



Quality Code

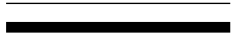
Software Testing Principles,
Practices, and Patterns

Stephen Vance

FREE SAMPLE CHAPTER



SHARE WITH OTHERS



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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

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

Copyright © 2014 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

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 자그/야
for your continuous love and support.*

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

This page intentionally left blank

Contents

Preface	xiii
Acknowledgments	xvii
About the Author	xix
Part I: Principles and Practices of Testing	1
Chapter 1: Engineering, Craftsmanship, and First-Time Quality	3
Engineering and Craftsmanship	4
The Role of Craftsmanship in First-Time Quality	4
Practices Supporting Software Craftsmanship	6
Testing	8
Unit Testing under Code Checker Constraints	10
Unit Testing for Coverage	10
Chapter 2: Intent of Code	17
Where Did I Put That Intent?	18
Separating Intent from Implementation	18
A Simple Example That Makes You Think	19
Chapter 3: Where Do I Start?	23
An Approach to Testing	23
Understand Your Scope	24
A Conceptual Framework for Testing	25
State and Behavioral Testing	26
To Test or Not to Test	27

The Recipe	28
Test the “Happy Path”	29
Test the Alternate Paths	29
Test the Error Paths	30
Test the Data Permutations	30
Test the Defects	35
Chapter 4: Design and Testability	37
A Word on Design Paradigms	37
Encapsulation and Observability	38
Representational Encapsulation	38
Behavioral Encapsulation	39
Shades of Testing	40
Encapsulation, Observability and Testability	42
Coupling and Testability	42
Chapter 5: Testing Principles	47
Craft Your Tests Well	47
Couple Inputs to Outputs	48
Use a Naming Convention	49
Avoid Test Code in Production	51
Verify Intent over Implementation	52
Minimize Coupling	53
Prefer Minimal, Fresh, Transient Fixtures	54
Use Available Facilities	55
Prefer Complete over Partial Verification	55
Write Small Tests	55
Separate Your Concerns	56
Use Unique Values	57
Keep It Simple: Remove Code	58
Don’t Test the Framework	58
Don’t Test Generated Code	59
Sometimes Test the Framework	60

Part II: Testing and Testability Patterns	61
Chapter 6: The Basics	63
Bootstrapping Constructors	63
Testing Simple Getters and Setters	66
Share Constants	67
Locally Redefine	70
Temporarily Replace	71
Encapsulate and Override	72
Adjust Visibility	75
Verification by Injection	77
Chapter 7: String Handling	81
Verification by Containment	81
Verification by Pattern	83
Exact Verification by Value	85
Exact Verification with Formatted Results	88
Chapter 8: Encapsulation and Override Variations	91
Data Injection	91
Encapsulate Loop Conditions	94
Error Injection	96
Replace Collaborators	98
Use Existing No-Op Classes	101
Chapter 9: Adjusting Visibility	105
Packaging Tests with Code	105
Break It Down	108
Changing Access Levels	109
Test-Only Interfaces	111
Naming the Unnamed	112
Becoming <i>friendly</i>	113
Coerced Access via Reflection	114
Declarative Scope Changing	116

Chapter 10: Interlude: Revisiting Intent	119
Testing the Singleton Pattern	120
Singleton Intent	121
The Testing Strategy	121
Testing the Singleton Nature	122
Testing the Class Purpose	123
Discerning Intent	127
Chapter 11: Error Condition Verification	129
Check the Return Value	129
Verify the Exception Type	130
Verify the Exception Message	132
Verify the Exception Payload	134
Verify the Exception Instance	137
Thoughts on Exception Design	140
Chapter 12: Use Existing Seams	145
Direct Calls	146
Interfaces	146
Implementations	146
Dependency Injection	147
Callbacks, Observers, Listeners, and Notifiers	150
Registries	154
Factories	156
Logging and Other Facilities of Last Resort	159
Chapter 13: Parallelism	165
A Brief Introduction to Threads and Race Conditions	166
Some History	166
Race Conditions	167
Deadlocks	169
A Strategy for Race Condition Reproduction	170
Test the Thread Task Directly	173
Synchronize through Common Lock	176

Synchronize through Injection	181
Example: Injecting Synchronization through Logging	181
Use Supervisory Control	184
Statistical Verification	187
Debugger APIs	189
Part III: Worked Examples	193
Chapter 14: Test-Driven Java	195
Bootstrapping	196
First Functionality	197
Cutting the Cord	198
Moving to Multiples	199
Ghost Protocol	200
Dead End	200
Spy Craft	201
Exercising Options	203
Moving Downstream	204
Retrospective	207
Chapter 15: Legacy JavaScript	209
Getting Started	210
DOMination	211
On Toothpaste and Testing	213
Scaling Up	215
Software Archeology	217
Retrospective	218
Bibliography	219
Index	221

This page intentionally left blank

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.

This page intentionally left blank

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 CodeMash¹ 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, YAGNI⁵
- 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 PMD⁷ 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 smell⁸ 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.

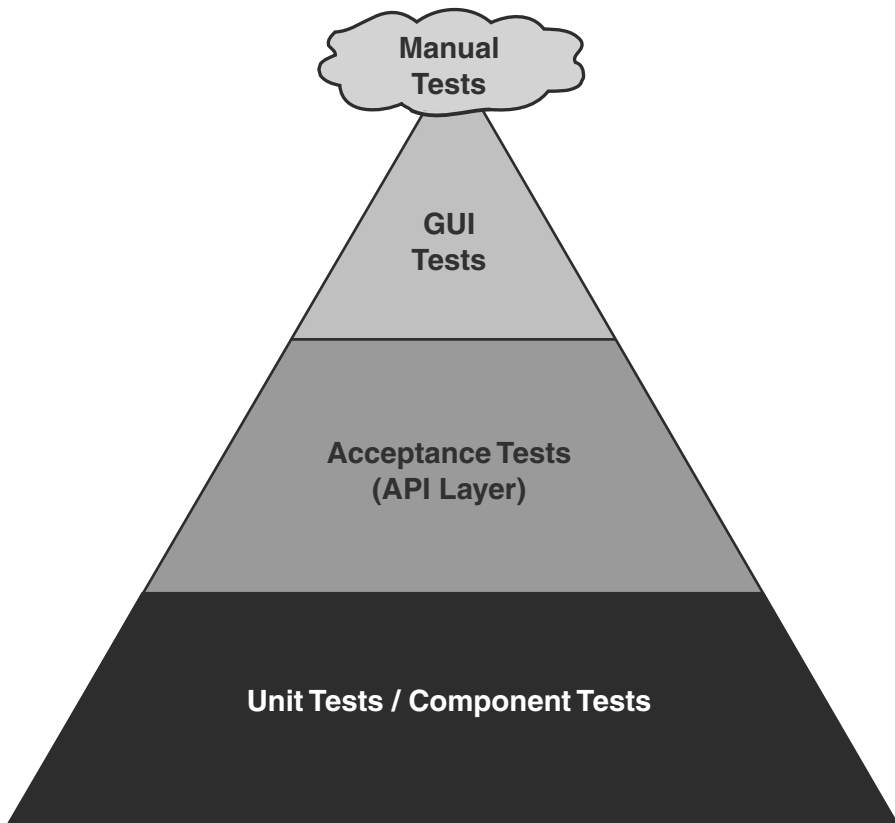


Figure 1-1: *The Automated Testing Triangle puts high investment in smaller tests because, when well written, they are less brittle and easier to maintain.*

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 coverage¹⁰ 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 chains¹¹ [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

Symbols

- . (dot) character, regular expressions, 84
- / (slash) character, regular expressions, 84
- \$ (dollar sign), regular expressions, 84

A

- AAA (arrange, act, and assert) test structure, 48, 66
- Abstract Factory pattern
 - overview of, 156–159
 - testing data-driven execution, 33
- Acceptance tests, 9, 41, 200
- Access levels
 - adjusting visibility by changing, 109–111
 - adjusting visibility by packaging tests, 105–107
 - designating with test-only interfaces, 111–112
 - using reflection to coerce access, 114–115, 126
- AccessibleObject class, 115
- Agile development
 - Automated Testing Triangle, 9
 - overview of, 5–6
- Alternate path tests, 29–31
- Annotations
 - simplifying error condition tests with, 131
 - in test naming conventions, 49–50
- Anonymous entities, difficult to test, 113
- Anonymous inner classes, Java
 - adjusting visibility, 111–112
 - in error injection, 98
- API specification, testing interfaces using, 146
- Appender
 - testing code using custom logging, 161–163
 - in thread-synchronization, 182–184
- Approach to testing, 41
- Architectural layers, influencing scope, 24
- Assertion failures, using unique values, 57
- Attribute encapsulation
 - accessing with getter methods, 64
 - resulting in coupling, 53
 - testing getters and setters, 66–67

Attributes

- adjusting visibility, 105
- encapsulation resulting in coupling, 45–46
- in exception design, 141
- exception-based, 134–135, 137–138, 141
- hiding for security, 55
- in naming conventions, 49
- in state verification tests, 27
- tests that pass when lying about, 19
- tests verifying individual, 45
- Authentication, expiration boundary test for, 72–75
- Automated testing. *See* Starting Automated Testing Triangle, 9, 28
- Automotive engineers, 4

B

- Basics, testing
 - adjusting visibility, 75–77
 - bootstrapping constructors, 63–66
 - encapsulating and overriding, 72–75
 - locally redefine, 70–71
 - sharing constants, 67–70
 - temporarily replace, 71–72
 - testing simple getters and setters, 66–67
 - verification by injection, 77–79
- Behavioral encapsulation, 39–40
- Behavioral testing
 - of alternate paths, 29–30
 - clarifying use of, 26–27
 - JavaScript example of, 27
 - purpose of regression test, 41
 - of Singleton nature, 122–123
- Black-box tests
 - coupling and, 42–43
 - defined, 40–41
 - testing transparency, 41
- BlockingQueue interface, 185
- Bootstrapping
 - constructors, 63–66
 - test-driven Java, 196–197
 - testing simple getters and setters, 66–67
- Bottom-up testing, 213
- Boundary conditions test, 31–32
- Branch coverage, 13–14

- Buffered I/O classes, 39
- Bugs
 - avoiding test code in production, 52
 - code coverage and, 12
 - eliminating rework of, 5
 - minimalism for less, 18, 122
 - preventing rather than catching, 8
 - race-condition, 165, 167, 185
 - in threads, 167
- C**
- C
 - dynamic-binding and reflection, 34
 - function pointers, 151
 - implementing callbacks, 151
 - printf format strings, 88–89
 - static analysis tool for, 7
 - verifying interface definition, 146
- C++
 - access level changes, 109
 - context mutability, 21
 - declaring entities in nested classes, 112
 - dynamic-binding and reflection, 34
 - error condition verification, 134
 - error condition verification in, 137
 - friend keyword, 113–114, 126
 - implementing callbacks, 151–153
 - nested anonymous enumerations, 70
 - pointer-to-member function, 151, 153
 - static analysis tool for, 7
 - verifying interface definition, 146
- Callable interface, Java
 - in common lock synchronization, 177–178, 180
 - directly testing thread task, 173
- Callbacks, 150–154
- Catch clause, verifying exception type, 131–132
- Characterization testing
 - legacy code and economics of, 28
 - mess of code requiring, 25
 - scaling up legacy code, 215
- Checkstyle, 7
- Chunking, 127
- Civil engineers, 4
- Clarity of intention, in minimalist approaches, 18
- Class
 - refactoring to enhance testability, 108–109
 - testing Singleton’s purpose, 123–127
- Class::Std module, Perl, 109, 116–117
- Cleanliness of code, and scope of testing, 24–25
- Clover, 14
- Code
 - avoid testing the framework, 58–60
 - simplifying tests by deleting, 58
 - testing untested. *See* Legacy JavaScript thread-safe, 167
- Code checker constraints, in unit testing, 10
- Code coverage
 - avoid testing code in production with, 52
 - blind spots, 11–12
 - deceptions of, 12–13
 - defined, 10–11
 - as guide, 13
 - metrics, 13–15
 - removing code in testing to increase, 58
 - unit testing with, 7
- Code generation tools, 7, 59–60
- Code smell
 - concurrency, 169
 - defined, 7
 - encountering through your tests, 128
- CodeCover, 14
- CodeMash, 3
- Coincidental equality, 73, 82, 86
- Coincidental similarity, 82
- Collaborators
 - defined, 77, 98
 - injecting into tests, 148–149
 - legacy code and, 215–217
 - replacing, 98–100
 - Test Doubles and, 100
- Command-line debuggers, 190
- Comments, relaxing visibility with, 111
- Common lock synchronization, 176–181
- Complete verification, testing for, 55
- Component level
 - cost of integration tests at, 28
 - testing purpose of code at, 25
- Conceptual framework, for testing, 25–26
- Concurrency
 - code smell, 169
 - monkey test for, 189
 - race conditions in, 185, 188
- Condition coverage, 13–14
- ConnectException, 133, 134
- Connection collaborator, 99–100, 102
- Constants
 - exact verification of strings with shared, 85–87
 - exact verification with formatted results using, 88–89
 - limitations of, 88
 - testing constructors using shared, 67–70
- Constructor injection, 148–149

- Constructors
 - bootstrapping, 63–66, 196–197
 - test-driven Java, 196–197, 200
 - testing Singleton with relaxed access to, 125–127
 - testing using shared constants, 67–70
 - Containment
 - false-positive weaknesses of, 88
 - verification of strings by, 81–83
 - Context, differentiating exceptions, 137–138
 - Continuous integration, in software craftsmanship, 7–8
 - Cooperative multitasking, in parallel programming, 166
 - Coupling
 - concept of, 53
 - defined, 42
 - factories as test seams and, 159
 - inputs to outputs in tests, 48–49
 - less likely with small tests, 55–56
 - minimizing in software, 42–46
 - mocks introducing high degree of, 201
 - null object implementations and, 103
 - reducing test, 53–54
 - Coverage
 - metrics, 13–14
 - testing alternate paths, 30
 - unit testing for, 10–15
 - Craftsmanship
 - practices supporting software, 6–8
 - principles for testing, 47–51
 - role in first-time quality, 3–5
 - software engineering and, 2–3
 - testing and, 8–10
 - unit testing for coverage, 10–15
 - unit testing under code checker constraints, 10
 - Critical section of code, read-write race conditions, 168
 - Cucumber tool, 200
 - curl tool. *See* Test-driven Java
 - Custom logging appender, 161–163
- D**
- Data path coverage, 14
 - Data permutations tests
 - boundary conditions, 31–32
 - data-driven execution, 32–33
 - overview of, 30–31
 - run-time and dynamic binding, 33–35
 - Data type injection, overriding encapsulation, 91–94
 - Data-driven execution
 - testing for permutations, 32–33
 - unit testing for coverage, 11
 - DataProvider parameter, @Test, 93–94
 - Deadlocks, 169
 - Debugger
 - APIs, 189–191
 - Java code using log4j as seam for, 160–163
 - reproducing race conditions, 171–172
 - Declarations, runtime exceptions, 134
 - Declarative scope changing, adjusting visibility, 116–117
 - Defects, testing for, 35
 - Def-use chains, 14
 - Deleting code, 58
 - Delimiters, naming conventions, 50
 - Depended-On Components (DOCs), 77
 - Dependencies. *See* Coupling
 - Dependency injection
 - as alternative to Singleton pattern, 122
 - overview of, 147–150
 - registries providing indirect form of, 155
 - synchronization through, 181–184
 - test-driven Java, 200
 - testing by, 77–79
 - Design and testability
 - about design paradigms, 37–38
 - behavioral encapsulation, 39–40
 - coupling, 42–46
 - dimensions of testing, 41
 - encapsulation, observability and, 42
 - overview of, 37
 - representational encapsulation, 38–39
 - shades of testing, 40–41
 - Devel::Cover, 14
 - Difficult-to-test designs, 122
 - Dimensions of testing, 41
 - Direct calls, using existing seams, 146–147
 - Direct testing, adjusting visibility, 75–77
 - DNS (Domain Name System), 32
 - DOCs (Depended-On Components), 77
 - Documentation, relaxing visibility with, 111
 - Dollar sign (\$), regular expressions, 84
 - The DOM, 211–212
 - Domain Name System (DNS), 32
 - Dot (.) character, regular expressions, 84
 - Dp4j tool, Java reflection, 115
 - Dynamic binding, reflection-based code, 33–35
 - Dynamic dispatch, unit test for coverage, 12

E

EasyMock

- in dependency injection, 149
- overview of, 100
- registries and, 155

Economics of testing types, 28

Encapsulation

- behavioral, 39–40
- breaking via direct testing, 75–77
- coupling in attribute, 45–46
- from coupling in unit testing, 53
- design paradigms and, 37
- observability, testing and, 42
- representational, 38–39

Encapsulation and override, 72–75

Encapsulation and override variations

- data injection, 91–94
- encapsulating loop conditions, 94–96
- error injection, 96–98
- overview of, 91
- replacing collaborators, 98–100
- using existing no-op classes, 101–103

Engineering and craftsmanship

- engineering, 2–3
- first-time quality and, 3–5
- practices supporting craftsmanship, 5–10
- software, 2–3
- testing and, 8–10
- unit testing for coverage, 10–15
- unit testing under code checker constraints, 10

Errno values, verifying error conditions, 129–130

Error condition verification

- checking return value, 129–130
- exception design, 140–143
- overview of, 129
- verifying exception instance, 137–140
- verifying exception message, 132–134
- verifying exception payload, 134–137
- verifying exception type, 130–132

Error conditions, null values indicating, 101

Error injection, testing using overrides for, 96–98

Error path testing

- for boundary conditions, 31
- scaling up legacy code, 215–216
- software archeology and legacy code, 217
- testing, 30

Errors, in Java, 134

Event objects, coupling in, 44

Exceptions

- error injection for testing, 96–98
- indicating with existing no-op classes, 101–103

injecting by replacing collaborators, 99–100

thoughts on designing, 140–143

verifying instance, 137–140

verifying message, 132–134

verifying payload, 134–137

verifying type, 130–132

Executor, Java packages, 173

Expiration boundary tests, 72–75

Explicit intent of development, testing implementation to match, 17

Externally triggered loop termination, encapsulation of, 94–95

Extract Class refactoring, 39–40

Extract Method refactoring, 39–40, 74, 113, 138

F

Facilities, testing with available, 55

Factories, 156

Factory Method and Builder, 156–159

Fail () assertion, 131–132

Failure paths, testing. *See* Error condition verification

Fakes, 100

Feathers, Michael, 2, 25–26, 28, 145, 209

Field length limits, testing for boundary conditions in, 31

FindBugs, 7

First-time quality, and craftsmanship, 3–5, 6–10

Fixture, 54, 123–126

Format string constants, exception design, 141–143

Formatted results, for exact verification of strings, 88–89

Four-Phase Test, 48, 66

Fowler, Martin, 101, 147

Fragile Tests, 138, 184

Frameworks

avoiding testing of, 58–59

inversion of control, 147–150

sometimes testing, 60

Fresh fixtures, 54

Friend keyword

adjusting visibility with, 113–114

testing Singleton with, 126

Full stack tests

of error paths, 30

of happy path, 29

overview of, 24

Functionality

to test or not to test, 27–28

testing code at module/component level, 25

- testing happy path for primary, 29
 - testing separately, 56–57
 - testing Singleton’s class purpose, 123–127
 - unit testing of value added by code, 26
- Functions, locally redefining within test, 70–71
- The Fundamental Theorem of Software Engineering, 32–33
- G**
- Garbage-collected teardown
 - in common lock synchronization, 179
 - testing simple getters and setters, 66–67
- Gdb commands, 190–191
- Generated code, avoid testing of, 59–60
- GetInstance() method
 - in common lock synchronization, 179–180
 - testing Singleton’s behaviors, 122–123
- Getters
 - constructor bootstrapping model, 65
 - naming conventions for, 75
 - testing simple, 66–67
 - verifying constructor state with, 64
- GitHub, worked examples in, 193–194
- Global Day of Code Retreat, 10–15
- Granularity, testing, 41
- Gray-box tests
 - coupling and, 42–43
 - defined, 40–41
 - testing transparency, 41
- H**
- Hamcrest matchers, 67
- Happy path testing
 - for primary purpose of code, 29
 - scaling up legacy code, 215–216
 - for test-driven Java, 197–198
- Hierarchical locking, 169
- HttpClient library, 197–198
- Hypothesis, race conditions, 171–172
- I**
- IDE (Integrated development environment), software craftsmanship, 6–7
- IllegalArgumentException, Java, 134–136
- Immediately invoked function expression (IIFE), 210–211
- Implementation tests, scaling up legacy code, 216
- Implementations
 - direct testing of refactored, 39–40
 - separating intent from, 18–22
 - shades of testing for, 40–41
 - testing code using, 146–147
 - using factories as seams, coupling tests to, 159
 - using registries as seams, 155–156
 - verifying intent over, 52
- Implicit intent of development, testing
 - implementation to match, 17
- Infinite loops, encapsulation of, 95
- Infinite Monkey Theorem, 189
- Initialization, and factory seams, 156–159
- Injection, verification by, 77–79
- Injection points, as seams, 181
- Inputs, coupling test outputs to test, 48–49
- Instance, testing Singleton with constraints on, 124–125
- Instance, verifying exception, 137–140
- Instruction-level write-write race conditions, 168
- Integrated development environment (IDE), software craftsmanship, 6–7
- Integration tests, 25, 28
- Intent of code
 - crafting tests for, 47–48
 - for difficult-to-test designs, 122
 - discerning, 127–128
 - example of, 19–22
 - exception design and, 141–143
 - overview of, 17, 119–120
 - separating implementation from, 18–19
 - slipperiness of, 18
 - testing Singleton pattern, 120–121
 - testing strategy, 121–127
 - verifying over implementation, 52
- Intention Checker, 17
- Interface-based design, 100
- Interfaces
 - direct testing of thread task, 173
 - implementing callbacks through, 151
 - testing code using, 146
- Internationalization error messages, exception design, 141
- Inter-Process Communications (IPC) mechanisms, 166–167
- InterruptedException, Java, 140
- Inventory, as waste in Lean manufacturing, 5
- IPC (Inter-Process Communications) mechanisms, 166–167
- Isolation tests, 41
- J**
- Jasmine test framework
 - scaling up legacy code, 215–216
 - temporarily replacing methods in, 72
 - testing DOM set up using, 212
- Java
 - adjusting visibility, 105–109, 111–112, 114–115
 - breaking encapsulation, 75–76

- Java, *continued*
 - error condition verification, 134
 - exception design, 140
 - implementing callbacks through interfaces, 151
 - logging seam, 160–163
 - packages, 76–77, 105–107
 - Singleton pattern, 120
 - synchronization, 179
 - test-driven example of. *See* Test-driven Java
 - verifying interface definition, 146
 - Java Concurrency in Practice*, 166
 - Java.net .URI class, 201–203
 - JavaScript
 - callback functions, 151
 - de facto interface of, 100
 - declaring functions in any scope, 112
 - function pointers, 151
 - implementing callbacks, 151
 - testing UI component, 193–194
 - write-read race condition, 167
 - JavaScript, legacy
 - getting started, 210–211
 - overview of, 209–210
 - retrospective, 218
 - scaling up, 215–217
 - software archeology, 217
 - on toothpaste and testing, 213–214
 - Java.util.concurrent packages, Java 5, 173
 - JMock, 100–101
 - jQuery Timepicker Addon project
 - getting started, 210–211
 - overview of, 209–210
 - retrospective, 218
 - scaling up, 215–217
 - software archeology, 217
 - on toothpaste and testing, 213–214
 - Jshint, 7
 - Jslint, 7
 - JUnit
 - testing simple getters and setters, 67
 - TestNG vs., 93
 - verifying exception message in, 133
 - verifying exception type in, 131
- L**
- Law of Demeter, 46
 - Lean production
 - building quality into product, 4
 - preventing vs. catching defects in, 8
 - rework as waste in, 5
 - as testing approach, 5–6
 - Legacy code, 209
 - Legacy JavaScript
 - getting started, 210–211
 - overview of, 209–210
 - retrospective, 218
 - scaling up, 215–217
 - software archeology, 217
 - on toothpaste and testing, 213–214
- Listeners, 150–154
- Localized error messages, exception design, 141–143
- Log4j, 160–161, 182–183
- Logging
 - injecting synchronization through, 181–184
 - testing code using, 159–162
- Lookup key, dependency injection in registries, 155
- Loop conditions, encapsulation of, 94–96
- Loop coverage, 13–14
- M**
- Main() method, testing, 196–197
 - Manual tests
 - Automated Testing Triangle, 9
 - test-driven Java, 200
 - MapReduce frameworks, 174
 - Martin, Robert C., 2, 101
 - Mathematical functions
 - state verification tests of, 26–27
 - testing for boundary conditions in, 31–32
 - Mathematical view of tests, 64–66
 - Maturity of code, and scope of testing, 24–25
 - Messages, exception
 - design of, 141
 - verifying, 132–134
 - Meszaros, Gerard, 2, 24, 37, 54, 63, 77, 100
 - Methodologies, Agile and Lean testing, 5–6
 - Methods
 - adjusting visibility by changing access levels, 109–111
 - refactoring to enhance testability, 108–109
 - replacing temporarily within test, 71–72
 - Meyers, Scott, 113
 - Milestones, for coverage metrics, 14–15
 - Minimal fixtures, 54
 - Minimalism, applied to software, 18
 - Mockito, 100–101
 - Mocks
 - creating test doubles, 149
 - danger of testing with, 201
 - overriding methods, 101
 - overview of, 100
 - using registries as seams, 155

- Module level
 - integration tests at, 28
 - testing purpose of code at, 25
- Monitoring results, reproducing race conditions, 172–173
- Monkey testing, 41, 189
- Monte Carlo testing, 41, 190
- Mutability, exception, 137
- N**
- Naming conventions
 - for designating exposed functions as private, 210–211
 - exception design, 141–143
 - getters, 75
 - for specific use of callbacks, 154
 - for tests, 49–51
- Nested anonymous enumerations, 70
- Notifiers, as specialized callback, 154
- Null Object pattern, 101–103
- `NullPointerException`, unit testing, 12
- Numerical functions, testing for boundary conditions, 31–32
- O**
- Object-oriented (OO) design
 - as dominant design paradigm, 37–38
 - encapsulation and observability in, 38–42
 - implementation hiding in, 38
- Observer pattern, 154
- Operations, behavioral verification tests of, 27
- OU (organizational unit), LDAP, 154
- Outputs, coupling test inputs to test, 48–49
- Override
 - encapsulation and. *See* Encapsulation and override
 - injection, 93
- P**
- Package declaration, `Class::Std`, 116
- Packages
 - elevating access to code being tested, 105–107
 - in Java, 76–77
- Parallelism
 - deadlocks, 169
 - debugger APIs, 189–191
 - history of, 166
 - overview of, 165–166
 - race conditions, 167–169
 - reproducing race conditions, 170–173
 - statistical verification, 187–189
 - supervisory control, 184–187
 - synchronizing through common lock, 176–181
 - synchronizing through injection, 181–184
 - testing thread task directly, 173–175
 - threads, 166
- Partial verification, testing for, 55
- Pattern, verification of strings by, 83–85
- Pattern Languages of Program Design*, 101
- Payload, verifying exception, 134–137
- Performance tests, 41
- Perl
 - checking stylistic problems, 7
 - `Class::Std` module, 109
 - function pointers, 151
 - package declarations, 107
 - testing Singleton, 126–127
 - verifying strings by pattern, 83–85
- Perlcritic, 7
- Plug-ins, avoiding testing of, 58–59
- PMD
 - code coverage metrics, 14
 - static analysis tool, 7
- POSIX
 - loading and invoking functions as callbacks, 151–152
 - runtime binding without error handling, 34–35
 - verifying error conditions, 129–130
- PowerMock, 115
- Preallocated exceptions, verifying exception instance, 138–139
- Preemptive multitasking, 166
- Principle of Least Knowledge, minimizing coupling, 46
- Principles, testing. *See* Testing principles
- Private functions, legacy JavaScript, 210–211
- Private methods
 - adjusting visibility, 75
 - changing access levels, 109–110
 - coercing test access with reflection, 114–115
 - debate on testing, 40
 - as implementation details, 110
 - simplifying public interface with, 108
- `PRIVATE` trait, `Class::Std`, 116
- Privateer, 115
- Privileges, debugger, 190
- Processes, in parallel programming, 166
- Production, avoiding test code in, 51–52
- `Protected` method, changing access levels, 109–111
- `Public` method, changing access levels, 109–111
- Purpose
 - as dimension of testing, 41
 - testing based on, 25–26
 - testing happy path for primary, 29
- Python modules, 106

R

Race conditions
 in common lock synchronization,
 176–178, 180
 deadlocks vs., 169
 identifying, 170–171
 isolating using gdb commands, 190–191
 with logging seam, 181–184
 reproducing, 171–173
 statistical verification of, 187–189
 threads and, 167–169
 with thread-safe components, 185–186

Random event testing, 41

Read-write race conditions, 167

Recipe, testing. *See* Testing recipe

Redefining variable or function, local tests,
 70–71

Redundant tests, removing, 58

Refactoring
 behavioral encapsulation and, 39–40
 for better testability, 86–87
 of classes, 108–109
 expiration boundary test with, 72–75
 to provide seam for test-driven Java,
 198–199
 software archeology and legacy code, 217
 for testable loop control, 96
 testing thread task directly, 175
 toothpaste, 213

References, storing in newer registries, 154–155

Reflection
 coerced access via, 114–115
 overview of, 33–34
 testing Singleton, 126

Registration management, and factory seams,
 157–159

Registries, as seams for testing, 154–156

Regression tests, 41

Regular expressions, 84–85, 88

RemoteException, 130–134

Replace methods, temporarily within test,
 71–72

Representational encapsulation, 38–39

Reproduction, race conditions, 170–173,
 177–179

Resource constraints, testing for boundary
 conditions, 31–32

RESTRICTED trait, Class:Std, 116

Reverse engineering, bottom-up testing, 213

Rework, as waste in Lean, 4

Risk assessment, testing third-party code, 60

Ruby, packages in, 106

Runnable interface, Java thread task, 173–175

Run-time, testing for data permutations at,
 33–35

RuntimeExceptions, 134

S

Scaling, 6–8

Scope
 changing declarative, 116–117
 scaling up for legacy code, 215–217
 small tests for limiting, 55–56
 testing error paths and, 30
 testing granularity, 41
 understanding for testing, 24–25

Seams
 bringing code under test with, 145
 callbacks, observers, listeners, and
 notifiers as, 150–154
 in common lock synchronization, 176–181
 complex situations rich in, 170
 dependency injection as, 147–150
 direct calls as, 146–147
 factories as, 156–159
 identifying in mess of code, 25
 identifying in race conditions, 172
 injection points as, 181
 logging as, 159–163
 overview of, 145
 registries as, 154–156
 reproducing race conditions, 172
 for test-driven Java, 198–200

Semantically handled edge cases, code
 coverage in, 12

Separating concerns, in testing, 56–57

Sequencing issues, race conditions as, 171

ServerException, 133

ServiceFailureException, 130–134

SetAccessible() method, Java reflection, 115

Setters
 constructor bootstrapping model, 65
 dependency injection using, 147–149
 testing simple, 66–67

Shades of testing, 40–41

Shared constants, 67–70, 85–87

Shared fixture, testing Singleton's purpose,
 124–125

Simplicity, software hygiene, 6–7

Single Responsibility Principle, 119

Singleton object, 120

Singleton pattern
 as difficult-to-test design, 119–120, 122
 intent, 121
 removing Singleton nature, 124
 testing class purpose, 123–127
 testing Singleton nature, 122–123
 testing strategy, 121–122
 using factories as seams, 156–158

Slash (/) character, regular expressions, 84

Small tests, writing, 55–56

Smalltalk, 110

Smoke testing, 41

- Software
 - archeology and legacy code, 217
 - engineering and craftsmanship. *See* Engineering and craftsmanship
- SOLID principles, 119
- Solution, software hygiene, 6–7
- Spring framework
 - dependency injection, 147–150
 - reflection-based scheduling callbacks, 151–152
- Spy feature
 - defined, 100
 - in test-driven Java, 201–203
 - verifying value-added by function, 214
- SpyOn () method, 72, 100
- Standards, boundary conditions from, 32
- Starting
 - conceptual framework for, 25–26
 - overview of, 23
 - state and behavioral testing, 26–27
 - to test or not to test, 27–28
 - testing alternate paths, 29–30
 - testing data permutations, 30–35
 - testing defects, 35
 - testing error paths, 30
 - testing “happy path,” 29
 - testing recipe, 28–35
 - understanding scope, 24–25
- State testing, 26–27
- Statement coverage, 13–14
- Static analysis tools, 7
- Static checkers
 - disabling for test code, 48
 - enforcing naming convention, 49
 - unit testing under constraints of, 10
- Statistical verification, parallelism and, 187–189
- String handling
 - cautionary note, 90
 - exact verification by value, 85–87
 - exact verification with formatted results, 88–89
 - implementing callbacks through, 151–152
 - verification by containment, 81–83
 - verification by pattern, 83–85
 - verifying strings, 81
- Stubs
 - defined, 100
 - in test-driven Java, 199–200
 - testing with null object implementations, 101–103
- Style, software hygiene, 6–7
- Subsets of system, testing purpose of code for, 26
- Supervisory control, parallelism and, 184–187
- Sustainability, testing code for, 28
- Swallowed exception, 140
- Synchronization
 - common lock, 176–181
 - injecting, 181
- Syntactic marker, naming conventions, 49
- Syntax, software hygiene, 6–7
- System tests
 - based on purpose of code, 25–26
 - economics of, 28
- T**
- Tasks, testing thread, 173–175
- Taxonomy, of software hygiene, 5
- Technologies, influencing testing scope, 24
- Templates, and factory seams, 156–159
- Test Doubles
 - collaborators and, 100
 - creating substitutes for collaborators, 77
 - defined, 24
- Test subpackages, Java, 106
- Test through the interface, 55
- Test-driven development (TDD)
 - Java example of. *See* Test-driven Java
 - purpose of tests over time, 41
 - as test- first approach, 8–10
- Test-driven Java
 - bootstrapping, 196–197
 - cutting cord, 198–199
 - exercising options, 203–204
 - first functionality, 197–198
 - ghost protocol, 200–203
 - moving downstream, 204–206
 - moving to multiples, 199–200
 - overview of, 195–196
 - retrospective, 207
- Testing
 - code coverage tools for, 7
 - craftsmanship assisted by, 8–10
 - inefficiencies of retesting, 5
 - methodologies for automated, 5–6
 - principles and practices, 1–2
 - separate intent from implementation, 18–19, 21–22
 - unit test for coverage, 10–15
 - unit test under code checker constraints, 10
- Testing principles
 - avoid test code in production, 51–52
 - couple inputs to outputs, 48–49
 - craft tests well, 47–51
 - do not test framework, 58–60
 - minimize coupling, 53–54
 - overview of, 47
 - prefer complete over partial verification, 55
 - prefer minimal, fresh, transient fixtures, 54
 - remove code for simplicity, 58
 - separate your concerns, 56–57

- Testing principles, *continued*
 - sometimes test framework, 60
 - use available facilities, 55
 - use naming conventions, 49–51
 - use unique values, 57
 - verify intent over implementation, 52
 - write smaller tests, 55–56
 - Testing recipe
 - overview of, 28
 - test alternate paths, 29–30
 - test data permutations, 30–35
 - test defects, 35
 - test error paths, 30
 - test “happy path,” 29
 - visualization of, 29
 - TestNG, 93–94, 133
 - Test-only interfaces, adjusting visibility with, 111–112
 - Test-then-set race conditions, 167–168
 - Third-party code, testing, 60
 - Thread class, Java, 173–175
 - Threads
 - in common lock synchronization, 176–181
 - and deadlocks, 169
 - debugger APIs handling multiple, 189–191
 - overview of, 166
 - and race conditions, 167–169
 - reproducing race conditions, 170–173
 - statistical verification, 187–189
 - synchronization through injection and, 181–184
 - testing thread task directly, 173–175
 - using supervisory control, 184–187
 - Thread-safe code, 167
 - Thread.State.BLOCKED state, Java, 180
 - Throwable class, verifying exception payload, 134–137
 - Toothpaste refactoring, 213
 - Toothpaste testing, 213
 - Top-level domains, DNS, 32
 - Toyota Production System, 5
 - Transient fixtures, tests, 54
 - Transparency, 41
 - Try/catch, 217
 - Types, exception, 130–133, 141–143
- U**
- UI component, JavaScript testing of, 193–194
 - Unified exception handling, 130
 - Unique values, testing with, 57
 - Unit testing
 - adjusting visibility with `friend` keyword, 113–114
 - of alternate paths, 30
 - Automated Testing Triangle for, 9
 - under code checker constraints, 10
 - coupling in, 43–44
 - for coverage, 10–15
 - defined, 26
 - economics of, 28
 - granularity of, 41
 - of happy path, 29
 - mathematical view of, 65–66
 - objections to, 10
 - reducing coupling in, 53–54
 - testing defects, 35
 - value-added perspective on, 25–26
 - URI class, test-driven Java, 201–203
 - Usage of object, representational encapsulation, 38–39
- V**
- Value constraints, boundary conditions in, 31–32
 - Value-added perspective, unit testing, 25–26
 - Value-added tax (VAT), 25–26
 - Values
 - exact verification of strings by, 85–87
 - older registries allowing storage of, 154
 - testing using unique, 57
 - verifying error conditions with return, 129–130
 - Variables, locally redefining test, 70–71
 - Verification
 - complete vs. partial, 55
 - by injection, 77–79
 - statistical, 187–189
 - using callbacks, 154
 - Verification, error condition
 - checking return value, 129–130
 - exception design and, 140–143
 - exception instance, 137–140
 - exception message, 132–134
 - exception payload, 134–137
 - exception type, 130–132
 - overview of, 129
 - Verification, string testing
 - cautionary note for, 90
 - by containment, 81–83
 - with formatted results, 88–89
 - by pattern, 83–85
 - by value, 85–87
 - Visibility, adjusting
 - becoming `friend-ly`, 113–114
 - breaking it down, 108–109
 - changing access levels, 109–111
 - coerced access via reflection, 114–115
 - declarative scope, 116–117

- improving testability by, 75–77, 105
- naming the unnamed, 112–113
- packaging tests with code, 105–107
- test-only interfaces, 111–112

Visualization, of testing recipe, 29

W

Waste, Lean manufacturing minimizing, 5

WebRetriever. *See* Test-driven Java

White-box tests

- coupling and, 42–43
- defined, 40–41
- testing transparency, 41

- Worked examples, 193–194
- Write-read race conditions, 167
- Write-write race conditions, 168–169

X

XUnit Test Patterns, Meszaros, 54

Y

Yan Framework, 147