the home page under a certain obscure combination of conditions. Root cause analysis revealed that the empty string placeholder had never been replaced once the right page was defined. The empty string was concatenated to the domain and the context path for the application, redirecting to the home page.
- A developer who had recently discovered and fallen under the spell of mocks wrote a test. The developer inadvertently mocked the code under test, thus executing a passing test. Incidental use of the code under test from other tests resulted in some coverage of the code in question. This particular system did not have full coverage. Later inspection of tests while trying to meaningfully increase the coverage discovered the gaffe, and a test was written that executed the code under test instead of only the mock.

Code coverage is a guide, not a goal. Coverage helps you write the right tests to exercise the syntactic execution paths of your code. Your brain still needs to be engaged. Similarly, the quality of the tests you write depends on the skill and attention you apply to the task of writing them. Coverage has little power to detect accidentally or deliberately shoddy tests.

Notice that at this point I have not talked about which coverage metric to use. Almost everyone thinks of statement or line coverage. Statement coverage is your entry point, your table stakes, provided by almost all coverage tools. Unfortunately, many stop there, sometimes supplementing it with the even weaker class and method/function coverage. I prefer minimally to use branch and condition coverage[10] as well. Several tools include branch coverage. Woefully few include condition coverage and beyond. Some additional metrics include loop coverage—each loop must be exercised zero, one,

10. Branch coverage evaluates whether each option in the syntactic flow control statements is covered. Condition coverage looks at whether the full effective truth table after short-circuit evaluation is exercised for complex conditions.

and many times—and data path metrics such as def-use chains[11] [LAS83, KOR85]. In Java, the open-source tool CodeCover (http://codecover.org) and Atlassian's commercial tool Clover do well. Perl's Devel::Cover handles multiple metrics as well. Although its messages could use some improvement, PMD includes dataflow analysis errors and warnings for UR, DU, and DD anomalies.[12]

I seem to have an affinity for high-availability, high-reliability, and safety-critical software. I have led and worked on teams developing emergency-response software, real-time robotic control (sometimes in conjunction with non-eye-safe lasers), and high-utilization build and test systems for which downtime means real business delays and losses. I have led projects in which we treated 100% statement, branch, condition, and loop coverage as only a milestone to thorough *unit* testing. Not only did we only derive coverage from unit tests—as opposed to the many other levels of testing we applied to the systems—but we only counted coverage obtained for a class by the test for that class. Incidental coverage by use from other classes was not counted.

In general, I have found that you start to get a quality return at around 50% statement coverage. The return becomes meaningful as you approach 80%. You can get significant return as you pass the milestone of 100%, but the cost of doing so depends on your skill at testing and writing testable, low-complexity, loosely coupled code.[13] Whether the cost justifies your return depends on your problem domain, but most teams do not have the experience in achieving it to accurately assess the tradeoff.

11. Definition-use, define-use, or def-use chains are the paths from the definition of a variable to the uses of that variable without an intervening redefinition. See also http://en.wikipedia.org/wiki/Use-define_chain for the opposite analysis of a use of the variable and all the paths from definitions of that variable to the use without intervening redefinitions. The set of paths is the same for the two metrics, but the grouping is based on opposite end points. Coverage of these paths is a stricter form of coverage than is implemented in most tools.

12. See http://pmd.sourceforge.net/pmd-4.2.5/rules/controversial.html#DataflowAnomalyAnalysis.

13. These milestones are purely anecdotal, but correlate well with the observations of others, including http://brett.maytom.net/2010/08/08/unrealistic-100-code-coverage-with-unit-tests/ and the common targets of 70–80% coverage in many organizations. There are several possible explanations for this effect, ranging from the fact of the tests to the increased focus on design from practices like TDD.

Typically, teams choose an arbitrary number less than 100% based on arguments that it is not worth it to reach 100%. Generally, arguments for what not to test fall into two groups: the trivial and the difficult. Arguments against writing the difficult tests focus on either the algorithmically complex items or the error paths.

The trivial includes things like simple getters and setters. Yes, they can be boring to test, but testing them takes little time and you will never need to wonder if your coverage gap is only due to the trivial omissions.

The algorithmically complex code is most likely the heart of your secret sauce—the thing that distinguishes your software from everyone else's. That sounds to me like something that requires testing. If the implementation complexity discourages testing, it probably needs design improvements, which can be driven by the need for testability.

The error path tests verify the parts most likely to upset your customers. You rarely see kudos in online reviews for software that does not crash and that handles errors gracefully. Software that crashes, loses data, and otherwise fails badly gets poor and very public reviews. In our eternal optimism, we developers hope and almost assume that the error paths will rarely be traversed, but the reality that the world is perfectly imperfect guarantees that they will. Testing the error paths invests in your customers' good will under adversity.

Ultimately, you make the decision about the degree of testing your business needs. I recommend that you make that decision from the position of the skilled craftsman who can achieve whatever coverage the business requires rather than from the position of avoiding high coverage because it seems too hard. The purpose of this book is to fill out your tool belt with the patterns, principles, and techniques to make that possible.

This page intentionally left blank

Index